

# **Data Science I with python**

**STAT 303-1-Sec20**

Lizhen Shi

# Table of contents

<b>Preface</b>	<b>11</b>
<b>I Getting started</b>	<b>12</b>
<b>1 Setting up your environment with VS Code</b>	<b>13</b>
1.1 Learning Objectives . . . . .	13
1.2 Why Choose Visual Studio Code (VS Code)? . . . . .	13
1.3 Quick Start: Setting Up Python & VS Code . . . . .	14
1.4 Full Setup: Reliable Python Environment in VS Code . . . . .	15
1.5 Jupyter Notebooks in VS Code . . . . .	17
1.6 Independent Study: Practice Your Python Data Science Setup . . . . .	18
1.6.1 Step-by-Step Instructions . . . . .	18
1.7 Reference . . . . .	19
<b>2 Python Environments and Package Management</b>	<b>20</b>
2.1 Learning Objectives . . . . .	20
2.2 Data Science packages in Python . . . . .	20
2.3 Python Virtual Environments . . . . .	21
2.4 Install Data Science Packages Within Your Environment . . . . .	21
2.4.1 How to Install Packages . . . . .	22
2.4.2 Backing Up and Sharing Your Environment . . . . .	23
2.5 Dependency Conflicts . . . . .	27
2.5.1 Why do dependency conflicts happen? . . . . .	27
2.5.2 How to avoid dependency conflicts . . . . .	27
2.6 How to Resolve Conflicts and Manage Environments . . . . .	27
2.6.1 Use <code>conda</code> for Robust Environment Management . . . . .	27
2.6.2 Use <code>poetry</code> for Modern Python Projects . . . . .	29
2.6.3 Difference Between <code>conda</code> and <code>poetry</code> . . . . .	30
2.6.4 Modern Python Package Managers: <code>pip</code> , <code>conda</code> , <code>poetry</code> , <code>uv</code> . . . . .	31
2.7 Independent Study: Practice Environment Setup with <code>pip</code> . . . . .	32
2.7.1 Instructions . . . . .	32
<b>3 Enhancing Workflow in Jupyter Notebooks</b>	<b>34</b>
3.1 Learning Objectives . . . . .	34

3.2	Magic Commands . . . . .	34
3.2.1	What are Magic Commands? . . . . .	34
3.2.2	Line Magic Commands . . . . .	35
3.2.3	Cell Magic Commands . . . . .	38
3.3	Shell Commands in Jupyter Notebooks . . . . .	38
3.3.1	Using Shell Commands . . . . .	39
3.4	Installing Required Packages Within Your Notebook . . . . .	42
3.5	Specifying Your File Path . . . . .	45
3.5.1	Absolute Path . . . . .	45
3.5.2	Relative Path . . . . .	46
3.5.3	Methods to Specify File Paths in Windows . . . . .	46
3.5.4	File paths in macOS and Linux . . . . .	48
3.6	Interacting with the OS and Filesystem . . . . .	48
3.7	Independent Study . . . . .	50
3.7.1	Setting Up Your Data Science Workspace . . . . .	50
3.7.2	Navigate between directories and understand .. and .. in paths . . . . .	51
3.7.3	Check the existence of a directory or file before creating it . . . . .	52
<b>II</b>	<b>Python refresher</b>	<b>54</b>
<b>4</b>	<b>Python Basics</b>	<b>55</b>
4.1	Python Variables . . . . .	55
4.1.1	Rules for variable names . . . . .	55
4.1.2	Dynamic Typing in Python Variables . . . . .	56
4.1.3	Multiple Variable Assignment in Python . . . . .	56
4.2	Built-in data types . . . . .	57
4.3	Python Standard Library . . . . .	58
4.4	Third-Party Packages (Libraries) . . . . .	60
4.4.1	Importing a Library . . . . .	60
4.5	User-defined functions . . . . .	61
4.5.1	Creating and using functions . . . . .	61
4.5.2	Variable scope: Local and global Variables . . . . .	62
4.5.3	Function Arguments . . . . .	63
4.6	Control Flow . . . . .	65
4.6.1	Conditional Statements . . . . .	65
4.6.2	Loops . . . . .	71
4.7	Object Oriented Programming . . . . .	77
<b>5</b>	<b>Data structures</b>	<b>80</b>
5.1	Primitives vs. Containers . . . . .	80
5.2	Core Built-in Data Structures . . . . .	80
5.2.1	Sequences (ordered, indexable) . . . . .	80

5.2.2	Sets (unordered, unique) . . . . .	81
5.2.3	Mappings (key–value) . . . . .	81
5.3	Iterables vs. Iterators . . . . .	82
5.4	Common, Efficient Operations . . . . .	82
5.4.1	Comprehensions . . . . .	82
5.4.2	Membership & Lookups . . . . .	84
5.4.3	Unpacking . . . . .	84
5.4.4	Sorting in Python Iterables . . . . .	86
5.4.5	Lambda Functions in Python . . . . .	88
5.4.6	<code>enumerate()</code> . . . . .	90
5.4.7	<code>zip()</code> . . . . .	91
5.4.8	Common Functions for Iterables . . . . .	92
<b>III</b>	<b>Core library foundations</b>	<b>93</b>
<b>6</b>	<b>NumPy</b>	<b>94</b>
6.1	Learning Objectives . . . . .	94
6.2	Getting Started . . . . .	95
6.3	Data Types in NumPy . . . . .	95
6.3.1	Upcasting . . . . .	96
6.4	Why do we need NumPy arrays? . . . . .	97
6.4.1	Numpy arrays are memory efficient: Homogeneity and Contiguous Memory Storage . . . . .	97
6.4.2	NumPy arrays are fast . . . . .	98
6.5	Basics of NumPy Arrays . . . . .	99
6.5.1	Creating NumPy Arrays . . . . .	99
6.5.2	Array Attributes: . . . . .	100
6.6	Array Indexing and Slicing . . . . .	101
6.6.1	Array Indexing . . . . .	101
6.6.2	Array Slicing . . . . .	103
6.6.3	Modify Sub-Arrays through Slicing . . . . .	104
6.6.4	Combining Indexing and Slicing . . . . .	104
6.6.5	<code>np.where()</code> and <code>np.select()</code> . . . . .	104
6.6.6	<code>np.argmin()</code> and <code>np.argmax()</code> . . . . .	105
6.7	Array Operations . . . . .	106
6.7.1	Arithmetic operations . . . . .	106
6.7.2	Comparison and Logical Operation . . . . .	107
6.7.3	Aggregate Functions: <code>np.sum()</code> , <code>np.mean()</code> , <code>np.min()</code> , <code>np.max()</code> . . . . .	108
6.8	Array Reshaping . . . . .	110
6.8.1	<code>reshape()</code> . . . . .	110
6.8.2	<code>flatten()</code> and <code>ravel()</code> . . . . .	111
6.8.3	<code>resize()</code> . . . . .	112

6.8.4 <code>transpose()</code> or <code>T</code>	112
6.9 Arrays Concatenating	112
6.9.1 <code>np.concatenate()</code>	113
6.9.2 <code>np.vstack()</code> and <code>np.hstack()</code>	114
6.10 Vectorization in NumPy	115
6.10.1 Introduction to Vectorization	115
6.10.2 Understanding Vectorized Operations with Examples	116
6.10.3 NumPy Vectorization Vs Python for Loop	116
6.10.4 Broadcasting and Its Role in Vectorization	117
6.10.5 Matrix Multiplication in NumPy with Vectorization	118
6.11 Converting Between NumPy Arrays and pandas DataFrames	120
6.11.1 Why Conversion Matters	120
6.11.2 Converting a NumPy Array to a pandas DataFrame	120
6.11.3 Converting a pandas DataFrame to a NumPy Array	121
6.11.4 Preserving Index and Column Information	122
6.11.5 Practice User Cases	122
6.11.6 Best Practices	123
6.11.7 Comparison Table	124
6.12 Random Number Generation in NumPy	124
6.12.1 Key Functions for Random Number Generation in NumPy	124
6.13 Independent Practice:	126
6.13.1 Practice exercise 1	126
6.13.2 Practice exercise 2:	128
6.13.3 Practice exercise 3	131
6.13.4 Practice exercise 4:	134
6.13.5 Practice exercise 5	135
<b>7 Pandas</b>	<b>137</b>
7.1 Introduction	137
7.2 Pandas data structures - Series and DataFrame	138
7.3 Creating a Pandas Series / DataFrame	139
7.3.1 Creating a Series/DataFrame from Python List or Dictionary	139
7.4 Creating a Series/DataFrame by Reading Data from a File	141
7.5 Attributes and Methods of a Pandas DataFrame	142
7.5.1 Attributes of a Pandas DataFrame	142
7.5.2 Methods of a Pandas DataFrame	146
7.6 Data manipulations with Pandas	151
7.6.1 Sub-setting data	151
7.6.2 Setting and Resetting indices	152
7.6.3 Dropping a column	155
7.6.4 Adding a column	156
7.6.5 Renaming a column	156
7.6.6 Sorting data	157

7.6.7	Ranking data . . . . .	158
7.7	Arithematic operations within and between DataFrames . . . . .	158
7.7.1	Arithematic operations within a DataFrame . . . . .	159
7.7.2	Arithematic operations between DataFrames . . . . .	160
7.7.3	Arithmetic Between DataFrame and Series (Broadcasting) . . . . .	162
7.7.4	Converting a DataFrame to a NumPy Array . . . . .	163
7.8	Advanced DataFrame Operations . . . . .	165
7.8.1	<code>apply()</code> : Row-wise, or Column-wise . . . . .	166
7.8.2	<code>map()</code> : Element-wise . . . . .	167
7.8.3	Key Differences Between <code>map()</code> and <code>apply()</code> . . . . .	168
7.9	Introduction to Lambda Functions in Pandas . . . . .	168
7.9.1	Syntax of Lambda Functions: . . . . .	169
7.9.2	Key Features . . . . .	169
7.10	Understanding <code>inplace=True</code> and <code>inplace=False</code> . . . . .	170
7.11	Case study . . . . .	171
7.12	Independent Practice: . . . . .	177
7.12.1	Practice exercise 1 . . . . .	177
7.12.2	Practice exercise 2 . . . . .	178
7.12.3	Practice exercise 3 . . . . .	180
<b>8</b>	<b>Reading Data</b>	<b>181</b>
8.1	Types of data - structured and unstructured . . . . .	181
8.2	Reading a <i>csv</i> file with <i>Pandas</i> . . . . .	182
8.2.1	Using the <code>read_csv</code> function . . . . .	182
8.2.2	Data Overview . . . . .	183
8.3	Data Selection and Filtering . . . . .	185
8.3.1	Extracting Column(s) . . . . .	185
8.3.2	Extracting Row(s) . . . . .	187
8.3.3	Extracting Subsets of Rows and Columns . . . . .	189
8.3.4	Finding minimum/maximum of a column . . . . .	194
8.3.5	Finding the top <i>n</i> minimum/maximum values of a column . . . . .	195
8.4	Writing data to a <i>.csv</i> file . . . . .	196
8.5	Reading other data formats - txt, html, json . . . . .	197
8.5.1	Reading <i>.txt</i> files . . . . .	197
8.5.2	Reading HTML data . . . . .	198
8.5.3	Reading JSON data . . . . .	200
8.5.4	Reading data from a URL in Python . . . . .	201
8.6	Independent Study . . . . .	205
8.6.1	Practice exercise 1: Reading <i>.csv</i> data . . . . .	205
8.6.2	Practice exercise 2: Reading <i>.txt</i> data . . . . .	206
8.6.3	Practice exercise 3: Reading HTML data . . . . .	207
8.6.4	Practice exercise 4: Reading JSON data . . . . .	207

<b>9 More on Pandas</b>	<b>208</b>
9.1 Types of Data . . . . .	208
9.1.1 Available Data Types and Associated Built-in Functions . . . . .	209
9.1.2 Data Type Filtering . . . . .	209
9.1.3 Data Type Conversion . . . . .	210
9.1.4 Working with <code>datetime</code> Data . . . . .	211
9.1.5 Working with <code>object</code> Data . . . . .	213
9.1.6 Working with <code>numerical</code> Data . . . . .	221
<b>10 Introduction to Data Visualization</b>	<b>224</b>
10.1 The Art of Visualization: Choosing the Right Plot Type . . . . .	224
10.1.1 Data Classification for Visualization . . . . .	224
10.1.2 The Role of Visualization in Data Analysis . . . . .	225
10.1.3 Summary . . . . .	227
10.2 Visulization Tools . . . . .	227
10.2.1 Basic Plotting with Pandas . . . . .	227
10.2.2 Data Plotting with Matplotlib Pyplot Interface . . . . .	234
10.2.3 Plotting with Seaborn . . . . .	253
10.3 Independent Study . . . . .	272
10.3.1 Practice exercise 1 . . . . .	272
10.3.2 Practice exercise 2 . . . . .	273
10.3.3 Practice exercise 3 . . . . .	273
10.3.4 Practice exercise 4 . . . . .	274
<b>11 Advanced Data Visualization</b>	<b>275</b>
11.1 Matplotlib Plotting Interfaces . . . . .	275
11.1.1 Pyplot interface and OOP Interface . . . . .	275
11.1.2 Plot a simple figure using two interfaces . . . . .	276
11.1.3 Pyplot Interface . . . . .	278
11.2 Plotting with Object-Oriented Interface of Matplotlib . . . . .	279
11.2.1 Matplotlib Object Anatomy . . . . .	279
11.2.2 Matplotlib Object Hierarchy . . . . .	280
11.2.3 Creating Complex Plots with Multiple Subplots . . . . .	284
11.2.4 Advanced Customization with Matplotlib's Object-Oriented Interface .	288
11.2.5 <code>pyplot</code> : a convenience wrapper around the object-oriented interface .	295
11.3 Creating Subplots with Seaborn . . . . .	295
11.3.1 Using <code>Facetgrid</code> . . . . .	296
11.3.2 Using <code>Pairplot</code> . . . . .	299
11.4 Geosptial Plotting . . . . .	302
11.4.1 Static Plots with GeoPandas . . . . .	302
11.4.2 Dataset: Bicycle Sharing in Chicago . . . . .	305
11.4.3 Adding the divvy station to the plot . . . . .	306
11.4.4 Change the chicago shapefile . . . . .	307

11.4.5	Interactive Plotting . . . . .	309
11.4.6	Adding the divvy station on the interactive Folium.Map . . . . .	311
11.5	Independent Study . . . . .	312
11.5.1	Multiple plots in a single figure using Seaborn . . . . .	312
<b>IV</b>	<b>From messy to insight</b>	<b>314</b>
<b>12</b>	<b>Data Cleaning and Preparation</b>	<b>315</b>
12.1	Handling missing data . . . . .	315
12.1.1	Identifying missing values in a dataframe . . . . .	316
12.1.2	Types of Missing Values . . . . .	318
12.1.3	Methods for Handling missing values . . . . .	319
12.2	Outlier detection and handling . . . . .	333
12.2.1	Outlier detection . . . . .	333
12.2.2	Common Methods for Handling outliers . . . . .	338
12.3	Data binning . . . . .	340
12.3.1	Key reasons for binning: . . . . .	340
12.3.2	Common binning methods: . . . . .	342
12.4	Dummy / Indicator variables . . . . .	355
12.4.1	Purpose: . . . . .	355
12.4.2	When to use dummy variables . . . . .	355
12.4.3	How to Create Dummy variables . . . . .	355
12.4.4	Using <code>drop_first</code> to Avoid Multicollinearity . . . . .	357
12.5	Independent Study . . . . .	357
12.5.1	Practice exercise 1 . . . . .	357
12.5.2	Practice exercise 2 . . . . .	358
<b>13</b>	<b>Data Grouping and Aggregation</b>	<b>359</b>
13.1	Grouping by a single column . . . . .	360
13.1.1	Syntax of <code>groupby()</code> . . . . .	360
13.1.2	Attributes and methods of the <i>GroupBy</i> object . . . . .	361
13.2	Data aggregation within groups . . . . .	364
13.2.1	Common Aggregation Functions . . . . .	364
13.2.2	Multiple aggregations and Custom aggregation using <code>agg()</code> . . . . .	365
13.2.3	Multiple aggregate functions on multiple columns . . . . .	368
13.2.4	Distinct aggregate functions on multiple columns . . . . .	368
13.3	Grouping by Multiple Columns . . . . .	370
13.3.1	Basic Syntax for Grouping by Multiple Columns . . . . .	370
13.3.2	Understanding Hierarchical (Multi-Level) Indexing . . . . .	371
13.3.3	Subsetting Data in a Hierarchical Index . . . . .	372
13.3.4	Grouping by multiple columns and aggregating multiple variables . . . . .	373

13.4 Advanced Operations within groups: <code>apply()</code> , <code>transform()</code> , and <code>filter()</code> . . . . .	375
13.4.1 Using <code>apply()</code> on groups . . . . .	375
13.4.2 Using <code>transform()</code> on Groups . . . . .	376
13.4.3 Using <code>filter()</code> on Groups . . . . .	381
13.5 Sampling data by group . . . . .	381
13.6 <code>corr()</code> : Correlation by group . . . . .	383
13.7 <code>pivot_table()</code> . . . . .	383
13.8 <code>crosstab()</code> . . . . .	385
13.8.1 Basic Usage of <code>crosstab()</code> . . . . .	385
13.8.2 Using <code>crosstab()</code> with Aggregation Functions . . . . .	386
13.9 Independent Study . . . . .	386
13.9.1 Practice exercise 1 . . . . .	386
13.9.2 Practice exercise 2 . . . . .	389
<b>14 Data Reshaping and Enrichment</b>	<b>391</b>
14.1 Data Reshaping . . . . .	391
14.1.1 Why Data Reshaping is Important: . . . . .	391
14.1.2 Wide vs. Long Data Format in Data Reshaping . . . . .	392
14.1.3 Pivoting “long” to “wide” using <code>pivot_table()</code> . . . . .	392
14.1.4 Melting “wide” to “long” using <code>melt()</code> . . . . .	396
14.1.5 Stacking and Unstacking using <code>stack()</code> and <code>unstack()</code> . . . . .	397
14.1.6 Transposing using <code>.T</code> attribute . . . . .	403
14.1.7 Summary of Common Reshaping Functions . . . . .	403
14.1.8 Converting from Multi-Level Index to Single-Level Index DataFrame . . . . .	404
14.2 Data Enriching . . . . .	405
14.2.1 Combining based on common keys: <code>merge()</code> . . . . .	406
14.2.2 Combining based on Indices: <code>join()</code> . . . . .	410
14.2.3 Stacking vertically or horizontally: <code>concat()</code> . . . . .	412
14.2.4 Missing values after data enriching . . . . .	413
14.3 Independent Study . . . . .	414
14.3.1 Practice exercise 1 . . . . .	414
14.3.2 Practice exercise 2 . . . . .	414
14.3.3 Merging datasets with <i>similar</i> values in the <i>key</i> column . . . . .	417
<b>Appendices</b>	<b>419</b>
<b>A Assignment A</b>	<b>419</b>
Instructions . . . . .	419
A.1 GDP per Capita (35 pts) . . . . .	420
A.1.1 List Comprehension . . . . .	420
A.1.3 Dictionary . . . . .	421

A.2	Student Survey (40 pts) . . . . .	422
A.2.1	Marriage age responses . . . . .	422
A.2.2	Majors/Minors: Nested lists of student majors. . . . .	423
A.2.4	Starting salary expectations . . . . .	424
A.3	Starbucks Drinks (20 pts) . . . . .	424
<b>B</b>	<b>Datasets &amp; Templates</b>	<b>426</b>

# Preface

Welcome to **STAT 303-1 (Data Science with Python I), Section 20** at Northwestern University.

This book is designed to support your journey into data science, blending foundational concepts with hands-on practice. It builds on the original work of **Professor Arvind Krishna**, whose materials inspired the structure and spirit of this resource. The content has been thoughtfully updated to meet the evolving needs of students and the goals of the course.

In this first course, you will:

- Learn to use essential Python libraries for data science, including but not limited to **NumPy**, **Pandas**, and **Matplotlib**. These three are the main focus, but you will also be introduced to other useful libraries and tools as needed for real-world data analysis.
- Develop skills in **Exploratory Data Analysis (EDA)**, transforming messy, real-world datasets into meaningful insights through visualization, summarization, and pattern discovery

The book is organized to guide you step-by-step:

- **Chapters 1–3:** Set up your coding environment in VS Code, understand Python environments, and enhance your workflow
- **Chapters 4–5:** Review key concepts from STAT 201. If you need a refresher, visit the [STAT 201 ebook](#).
- New material begins in **Chapter 5 – NumPy**

Throughout the quarter, this resource will be updated and refined to improve clarity, depth, and alignment with our teaching objectives.

As a **living document**, your feedback and suggestions are always welcome. Contributions from students, instructors, and the broader academic community help make this book stronger and more effective.

If you have ideas for improving the textbook, please share your feedback or suggestions using our dedicated [Textbook Improvement Form](#). Your input is invaluable in making this resource better for everyone.

Thank you for joining us on this learning adventure. We hope this book becomes a valuable companion as you build your skills and confidence in data science with Python.

# **Part I**

## **Getting started**

# 1 Setting up your environment with VS Code

```
<IPython.core.display.Image object>
```

## 1.1 Learning Objectives

Setting up your Python data science environment is the foundation for all your work in this course. A well-configured setup will save you time, prevent errors, and make your workflow smoother. Whether you're new to VS Code or Python, these steps will help you build a reliable environment for coding, analysis, and reproducibility.

By the end of this chapter, you will be able to:

- Install and configure **Visual Studio Code (VS Code)** for Python and Jupyter Notebooks.
- Create and manage a **virtual environment** for this course.
- Install essential **Python libraries** for data science.
- Verify that your environment is working properly.

## 1.2 Why Choose Visual Studio Code (VS Code)?

Choosing the right editor can make a big difference in your productivity and learning experience. VS Code is a top choice for data science and Python development because it combines power, flexibility, and ease of use.

- **Free and Lightweight**

VS Code is free, open-source, and runs efficiently on most systems without being resource-heavy.

- **Cross-Platform**

Works consistently on Windows, macOS, and Linux.

- **Rich Python Support**

With the official Python extension, you get:

- Syntax highlighting and IntelliSense (auto-completion, code hints).
- Built-in debugging tools.
- Easy integration with virtual environments and Jupyter Notebooks.

- **Integrated Jupyter Notebook Support**

You can open and run .ipynb notebooks directly in VS Code.

- **Customizable and Extensible**

Large marketplace of extensions (e.g., formatting, linting, Git, Docker, AI tools).

- **Version Control Integration**

Built-in Git and GitHub support for managing and sharing code.

- **Unified Workflow**

One place for editing, running, debugging, documenting, and version-controlling Python code.

Whether you’re just starting out or working on advanced projects, VS Code provides a modern, unified environment for all your coding needs.

## 1.3 Quick Start: Setting Up Python & VS Code

VS Code is a versatile editor. To use it for data science, you’ll complete a few extra setup steps (e.g., Python & Jupyter extensions). Follow this step-by-step setup guide to get started quickly and avoid common pitfalls.

### Step 1: Install Python

- Download and install the latest version from [python.org](https://www.python.org).
- On Windows, check the box “**Add Python to PATH.**” This makes Python available in your terminal.

### Step 2: Install VS Code & Extensions

- Download VS Code from [code.visualstudio.com](https://code.visualstudio.com).
- Open VS Code, go to the Extensions panel (**Ctrl+Shift+X**), and install:
  - **Python (Microsoft)**
  - **Jupyter (Microsoft)**

### Step 3: Create a Course Folder

- Make a folder for your notebooks (e.g., `datasci` on Desktop or Documents).
- Open this folder in VS Code.

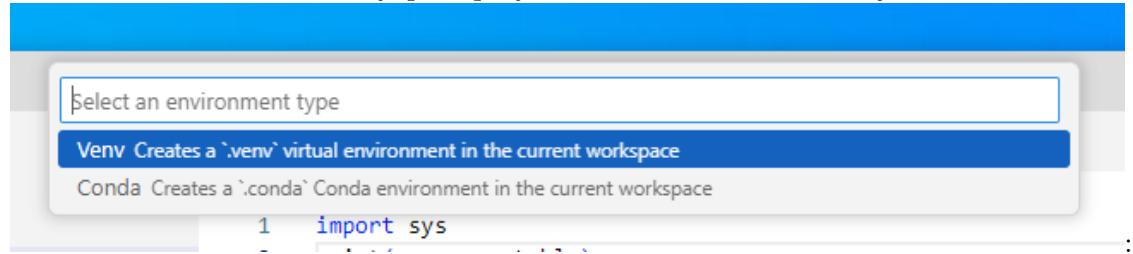
#### Step 4: Create Your First Notebook

- In VS Code, go to **File → New File → Save as → test.ipynb**.

- Add a code cell:

```
print("hello world")
```

- Run the cell. VS Code may prompt you to select or create a Python environment.



- If prompted, create a new environment using the GUI (recommended for beginners).
- Make sure the kernel (top right) shows **.venv (Python 3.xx.x)**.
- If you see `hello world` printed, your setup is working!

**If everything works, you can skip the Full Setup below.**

If you run into issues, continue with the detailed setup and troubleshooting steps in the next section.

This quick start helps you get coding fast, but the full setup ensures your environment is robust and reproducible for all course projects.

## 1.4 Full Setup: Reliable Python Environment in VS Code

If the quick start didn't work, or you want a more robust setup, follow these step-by-step instructions to ensure your Python environment is reproducible and ready for data science.

#### Step 1: Install Python

- Go to the [official Python website](#) and download the latest version for your operating system.
- **Important:** During installation, check the box “**Add Python to PATH**” so Python is available in your terminal.
- After installation, restart your terminal in VS Code.

- Verify installation by running:

```
python --version
```

You should see the installed Python version.

### Step 2: Install VS Code Extensions

- Open VS Code.
- Go to Extensions (**Ctrl+Shift+X**).
- Install:
  - **Python** (by Microsoft)
  - **Jupyter** (by Microsoft)

### Step 3: Set Up Your Workspace

- Create a new folder for your course work (e.g., STAT303-1 on Desktop or Documents).
- Open the folder in VS Code (**File > Open Folder**).

### Step 4: Create a Notebook for Your Work

- In VS Code, go to **File > New File** and select **Jupyter Notebook**.
- Save the file as `your_notebook.ipynb`.
- Add a code cell:

```
print("hello world")
```

- Make sure to select the `.venv` kernel for your notebook (see next step).

### Step 5A: Create a Virtual Environment (GUI Method)

- Run the cell. If prompted, create a Python environment using `venv` in the current workspace.
- Wait for Jupyter to start and install necessary components (e.g., `ipykernel`).
- After installation, ensure the selected kernel in the top right of the notebook says `.venv (Python 3.xx.x)`.
- If not, select the `.venv` kernel manually.

*If you successfully created the environment using the GUI, skip Step 5B.*

### Step 5B: Create a Virtual Environment (Command Line Method)

- If not prompted to create your environment, use the terminal:
  - Open the terminal in VS Code.

- Run:

```
python -m venv .venv
```

- Activate the environment:

- \* On Windows:

```
.\.venv\Scripts\activate
```

- \* On macOS/Linux:

```
source .venv/bin/activate
```

- You should see (.venv) at the start of your terminal prompt.

- Install essential packages:

```
pip install numpy pandas matplotlib
```

## Step 6: Verify Your Setup

- Run the code cell again. You should see `Hello world` printed out.
- In the terminal, check your Python version and installed packages:

```
python --version  
pip list
```

## Troubleshooting Tips

- If the kernel does not show `.venv`, restart VS Code and ensure the environment is activated.
- Make sure the Python and Jupyter extensions are installed and enabled.
- If you see errors, check that your notebook is saved with the `.ipynb` extension and the correct kernel is selected.

A reliable environment is essential for reproducible data science. These steps will help you avoid common issues and set yourself up for success throughout the course.

## 1.5 Jupyter Notebooks in VS Code

After setting up your environment, you can take full advantage of Jupyter Notebooks directly in VS Code. This integration lets you:

- Create, edit, and run `.ipynb` notebooks without leaving VS Code
- Use interactive code cells for data analysis, visualization, and documentation
- Access rich outputs (plots, tables, markdown) inline with your code

- Switch kernels to use different Python environments for each notebook
- Debug, refactor, and version-control your notebooks with built-in tools

### Getting Started:

- Open or create a notebook file (`.ipynb`) in VS Code
- Make sure the correct Python kernel/environment is selected (top right corner)
- Add code and markdown cells as needed
- Run cells individually or all at once

### Learn More:

- Explore the official guide: [Jupyter Notebooks in VS Code](#)

Jupyter Notebooks in VS Code provide a powerful, flexible workflow for data science and Python development.

## 1.6 Independent Study: Practice Your Python Data Science Setup

**Goal:** Get hands-on experience creating a reproducible Python environment using `pip` and `.venv` in VS Code.

### 1.6.1 Step-by-Step Instructions

#### Step 1: Create Your Workspace

- Make a folder named `STAT303_1` for your course work.
- Open this folder in VS Code.
- Create a new notebook: `setup_test.ipynb`.
- Add a code cell:

```
print("Hello, VS Code & .venv!")
```

- Run the cell. If prompted, create/select a Python environment (choose `.venv`).

#### Step 2: Create the `.venv` Environment

- If VS Code does not prompt you, open the terminal in VS Code.
- Run:

```
python -m venv .venv
```

- Activate the environment:

```
.\venv\Scripts\activate
```

- Install essential packages:

```
pip install numpy pandas matplotlib
```

### Step 3: Verify Environment Consistency

- In the terminal, check your Python version:

```
python --version
```

- In the notebook, run:

```
import sys
print(sys.executable)
```

- Make sure the Python path in the notebook matches the one in your terminal. This confirms you are using the same environment.

### Step 4: Troubleshooting Tips

If you run into issues (e.g., kernel not showing .venv, code cell errors): - Restart VS Code. - Deactivate/reactivate .venv. - Make sure **Python & Jupyter** extensions are installed. - Save your notebook with the .ipynb extension. - If the kernel is missing, use the kernel picker (top right) to select .venv.

#### Extra:

- Try installing another package (e.g., `seaborn`) and import it in a new cell to test.
- Practice switching kernels to see how environments affect your code.

## 1.7 Reference

- [Getting Started with VS Code](#)
- [Jupyter Notebooks in VS Code](#)
- [Install Python Packages](#)
- [Managing environments with conda](#)

# 2 Python Environments and Package Management

```
<IPython.core.display.Image object>
```

## 2.1 Learning Objectives

Managing Python environments and packages is essential for reproducible, error-free data science. Understanding these concepts will help you avoid common pitfalls and collaborate more effectively. This chapter will guide you through the tools and best practices for setting up, sharing, and troubleshooting Python environments.

By the end of this chapter , you will be able to:

- Explain why virtual environments are important.
- Install essential data science packages in your Python environment.
- Create a `requirements.txt` file and use it to quickly recreate an environment.
- Recognize why dependency conflicts happen.
- Describe how tools like `conda` or `poetry` can help resolve dependency conflicts.

## 2.2 Data Science packages in Python

Python has a rich ecosystem of packages that make data analysis easier and more powerful. In this course, you will install and use several core packages:

- **NumPy** → numerical computing and array operations
- **pandas** → data manipulation and analysis

- **Matplotlib** → data visualization
- **Seaborn** → statistical data visualization (built on top of Matplotlib)

These packages give you the essential tools to load, clean, analyze, and visualize data.

Before using them, you need to install them in your Python environment.

In your previous VS Code setup lesson, you created a Python environment using `.venv` and selected it as the kernel to run your notebook. This created a folder called `.venv`, which contains:

- A dedicated Python interpreter for your project
- A `Lib` folder where all packages you install are stored

Any packages you install (e.g., with `pip install ...`) will only affect this environment, keeping your project isolated from others.

Using virtual environments is considered best practice for Python development, especially in data science projects. This isolation helps you:

- Avoid version conflicts between packages
- Keep project dependencies separate
- Make your code more reproducible and shareable

Before we dive in, let's clarify what a Python environment is and why isolation matters.

## 2.3 Python Virtual Environments

A Python virtual environment is an **isolated workspace** with its own Python interpreter and its own set of installed packages.

This isolation makes it easy to manage dependencies for different projects and prevents package version conflicts.

### Why use virtual environments?

- Keep each project's dependencies separate
- Avoid breaking code by accidentally upgrading/downgrading packages
- Share your code with others and ensure it works the same way on their machine

Virtual environments are a key tool for professional, reproducible Python workflows.

## 2.4 Install Data Science Packages Within Your Environment

Python's power for data science comes from its rich ecosystem of packages. These packages provide essential tools for scientific computing, data analysis, visualization, and machine learning.

Packages like `numpy`, `pandas`, and `matplotlib` are not included in the Python standard library—you need to install them in your environment before you can use them.

**Test your setup:** Add the following code to your `test.ipynb` notebook to check if your environment is ready:

```
import numpy as np
import pandas as pd
print("Setup complete!")
```

If you see a `ModuleNotFoundError`, it means the package is not installed in your current environment. Use `pip install` to add missing packages.

**Tip:** Always install packages inside your active virtual environment to keep your project isolated and reproducible.

## 2.4.1 How to Install Packages

There are two main ways to install data science packages in your environment:

### 2.4.1.1 Installing from the Terminal

- **Open a New Terminal:** Check that your terminal is using the same environment as your notebook kernel.
  - If you see `(.venv)` at the start of the prompt, your virtual environment is active and matches the notebook kernel.
  - If you see `(base)`, it means the base conda environment is active (common if you have Anaconda installed).
  - To confirm which Python is being used, run:
    - \* On macOS/Linux: `which python`
    - \* On Windows: `where.exe python`

**Note:** If you have both Anaconda and VS Code installed, environments can sometimes conflict. If the terminal and notebook use different environments, packages may be installed in the wrong place, causing import errors in your notebook.

- **Install packages with pip:**

```
pip install numpy pandas
```

#### 2.4.1.2 Installing from the Notebook

You can also install packages directly from a Jupyter Notebook cell using a magic command. This is convenient for quick installs without leaving the notebook interface.

- Add a new code cell and run:

```
pip install numpy pandas
```

**Best Practice:** Always make sure you are installing packages into the correct environment. Double-check your kernel and terminal environment before running installation commands.

#### 2.4.2 Backing Up and Sharing Your Environment

Reproducibility is a cornerstone of good data science. To ensure your code works for you, your collaborators, and on other machines, you need a way to recreate your Python environment exactly.

Backing up your environment makes it easy to share your work and avoid “it works on my machine” problems.

**How to back up your environment:**

Step 1: **Create a requirements.txt file**

- This file lists all installed packages and their versions, so anyone can recreate your environment.

Run the following command in your terminal to list all installed packages and their versions:

```
pip freeze
```

```
asttokens==2.4.1
colorama==0.4.6
comm==0.2.2
contourpy==1.3.0
cycler==0.12.1
debugpy==1.8.6
decorator==5.1.1
executing==2.1.0
fonttools==4.54.1
ipykernel==6.29.5
ipython==8.27.0
jedi==0.19.1
```

```
jupyter_client==8.6.3
jupyter_core==5.7.2
kiwisolver==1.4.7
matplotlib==3.9.2
matplotlib-inline==0.1.7
nest-asyncio==1.6.0
numpy==2.1.1
packaging==24.1
pandas==2.2.3
parso==0.8.4
pillow==10.4.0
platformdirs==4.3.6
prompt_toolkit==3.0.48
psutil==6.0.0
pure_eval==0.2.3
Pygments==2.18.0
pyparsing==3.1.4
python-dateutil==2.9.0.post0
pytz==2024.2
pywin32==306
pyzmq==26.2.0
six==1.16.0
stack-data==0.6.3
tornado==6.4.1
traitlets==5.14.3
tzdata==2024.2
wcwidth==0.2.13
Note: you may need to restart the kernel to use updated packages.
```

Using the redirection operator `>`, you can save the output of `pip freeze` to a `requirement.txt`. This file can be used to install the same versions of packages in a different environment.

```
pip freeze > requirement.txt
```

Note: you may need to restart the kernel to use updated packages.

Let's check whether the `requirement.txt` is in the current working directory

```
%ls
```

```
Volume in drive C is Windows
Volume Serial Number is A80C-7DEC
```

```
Directory of c:\Users\lsi8012\OneDrive - Northwestern University\FA24\303-1\test_env
```

```
09/27/2024  02:25 PM    <DIR>        .
09/27/2024  02:25 PM    <DIR>        ..
09/27/2024  07:44 AM    <DIR>        .venv
09/27/2024  01:42 PM    <DIR>        images
09/27/2024  02:25 PM            695 requirement.txt
09/27/2024  02:25 PM           21,352 venv_setup.ipynb
                           2 File(s)      22,047 bytes
                           4 Dir(s)   166,334,562,304 bytes free
```

### Step 2: Share the file

- Send the requirements.txt file to your collaborator, or save it for future use.

### Step 3: Recreate the environment elsewhere

- On a new machine or environment, run:

```
pip install -r requirements.txt
```

```
pip install -r requirement.txt
```

```
Requirement already satisfied: asttokens==2.4.1 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: colorama==0.4.6 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: comm==0.2.2 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: contourpy==1.3.0 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: cycler==0.12.1 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: debugpy==1.8.6 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: decorator==5.1.1 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: executing==2.1.0 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: fonttools==4.54.1 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: ipykernel==6.29.5 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: ipython==8.27.0 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: jedi==0.19.1 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: jupyter_client==8.6.3 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: jupyter_core==5.7.2 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: kiwisolver==1.4.7 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: matplotlib==3.9.2 in c:\users\lsi8012\onedrive - northwestern unive
Requirement already satisfied: matplotlib-inline==0.1.7 in c:\users\lsi8012\onedrive - northwestern unive
```

```
Requirement already satisfied: nest-asyncio==1.6.0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: numpy==2.1.1 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: packaging==24.1 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pandas==2.2.3 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: parso==0.8.4 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pillow==10.4.0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: platformdirs==4.3.6 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: prompt_toolkit==3.0.48 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: psutil==6.0.0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pure_eval==0.2.3 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: Pygments==2.18.0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pyparsing==3.1.4 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: python-dateutil==2.9.0.post0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pytz==2024.2 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pywin32==306 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: pyzmq==26.2.0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: six==1.16.0 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: stack-data==0.6.3 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: tornado==6.4.1 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: traitlets==5.14.3 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: tzdata==2024.2 in c:\users\lsi8012\onedrive - northwestern university
Requirement already satisfied: wcwidth==0.2.13 in c:\users\lsi8012\onedrive - northwestern university
Note: you may need to restart the kernel to use updated packages.
```

Once you run the `install` command, all packages listed in your `requirements.txt` file will be installed, matching the exact versions you used in your original environment.

This ensures your code runs the same way everywhere—whether on your computer, a collaborator’s machine, or a server.

### Why is this important?

- Guarantees everyone is using the same package versions, reducing errors and “works on my machine” problems.
- Makes it easy to set up your project on a new computer, server, or cloud environment.
- Simplifies troubleshooting and collaboration—everyone starts from the same setup.

### Pro Tip:

- Update your `requirements.txt` after installing or upgrading packages to keep it current.
- Consider version-controlling your `requirements.txt` file (e.g., with Git) so changes are tracked and shared with collaborators.

A well-maintained environment file is essential for reproducible, shareable, and professional data science projects.

## 2.5 Dependency Conflicts

A **dependency conflict** occurs when two or more packages in your Python environment require different or incompatible versions of the same library.

This can cause errors, unexpected behavior, or even break your code.

### 2.5.1 Why do dependency conflicts happen?

- Many Python packages rely on other packages (called *dependencies*) to work.
- If you install packages that depend on different versions of the same dependency, they may not work together.
- This is especially common in data science, where libraries evolve quickly and have complex interdependencies.

**Example:** Suppose you install two packages:

- PackageA requires numpy==1.24.0
- PackageB requires numpy==2.2.0

If you install both with pip, the **last specified version wins**. One of the packages may then fail because it cannot use the version of numpy that was actually installed.

### 2.5.2 How to avoid dependency conflicts

- Use **virtual environments** to isolate dependencies for each project.
- Check package requirements before installing new libraries.
- Use tools like **Poetry** or **Conda** to detect and resolve conflicts automatically.

## 2.6 How to Resolve Conflicts and Manage Environments

### 2.6.1 Use conda for Robust Environment Management

conda is a powerful tool for managing both Python environments and packages, especially in data science workflows. It helps you avoid and resolve dependency conflicts by handling package versions and system dependencies more effectively than pip alone.

## Why use conda?

- Create fully isolated environments for different projects
- Install packages and all their dependencies from the Anaconda repository
- Easily switch between environments for different tasks
- Export and share environment configurations for reproducibility

## How conda prevents dependency conflicts:

- When you install a package, conda automatically checks for compatible versions of all dependencies and installs them together.
- If a conflict is detected, conda will warn you, suggest solutions, or prevent incompatible installations.

## Essential conda commands:

- Create a new environment:

```
conda create --name myenv numpy pandas matplotlib
```

- Activate an environment:

```
conda activate myenv
```

- Install a package in an environment:

```
conda install seaborn
```

- List all environments:

```
conda env list
```

- Export environment configuration:

```
conda env export > environment.yml
```

- Recreate an environment from a file:

```
conda env create -f environment.yml
```

**Example:** Suppose you need to work on two projects that require different versions of `scikit-learn`. You can create two separate environments:

```
conda create --name projectA scikit-learn=0.24
conda create --name projectB scikit-learn=1.2
```

Each environment will have its own compatible dependencies, so you avoid conflicts.

**Summary:** Using `conda` is highly recommended for managing complex dependencies, system-level packages, and reproducible environments in data science.

## 2.6.2 Use poetry for Modern Python Projects

poetry is a modern tool for managing Python dependencies and packaging. It streamlines environment setup, ensures reproducibility, and makes sharing your project easy.

### Why use poetry?

- Automatically creates and manages virtual environments for each project
- Uses a `pyproject.toml` file to specify and lock project requirements
- Prevents dependency conflicts with a lock file (`poetry.lock`)
- Simplifies publishing packages to PyPI and sharing with others

### How poetry helps with dependency management:

- Resolves and installs compatible versions of all dependencies automatically
- Keeps your environment reproducible and up-to-date
- Makes it easy to add, update, or remove packages

### Essential poetry commands:

- Install poetry (if not already installed):

```
pip install poetry
```

- Create a new project:

```
poetry new myproject
```

- Add packages:

```
poetry add numpy pandas matplotlib
```

- Install all dependencies:

```
poetry install
```

- Run commands inside the environment:

```
poetry run python script.py
```

- Update dependencies:

```
poetry update
```

### Example workflow:

1. Create a new project:

```
poetry new ds_project
cd ds_project
```

2. Add dependencies:

```
poetry add numpy pandas matplotlib seaborn
```

3. Install all dependencies:

```
poetry install
```

4. Run your code inside the poetry-managed environment:

```
poetry run python your_script.py
```

**Summary:** `poetry` is highly recommended for modern Python projects where you want reliable dependency management, easy environment setup, and reproducible results.

### 2.6.3 Difference Between `conda` and `poetry`

Both `conda` and `poetry` are tools for managing Python environments and dependencies, but they have different strengths and use cases.

**conda:**

- Manages both Python environments and packages, including non-Python dependencies (e.g., C libraries, compilers)
- Works with multiple languages (Python, R, etc.)
- Uses the Anaconda repository, which includes many scientific and data science packages
- Great for data science workflows, especially when you need packages with compiled code or system-level dependencies
- Handles complex dependency resolution and environment isolation

**poetry:**

- Focuses on Python projects and packages
- Uses `pyproject.toml` and `poetry.lock` for reproducible builds
- Automatically creates and manages virtual environments
- Excellent for modern Python application development and publishing to PyPI
- Handles dependency resolution and version locking for Python packages only

**Key differences:**

- `conda` can install system-level and non-Python dependencies; `poetry` only manages Python packages
- `conda` environments can include packages from the Anaconda repository; `poetry` uses PyPI
- `poetry` is ideal for pure Python projects and reproducible builds; `conda` is better for scientific computing and mixed-language projects

### **When to use each:**

- Use `conda` if you need scientific libraries, compiled code, or non-Python dependencies
- Use `poetry` for modern Python projects, apps, and libraries where you want easy dependency management and publishing

### **Summary Table:**

Feature	conda	poetry
Language support	Python, R, more	Python only
Non-Python dependencies	Yes	No
Environment isolation	Yes	Yes
Dependency resolution	Excellent	Excellent
Reproducibility	Good	Excellent
Publishing to PyPI	No	Yes

Choose the tool that best fits your project needs!

#### **2.6.4 Modern Python Package Managers: pip, conda, poetry, uv**

Python package management has evolved rapidly, making it easier to install, update, and manage dependencies for any project.

- `pip` is the standard tool for installing Python packages from PyPI. It works well for most projects and is simple to use.
- `.venv` helps you create isolated environments so each project has its own dependencies.
- `conda` offers advanced environment and package management, especially for scientific computing and projects with non-Python dependencies.
- `poetry` provides modern dependency management, automatic environment creation, and reproducibility for Python projects.
- `uv` is a new, high-performance package manager written in Rust. It's designed for speed and supports modern workflows. Learn more: [uv GitHub page](#)

### **For this course:**

- You'll use `pip` and `.venv` for installing packages and managing your environment.
- As you tackle larger or more complex projects, consider exploring `conda`, `poetry`, and `uv` for better performance, reproducibility, and ease of use.

### **Summary:**

- Choose the tool that fits your workflow and project needs.

- Stay curious—new tools like uv are making Python development faster and more reliable.

A good package manager and environment strategy will save you time, prevent headaches, and make your code more reproducible and shareable.

## 2.7 Independent Study: Practice Environment Setup with pip

**Objective:** Build hands-on skills in creating, managing, and verifying Python environments and packages using pip.

### 2.7.1 Instructions

#### Step 1: Create Your Workspace

- Make a folder named `test_pip_env`.
- Open the folder in VS Code.

#### Step 2: Create a Virtual Environment

- create a `.venv` environment

#### Step 3: Install Required Packages

- With the environment active, install packages using pip:

```
pip install numpy pandas
```

#### Step 4: Export Your Environment Configuration

- Save your environment's package list to a file:
- `bash pip freeze > stat303_env.txt`

#### Step 5: Deactivate and Remove the Environment

- Deactivate the environment:

```
deactivate
```

- Remove the environment folder to clean up.

#### Step 6: Recreate the Environment from the Exported File

- Create a new environment and activate it.

- Install packages from your saved file:

```
pip install -r stat303_env.txt
```

- Verify that `numpy`, `pandas`, `matplotlib`, and (if installed) `scikit-learn` are present.

### Step 7: Run a Jupyter Notebook

- Create a notebook in the recreated environment.
- Import `numpy`, `pandas`, `matplotlib`, and `scikit-learn`.
- Print the versions of these packages to confirm setup.

This exercise will help you master environment management and reproducibility—key skills for any data scientist.

# 3 Enhancing Workflow in Jupyter Notebooks

```
<IPython.core.display.Image object>
```

## 3.1 Learning Objectives

In this lecture, we'll explore how to optimize your workflow in Jupyter notebooks by leveraging **magic commands** and **shell commands**, understanding **file paths**, and interacting with the **filesystem** using the **os** module.

By completing this lecture, you will be able to:

- Run shell commands directly within a notebook.
- Navigate and manipulate files and directories efficiently.
- Integrate external tools and scripts seamlessly into your workflow.

By mastering these techniques, you'll be able to work more efficiently and handle complex data science tasks with ease.

## 3.2 Magic Commands

### 3.2.1 What are Magic Commands?

Magic commands in Jupyter are shortcuts that extend the functionality of the notebook environment. There are two types of magic commands: - **Line magics**: Commands that operate on a single line. - **Cell magics**: Commands that operate on the entire cell.

In Jupyter notebooks, line magic commands are invoked by placing a single percentage sign (%) in front of the statement, allowing for quick, inline operations, while cell magic commands are denoted with double percentage signs (%%) at the beginning of the cell, enabling you to apply commands to the entire cell for more complex tasks.

You can access the full list of magic commands by typing:

```
# show all the available magic commands on the system  
%lsmagic
```

Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd %clear %
```

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%cmd %%code_wrap %%debug %%file %%html %%jupyter %%
```

Automagic is ON, % prefix IS NOT needed for line magics.

### 3.2.2 Line Magic Commands

#### 3.2.2.1 %time: Timing the execution of code

In data science, it is often crucial to evaluate the performance of specific code snippets or algorithms, and the `%time` magic command provides a simple and efficient way to measure the execution time of individual statements, helping you identify bottlenecks and optimize your code for better performance.

```
def my_dot(a, b):  
    """  
    Compute the dot product of two vectors  
  
    Args:  
        a (ndarray (n,)): input vector  
        b (ndarray (n,)): input vector with same dimension as a  
  
    Returns:  
        x (scalar):  
    """  
    x=0  
    for i in range(a.shape[0]):  
        x = x + a[i] * b[i]  
    return x
```

```
import numpy as np  
np.random.seed(1)  
a = np.random.rand(10000000) # very large arrays  
b = np.random.rand(10000000)
```

Let's use `%time` to measure the execution time of a single line of code.

```
# Example: Timing a list comprehension
%time np.dot(a, b)
```

```
CPU times: total: 15.6 ms
Wall time: 32.9 ms
```

```
2501072.5816813153
```

```
%time my_dot(a, b)
```

```
CPU times: total: 1.88 s
Wall time: 1.86 s
```

```
2501072.5816813707
```

To capture the output of `%time` (or `%timeit`), you cannot directly assign it to a variable as it's a magic command that prints the result to the notebook's output. However, you can use Python's built-in `time` module to manually time your code and assign the execution time to a variable.

Here's how you can do it using the `time` module:

```
import time
tic = time.time() # capture start time
c = np.dot(a, b)
toc = time.time() # capture end time

print(f"np.dot(a, b) = {c:.4f}")
print(f"Vectorized version duration: {1000*(toc-tic):.4f} ms ")

tic = time.time() # capture start time
c = my_dot(a,b)
toc = time.time() # capture end time

print(f"my_dot(a, b) = {c:.4f}")
print(f"loop version duration: {1000*(toc-tic):.4f} ms ")

del(a);del(b) #remove these big arrays from memory
```

```
np.dot(a, b) =  2501072.5817
Vectorized version duration: 2.9922 ms
my_dot(a, b) =  2501072.5817
loop version duration: 2004.3225 ms
```

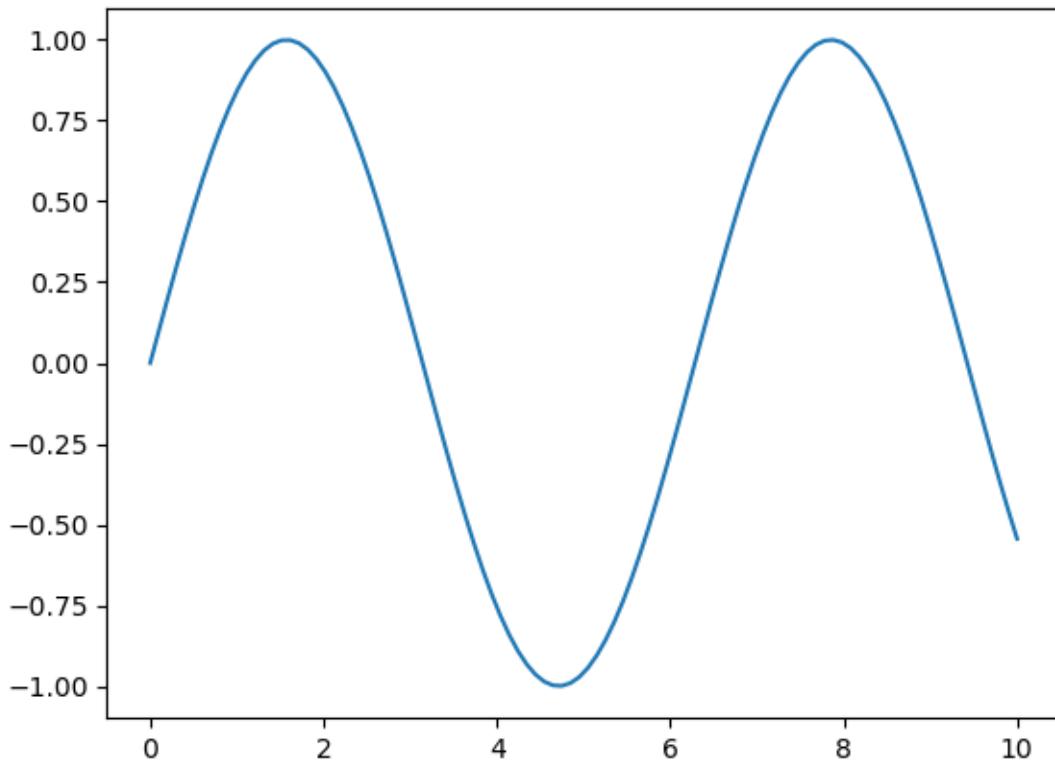
### 3.2.2.2 %matplotlib inline: Displaying plots inline

This command allows you to embed plots within the notebook.

```
# Example: Using %matplotlib inline to display a plot
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.plot(x, y);
```



### 3.2.3 Cell Magic Commands

A cell magic command in Jupyter notebook has to be the first line in the code cell

#### 3.2.3.1 %%time: Timing cell execution

This cell magic is useful for measuring the execution time of an entire cell.

```
%%time
# Example: Timing a cell with matrix multiplication

A = np.random.rand(1000, 1000)
B = np.random.rand(1000, 1000)
C = np.dot(A, B)
```

```
CPU times: total: 219 ms
Wall time: 23 ms
```

Question: there are several timing magic commands that can be confusing due to their similarities, they are `%time`, `%timeit`, `%%time`, and `%%timeit`. Do your own research on the differences among them

## 3.3 Shell Commands in Jupyter Notebooks

### What are Shell Commands?

Shell commands let you interact directly with your computer's operating system from within a Jupyter notebook. This means you can manage files, check your environment, and run system tools without leaving your notebook. To run a shell command in Jupyter, start the line with an exclamation mark (!). For example, `!ls` lists files on macOS/Linux, while `!dir` does the same on Windows.

**How does it work?** - Python code is executed by the IPython kernel inside the notebook. - Shell commands (lines starting with !) are sent to your computer's shell (like Bash, Command Prompt, or PowerShell), not to Python. - This separation means you can use both Python and shell commands in the same notebook, but they run in different environments.

### 3.3.1 Using Shell Commands

#### 3.3.1.1 Print the current working directory

To see where your notebook is running, use: - On macOS/Linux: !pwd - On Windows: !cd

This helps you understand where files will be saved or loaded from.

```
!cd
```

```
c:\Users\lsi8012\Documents\Courses\FA24\DataScience_Intro_python_fa24_Sec20_21
```

#### 3.3.1.2 !ls (!dir on Windows): Listing files and directories

To view all files and folders in your current directory, use a shell command:

- On macOS/Linux: !ls
- On Windows: !dir

This is useful for quickly checking what data, scripts, or notebooks are available in your workspace. If you want to see hidden files (those starting with a dot), use !ls -a on macOS/Linux or !dir /a on Windows.

**Tip:** If you get an error, double-check that you are using the correct command for your operating system.

```
!dir
```

```
Volume in drive C is Windows  
Volume Serial Number is A80C-7DEC
```

```
Directory of c:\Users\lsi8012\Documents\Courses\FA24\DataScience_Intro_python_fa24_Sec20_21
```

03/12/2025	10:46 AM	<DIR>	.
03/12/2025	10:46 AM	<DIR>	..
10/20/2024	04:13 PM	<DIR>	.github
10/20/2024	04:13 PM		11 .gitignore
10/20/2024	04:13 PM	<DIR>	.ipynb_checkpoints
10/20/2024	04:13 PM		8 .nojekyll
10/20/2024	04:24 PM	<DIR>	.quarto
03/12/2025	10:29 AM		6,933,867 Advanced_Data_Visualization.html
11/02/2024	03:12 PM		8,688,961 Advanced_Data_Visualization.ipynb

03/12/2025	10:31 AM	<DIR>	Advanced_Data_Visualization_files
10/20/2024	04:13 PM		9,325 Assignment 1 (Reading data).ipynb
10/20/2024	04:13 PM		14,013 Assignment 2 (NumPy).ipynb
10/20/2024	04:13 PM		18,098 Assignment 3 (Pandas).ipynb
10/20/2024	04:13 PM		22,821 Assignment 4 (Data Visualization).ipynb
10/20/2024	04:13 PM		96,997 Assignment 5 (Data cleaning and preparation).ipynb
10/20/2024	04:13 PM		17,038 Assignment 6 (Data wrangling).ipynb
10/20/2024	04:13 PM		14,220 Assignment 7 (Data aggregation).ipynb
03/12/2025	10:29 AM		137,354 Assignment A.html
10/20/2024	04:19 PM		43,769 Assignment A.ipynb
03/12/2025	10:29 AM		60,301 Assignment B.html
10/20/2024	04:19 PM		14,820 Assignment B.ipynb
03/12/2025	10:29 AM		60,966 Assignment C.html
10/20/2024	04:19 PM		18,169 Assignment C.ipynb
03/12/2025	10:29 AM		2,748,253 Assignment D.html
10/27/2024	11:43 AM		22,209 Assignment D.ipynb
03/12/2025	10:29 AM		68,850 Assignment E.html
11/19/2024	02:00 PM		19,393 Assignment E.ipynb
03/12/2025	10:29 AM		53,070 Assignment F.html
11/19/2024	02:08 PM		13,742 Assignment F.ipynb
10/20/2024	04:13 PM		14,782 Assignment G.ipynb
10/20/2024	04:13 PM		40,794 authors.jpg
10/20/2024	04:13 PM		7,562 chck.csv
10/20/2024	04:13 PM		4,299 co.csv
10/20/2024	04:13 PM		5,647 coords.csv
10/20/2024	04:13 PM		51,194 cover.png
10/20/2024	04:13 PM		651,392 Data aggregation inclass.ipynb
03/12/2025	10:29 AM		225,805 Data aggregation.html
12/14/2024	02:41 PM		300,468 Data aggregation.ipynb
03/12/2025	10:31 AM	<DIR>	Data aggregation_files
03/12/2025	10:29 AM		204,720 Data cleaning and preparation.html
11/10/2024	11:35 AM		1,050,426 Data cleaning and preparation.ipynb
03/12/2025	10:31 AM	<DIR>	Data cleaning and preparation_files
03/12/2025	10:29 AM		172,605 Data visualization.html
10/21/2024	09:53 AM		1,628,169 Data visualization.ipynb
03/12/2025	10:31 AM	<DIR>	Data visualization_files
11/10/2024	11:41 AM		367,702 Data wrangling copy.ipynb
03/12/2025	10:29 AM		185,198 Data wrangling.html
11/11/2024	09:53 AM		425,911 Data wrangling.ipynb
03/12/2025	10:31 AM	<DIR>	Data wrangling_files
11/25/2024	01:01 PM	<DIR>	Datasets
03/12/2025	10:29 AM		31,229 Datasets.html
10/20/2024	04:13 PM		1,156 Datasets.ipynb

10/20/2024	04:13 PM	37,147	datatypes.png
10/20/2024	04:13 PM	163,888	data_structures-Copy1.ipynb
03/12/2025	10:28 AM	288,147	data_structures.html
10/20/2024	04:13 PM	178,946	data_structures.ipynb
03/12/2025	10:29 AM	120,990	data_types_in_pandas.html
12/01/2024	11:00 PM	103,025	data_types_in_pandas.ipynb
03/12/2025	10:31 AM	<DIR>	data_types_in_pandas_files
10/20/2024	04:13 PM	5,834	dpop.csv
10/20/2024	04:13 PM	705,800	ds_image.jpg
11/22/2024	03:25 PM	<DIR>	example_directory
10/20/2024	04:13 PM	<DIR>	hello_files
11/25/2024	12:27 PM	<DIR>	images
10/20/2024	04:13 PM	457,384	inClass_Data_wrangling.ipynb
03/12/2025	10:31 AM	24,666	index.aux
03/12/2025	10:28 AM	32,316	index.html
03/12/2025	10:31 AM	70,189	index.log
03/12/2025	10:31 AM	1,041,600	index.pdf
03/11/2025	05:28 PM	1,251	index.qmd
03/12/2025	10:31 AM	1,462,894	index.tex
03/12/2025	10:31 AM	0	index.toc
10/20/2024	04:13 PM	10	intro.qmd
03/12/2025	10:28 AM	144,381	Introduction_to_Python_and_Jupyter_Notebooks.html
10/20/2024	04:13 PM	64,982	Introduction_to_Python_and_Jupyter_Notebooks.ipynb
10/20/2024	04:13 PM	361,557	Introduction-to-Data-Science-with-Python.pdf
10/20/2024	04:13 PM	1,115	In_class_exercise1.ipynb
10/20/2024	04:13 PM	<DIR>	In_class_exercise1_files
10/20/2024	04:13 PM	20,566	LICENSE.txt
10/20/2024	04:13 PM	1,086	movies_sample_data.csv
10/20/2024	04:13 PM	0	movie_data.json
10/20/2024	04:13 PM	242,761	movie_ratings.csv
10/20/2024	04:13 PM	109,910	NumPy_copy.ipynb
03/12/2025	10:29 AM	241,607	NumPy.html
11/24/2024	10:04 AM	113,396	NumPy.ipynb
10/20/2024	04:13 PM	23,179	NU_Stat_logo.png
03/12/2025	10:29 AM	269,968	Pandas.html
10/20/2024	04:13 PM	284,987	Pandas.ipynb
10/20/2024	04:13 PM	32,813,480	party_edited.csv
10/20/2024	04:13 PM	1,361	quarto_yml_file_Fall_2022.txt
10/20/2024	04:13 PM	6,496	questions_test - Copy.json
03/12/2025	10:29 AM	261,769	Reading_data.html
11/22/2024	08:49 PM	279,118	Reading_data.ipynb
10/20/2024	04:13 PM	243,254	Reading_data.html
10/20/2024	04:13 PM	<DIR>	Reading_Data_files

```

10/20/2024 04:13 PM      189 README.md
10/20/2024 04:13 PM      428 references.bib
10/20/2024 04:13 PM      48 references.qmd
10/20/2024 04:13 PM      811 requirements.txt
10/20/2024 04:13 PM          0 requirements.txt.txt
10/20/2024 04:13 PM      112,248 rough.ipynb
10/20/2024 04:13 PM      86 sample.txt
03/12/2025 10:46 AM      1,019,165 search.json
03/12/2025 10:34 AM      <DIR>      site_libs
10/20/2024 04:13 PM      34,315,856 spotify_data.csv
10/20/2024 04:13 PM      23,534,086 spotify_text.txt
10/20/2024 04:13 PM      149,442 student_solutions-Copy1.ipynb
10/20/2024 04:13 PM      286,932 student_solutions.ipynb
10/20/2024 04:13 PM      <DIR>      student_solutions_files
10/20/2024 04:13 PM          63 summary.qmd
10/20/2024 04:13 PM      1,205 test_file.ipynb
10/20/2024 04:13 PM      5,216 test_file.pdf
10/20/2024 04:13 PM      <DIR>      test_file_files
11/22/2024 03:28 PM      <DIR>      test_folder
10/20/2024 04:13 PM          1,116 text.txt
10/20/2024 04:13 PM      66,988 Untitled.ipynb
10/20/2024 04:13 PM      19,849 Untitled1.ipynb
10/20/2024 04:13 PM      3,942 Untitled2.ipynb
10/20/2024 04:13 PM      6,085 Untitled3.ipynb
10/20/2024 04:13 PM      18,861 Untitled4.ipynb
03/12/2025 10:28 AM      87,254 venv_setup.html
11/22/2024 09:12 PM      37,627 venv_setup.ipynb
10/20/2024 04:13 PM      1,864 welcome.ipynb
10/20/2024 04:13 PM      65,157 wordle.txt
03/12/2025 10:34 AM      104,246 workflow_enhance.html
03/12/2025 10:47 AM      74,123 workflow_enhance.ipynb
03/12/2025 10:34 AM      <DIR>      workflow_enhance_files
03/11/2025 05:31 PM          1,368 _quarto.yml
106 File(s)   124,538,629 bytes
22 Dir(s)   49,998,643,200 bytes free

```

### 3.4 Installing Required Packages Within Your Notebook

When working in Jupyter notebooks, you often need to install new Python packages. There are two main ways to do this:

1. Shell Command:

- Use an exclamation mark (!) before the command:
  - `!pip install package_name`
- This runs the command in your system shell, which may not always install the package in the same environment as your notebook kernel.

## 2. Magic Command:

- Use a percent sign (%) before the command:
  - `%pip install package_name`
- This is Jupyter-specific and ensures the package is installed in the environment where your notebook is running.

### Best Practice:

- Prefer `%pip install package_name` in Jupyter notebooks to avoid confusion about which environment is being updated.
- If you use `!pip install`, double-check that the package is available in your notebook (sometimes you may need to restart the kernel).

### Example:

- `%pip install numpy` # Installs numpy in the notebook's environment
- `!pip install numpy` # Installs numpy using the system shell (may differ from notebook environment)

To guarantee that package installation happens in the same environment as your notebook kernel, use the Python executable associated with the running notebook. This avoids issues where packages are installed in a different environment than the one your code is using.

### How to do this:

- Use the following command to ensure pip installs to the correct environment:

```
import sys
!{sys.executable} -m pip install package_name
```

This method is especially useful if you have multiple Python environments or are unsure which one is active in your notebook.

```
import sys
!{sys.executable} -m pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\lsi8012\appdata\local\anaconda3\lib\site-pa
```

Unlike shell commands, the `%pip` magic command is designed specifically for Jupyter notebooks. It automatically installs packages into the exact environment your notebook kernel is using, reducing confusion and installation errors.

### Why use `%pip`?

- It guarantees that the installed package will be available in your current notebook session.
- No need to worry about mismatched environments or restarting the kernel after installation.

### Example:

```
%pip install numpy
```

This will install `numpy` directly into the notebook's environment, making it immediately available for import and use.

```
%pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\lsi8012\appdata\local\anaconda3\lib\site-pa  
Note: you may need to restart the kernel to use updated packages.
```

In most cases, you don't need to type `%` before `pip install numpy` because Jupyter's "automagic" feature is enabled by default. Automagic lets you use some magic commands without the `%` prefix, as long as there isn't a variable with the same name in your code.

### How automagic works:

- If you type `pip install numpy`, Jupyter will interpret it as `%pip install numpy` automatically.
- If you have a variable named `pip`, the command will use the variable instead, which can cause unexpected behavior.

### Best practice:

- Use `%pip install ...` for clarity and to avoid conflicts.
- Automagic is convenient, but explicit magic commands are safer in shared or complex notebooks.

**Tip:** Not all magic commands work with automagic—some still require the `%` or `%%` prefix.

```
pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\lsi8012\appdata\local\anaconda3\lib\site-pa  
Note: you may need to restart the kernel to use updated packages.
```

## 3.5 Specifying Your File Path

Understanding how to specify file paths in Python is essential for loading and saving data. File paths tell Python where to find or store your files, such as datasets, results, or scripts.

There are two main types of file paths: **absolute paths** and **relative paths**.

- **Absolute Path:** Gives the complete location of a file or folder from the root of your file system. It always points to the same place, no matter where your code is running.
- **Relative Path:** Specifies the location of a file or folder in relation to your current working directory. It is shorter and more flexible, making your code easier to share and reuse.

Choosing the right type of path helps avoid errors and makes your code more portable across different computers and operating systems.

### 3.5.1 Absolute Path

An **absolute path** provides the full address to a file or directory, starting from the root of your system. It does not depend on where your code is running.

**Example (Windows):**

```
# Absolute path example (Windows)  
file_path = r"C:\Users\Username\Documents\data.csv"
```

```
!conda env list
```

```
# conda environments:  
#  
base * C:\Users\lsi8012\AppData\Local\anaconda3
```

The path associated with each conda env is absolute path.

### 3.5.2 Relative Path

A **relative path** describes the location of a file or folder in relation to your current working directory. It is shorter, more flexible, and makes your code easier to share and run on different computers.

Relative paths are especially useful in projects with organized folder structures, because they allow you to move your code and data together without changing file references.

To find your current working directory in Python, you can use either a magic command or a shell command:

- Magic command: %pwd
- Shell command: !cd (Windows) or !pwd (macOS/Linux)

Knowing your current working directory helps you understand where Python will look for files when using relative paths.

```
# Magic command  
%pwd
```

```
'c:\\\\Users\\\\lsi8012\\\\Documents\\\\Courses\\\\FA24\\\\DataScience_Intro_python_fa24_Sec20_21'
```

```
# Shell command  
!cd
```

```
c:\\Users\\lsi8012\\Documents\\Courses\\FA24\\DataScience_Intro_python_fa24_Sec20_21
```

Example of a Relative Path:

```
# Example of relative path  
file_path = "sample.txt" # Relative to the current directory
```

The relative path `sample.txt` means that there is a file in the current working directory.

### 3.5.3 Methods to Specify File Paths in Windows

Specifying file paths correctly is crucial for avoiding errors and making your code portable. Windows uses backslashes (\) to separate folders, but in Python, a single backslash is an escape character (e.g., \n for newline), which can lead to mistakes if not handled properly.

Here are four reliable ways to specify file paths in Windows:

### **3.5.3.1 Method 1: Using Escaped Backslashes**

Use double backslashes (\\) to prevent Python from interpreting them as escape characters.

```
file_path = "C:\\Users\\Username\\Documents\\data.csv"
```

### **3.5.3.2 Method 2: Using Raw Strings**

Prefix the path with r to tell Python to treat backslashes as literal characters.

```
file_path = r"C:\\Users\\Username\\Documents\\data.csv"
```

### **3.5.3.3 Method 3: Using forward slashes (/)**

Python accepts forward slashes (/) in file paths, even on Windows. This is often the simplest and most portable method.

```
file_path = "C:/Users/Username/Documents/data.csv"
```

### **3.5.3.4 Method 4: Using os.path.join**

Use `os.path.join()` to build paths programmatically. This method automatically uses the correct separator for your operating system, making your code cross-platform.

Using these methods helps prevent bugs, makes your code easier to share, and ensures it works on different computers and operating systems.

```
import os
file_path = os.path.join("C:", "Users", "Username", "Documents", "data.csv")
```

This method works across different operating systems because `os.path.join` automatically uses the correct path separator (\ for Windows and /for Linux/Mac).

### 3.5.4 File paths in macOS and Linux

macOS and Linux use forward slashes (/) as path separators, which is exactly what Python expects by default. This means you can specify file paths directly, like `/Users/yourusername/Documents/data.csv`, without worrying about escape characters or compatibility issues.

**Why is this helpful?** - Forward slashes work seamlessly in Python on macOS, Linux, and even Windows. - You avoid common errors caused by backslashes (which are escape characters in Python). - Code written with forward slashes is portable and recommended for all platforms.

#### 3.5.4.1 Best Practices for File Paths in Data Science

- Prefer **relative paths** within your project folders—this makes your code portable and easy to share or move.
- Use **absolute paths** only for files outside your project or when you need a fixed location.
- Always check your **current working directory** before reading or writing files to avoid confusion and errors.
- Avoid hardcoding file paths directly in your code; use variables or configuration files for flexibility.
- For cross-platform compatibility, use **forward slashes (/)** in file paths or build paths programmatically with `os.path.join()`.
- Document your file structure and path conventions in your project README to help collaborators.
- When sharing code, test file paths on both Windows and macOS/Linux to ensure portability.

## 3.6 Interacting with the OS and Filesystem

In data science projects, you often work with data files (such as CSV, Excel, or JSON) stored in various folders. Managing these files efficiently is essential for reproducible workflows and organized projects.

The Python `os` module provides powerful tools to interact with your operating system and manage files and directories directly from your notebook. With `os`, you can:

- Check your current working directory
- List files and folders
- Create, rename, or delete directories
- Move between folders
- Check if files or folders exist before using them

Let's import the `os` module and explore some of its most useful functions for data science tasks.

```
import os
```

We can get the location of the current working directory using the `os.getcwd` function.

```
os.getcwd()
```

```
'c:\\\\Users\\\\lsi8012\\\\Documents\\\\Courses\\\\FA24\\\\DataScience_Intro_python_fa24_Sec20_21'
```

The command `os.chdir('..')` in Python changes the current working directory to the parent directory of the current one.

Note that `..` as the path notation for the parent directory is universally true across all major operating systems, including Windows, macOS, and Linux. It allows you to move one level up in the directory hierarchy, which is very useful when navigating directories programmatically, especially in scripts where directory traversal is needed.

```
os.chdir('..')
```

```
os.getcwd()
```

```
'c:\\\\Users\\\\lsi8012\\\\Documents\\\\Courses\\\\FA24'
```

`os.chdir()` is used to change the current working directory.

For example: `os.chdir('./week2')`

`./week2` is a relative path:

- `.` refers to the current directory.
- `week2` is a folder inside the current directory.

The `os.listdir()` function in Python returns a list of all files and directories in the specified path. If no path is provided, it returns the contents of the current working directory.

```
os.listdir()
```

```
['.ipynb_checkpoints',
 '303-1',
 '362',
 'DataScience_Intro_python_fa24_Sec20_21',
 'EDA.pdf',
 'test_folder',
 'UG TA info for Instructors.pdf']
```

Check whether a specific folder/file exist in the current working directory

```
'data' in os.listdir('..')
```

```
False
```

## 3.7 Independent Study

### 3.7.1 Setting Up Your Data Science Workspace

To reinforce and apply the skills from this lecture, complete the following hands-on tasks:

#### 1. Set Up Your Workspace

- Create a folder named `STAT303-1` to organize all course materials.
- Set up a dedicated `pip` environment for your coursework to keep dependencies isolated.
- Organize your files into subfolders for datasets, assignments, projects, quizzes, lectures, and exams.
- Use the `os` module or shell commands in your notebook to create these directories programmatically.

#### 2. Practice Magic Commands

- Use `%timeit` to measure the execution time of a simple Python function in your notebook.
- Explore `%lsmagic` to discover all available magic commands and experiment with a few.

#### 3. Run Shell Commands

- Use `!ls` (or `!dir` on Windows) to list the contents of the directories you created.
- Use `!pwd` (or `!cd` on Windows) to print your current working directory.

#### 4. Explore File Paths

- Write a Python script to navigate between directories using both relative and absolute paths.
- Use the `os` module to check if a directory or file exists before creating it, and handle cases where it already exists.

**Tip:** Document your process and any issues you encounter. This will help you troubleshoot and share your workflow with others.

### 3.7.2 Navigate between directories and understand . and .. in paths

In Python, you can use the `os` module to move between directories and understand how relative paths (`.` for current directory, `..` for parent directory) work. This is essential for organizing your files and writing portable code.

- `.` refers to the current directory.
- `..` refers to the parent directory.

The following code demonstrates:

- How to print your current directory
- How to get absolute paths for `.` and `..`
- How to create and navigate into a new folder
- How to move back to the parent directory
- How to use absolute paths for navigation

Try running the code below and observe how the working directory changes as you use `.` and `..`

```
import os

# Print the current directory
print("Current Directory:", os.getcwd()) # Get the current working directory

# Use '.' to refer to the current directory
current_dir = os.path.abspath(".") # Absolute path of the current directory
print("Using .'': Current Directory Path:", current_dir)

# Use '..' to refer to the parent directory
parent_dir = os.path.abspath("..") # Absolute path of the parent directory
print("Using '..': Parent Directory Path:", parent_dir)

# Create a new folder in the current directory and navigate into it
os.rmdir("test_folder") # Remove the folder if it already exists
os.mkdir("test_folder") # Create a new folder
print("Created 'test_folder' in:", os.path.abspath("."))

os.chdir("test_folder") # Change to the new folder
print("After Changing Directory (Relative to '.'): ", os.getcwd())

# Navigate back to the parent directory using '...'
os.chdir("../")
print("After Navigating Back to Parent Directory (Relative to '...'): ", os.getcwd())

# Navigate to the parent directory directly using an absolute path
os.chdir(parent_dir)
```

```
print("Using Absolute Path to Navigate to Parent Directory:", os.getcwd())
```

```
Current Directory: c:\Users\lsi8012\Documents\Courses\FA24
Using '.': Current Directory Path: c:\Users\lsi8012\Documents\Courses\FA24
Using '..': Parent Directory Path: c:\Users\lsi8012\Documents\Courses
Created 'test_folder' in: c:\Users\lsi8012\Documents\Courses\FA24
After Changing Directory (Relative to '.'):  c:\Users\lsi8012\Documents\Courses\FA24\test_fo
After Navigating Back to Parent Directory (Relative to '..'): c:\Users\lsi8012\Documents\Cou
Using Absolute Path to Navigate to Parent Directory: c:\Users\lsi8012\Documents\Courses
```

### 3.7.3 Check the existence of a directory or file before creating it

When working with files and folders in Python, it's important to check if they already exist before creating them. This prevents errors, avoids overwriting important data, and makes your code more robust and user-friendly.

- Use `os.path.exists(path)` to check if a file or directory exists.
- Use `os.makedirs(path)` to create a directory (it can create intermediate folders if needed).
- Use `open(path, 'w')` to create a file, but only after confirming it doesn't already exist.

**Why is this important?** - Prevents accidental overwriting of files. - Avoids errors when trying to create a directory that already exists. - Makes your scripts safer and more portable.

**Best practices:** - Always check for existence before creating or deleting files/folders. - Use clear messages to inform users about what your code is doing. - Handle both cases: when the file/folder exists and when it does not.

The code below demonstrates how to safely check and create directories and files. Try running it and observe the output messages for each scenario.

```
# Define directory and file paths
dir_path = "example_directory"
file_path = os.path.join(dir_path, "example_file.txt")

# Check and create directory
if not os.path.exists(dir_path):
    os.makedirs(dir_path)
    print(f"Directory '{dir_path}' created.")
else:
    print(f"Directory '{dir_path}' already exists.")
```

```
# Check and create file
if not os.path.exists(file_path):
    with open(file_path, "w") as f:
        f.write("This is a test file.") # Write some content to the file
        print(f"File '{file_path}' created.")
else:
    print(f"File '{file_path}' already exists.")
```

```
Directory 'example_directory' created.
File 'example_directory\example_file.txt' created.
```

## **Part II**

# **Python refresher**

# 4 Python Basics

```
<IPython.core.display.Image object>
```

Before diving into new material, this chapter provides a quick refresher on key Python concepts covered in STAT201, which is the prerequisite for this course. If you have not taken STAT201, or if you feel you need to strengthen your Python skills, please review chapters 3–8 in the [STAT201 book](#). A solid understanding of Python basics will help you succeed in this course and make it easier to follow the advanced topics ahead.

## 4.1 Python Variables

Variables are fundamental building blocks in Python—they allow you to store, update, and reference data throughout your code. Choosing clear and descriptive variable names makes your code easier to read, debug, and share with others.

### 4.1.1 Rules for variable names

- Variable names must start with a letter (a–z, A–Z) or an underscore (\_). They cannot begin with a number.
- Names can include letters, digits, and underscores (`a_variable`, `profit_margin`, `the_3_musketeers`).
- Variable names are case-sensitive: `score`, `Score`, and `SCORE` are all different variables.
- Avoid using Python reserved words (like `for`, `if`, `class`, etc.) as variable names.

**Tip:** Use descriptive names that reflect the purpose of the variable. This helps you and others understand your code at a glance.

Here are some examples of good variable names:

```
# Examples of valid variable names

a_variable = 23
is_today_Saturday = False
my_favorite_car = "Delorean"
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
```

If you use an invalid variable name, Python will raise a `SyntaxError` and stop running your code. Always follow the naming rules to avoid these errors and keep your code readable.

#### 4.1.2 Dynamic Typing in Python Variables

Python variables are dynamically typed, meaning you don't need to declare their type before using them. You can assign a value of any type to a variable, and even change its type later in your code:

```
x = 5      # x is an integer
x = "cat" # now x is a string
```

To check the type of a variable, use the built-in `type()` function:

```
type(x)
```

This flexibility makes Python easy to use, but it's important to keep track of your variable types to avoid confusion in your code.

```
# Print the variables defined above their values and their types
print("a_variable:", a_variable, "| type:", type(a_variable))
print("is_today_Saturday:", is_today_Saturday, "| type:", type(is_today_Saturday))
print("my_favorite_car:", my_favorite_car, "| type:", type(my_favorite_car))
print("the_3_musketeers:", the_3_musketeers, "| type:", type(the_3_musketeers))
```

```
a_variable: 23 | type: <class 'int'>
is_today_Saturday: False | type: <class 'bool'>
my_favorite_car: Delorean | type: <class 'str'>
the_3_musketeers: ['Athos', 'Porthos', 'Aramis'] | type: <class 'list'>
```

#### 4.1.3 Multiple Variable Assignment in Python

Python allows you to assign values to several variables at once in a single line. This technique is especially useful for initializing related variables and can make your code cleaner and more efficient.

**Example:**

```
color1, color2, color3 = "red", "green", "blue"
```

After this assignment: - `color1` is “red” - `color2` is “green” - `color3` is “blue”

This approach works for any number of variables, as long as the number of values matches the number of variable names.

```
color1, color2, color3 = "red", "green", "blue"  
print(color1)  
print(color3)
```

```
red  
blue
```

The same value can be assigned to multiple variables by chaining multiple assignment operations within a single statement.

```
color4 = color5 = color6 = "magenta"  
print(color4)  
print(color6)
```

```
magenta  
magenta
```

## 4.2 Built-in data types

Python has several built-in data types for storing different kinds of information in variables.

```
<IPython.core.display.Image object>
```

### Primitive Types

In Python, **integers**, **floats**, **booleans**, and **None** are often called *primitive data types* because they represent a single value.

### Container (Data Structure) Types

Types such as **strings**, **lists**, **tuples**, **sets**, and **dictionaries** are *containers* because they can hold multiple values (characters in a string, items in a list, key–value pairs in a dictionary, etc.). We’ll explore these container types in more detail in the next chapter.

### Identifying Types

You can check the type of any object using Python’s built-in `type()` function. For example:

```
print(type(42))          # int
print(type(3.14))         # float
print(type(True))          # bool
print(type(None))          # NoneType
print(type("hello"))        # str (a sequence / container of characters)
print(type([1, 2, 3]))      # list
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'NoneType'>
<class 'str'>
<class 'list'>
```

## 4.3 Python Standard Library

The [Python Standard Library](#) provides a wide range of modules and built-in functions that support everyday programming tasks. These tools allow you to write code that is both efficient and readable without reinventing common functionality.

**Examples of built-in functions:** - `print()`: Displays output to the screen. - `len()`: Returns the length of an object (like a list or string). - `type()`: Shows the type of a variable. - `sum()`: Adds up all items in an iterable (like a list). - `range()`: Generates a sequence of numbers, often used in loops.

You can explore more built-in functions and modules in the official documentation. Using these tools makes your code more readable and powerful.

### Example:

`range()`: The `range()` function returns a sequence of evenly-spaced integer values. It is commonly used in `for` loops to define the sequence of elements over which the iterations are performed.

Below is an example where the `range()` function is used to create a sequence of whole numbers upto 10:

```
print(list(range(1,10)))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Date and Time:

Python includes a powerful built-in module called `datetime` for working with dates and times. This module lets you create, manipulate, and format date/time objects easily.

- You can get the current date and time.
- You can perform arithmetic with dates (e.g., add days, subtract dates).
- You can format dates and times for display or parsing.

This is essential for tasks like timestamping data, scheduling, or analyzing time series.

```
import datetime as dt

#Defining a date-time object
dt_object = dt.datetime(2022, 9, 20, 11, 30, 0)
```

Information about date and time can be accessed with the relevant attribute of the `datetime` object.

```
dt_object.day, dt_object.year
```

(20, 2022)

## Formatting Dates and Times:

The `strftime` method in the `datetime` module lets you convert a `datetime` object into a readable string using custom formats. This is useful for displaying dates in a way that matches your needs (e.g., for reports, logs, or user interfaces).

Common format codes include: - `%Y`: 4-digit year (e.g., 2025) - `%m`: 2-digit month (01-12) - `%d`: 2-digit day (01-31) - `%H`: Hour (00-23) - `%M`: Minute (00-59) - `%S`: Second (00-59)

See the [Python documentation](#) for more formatting options.

```
dt_object.strftime('%m/%d/%Y')
```

'09/20/2022'

```
dt_object.strftime('%m/%d/%y %H:%M')
```

'09/20/22 11:30'

```
dt_object.strftime('%h-%d-%Y')
```

```
'Sep-20-2022'
```

## 4.4 Third-Party Packages (Libraries)

While Python has many useful [built-in functions](#) like `print()`, `abs()`, `max()`, and `sum()`, these are often not enough for data analysis. Third-party libraries extend Python's capabilities and are essential for scientific computing and data science.

**Popular libraries and their main uses:**

1. **NumPy**: Efficient numerical operations, arrays, and mathematical functions. Essential for scientific and data analysis tasks.
2. **Pandas**: Powerful data manipulation and analysis. DataFrames and Series make reading, cleaning, and transforming data easy.
3. **Matplotlib & Seaborn**: Data visualization. Matplotlib creates a wide range of plots; Seaborn builds on Matplotlib for attractive statistical graphics.
4. **SciPy**: Advanced scientific computing—optimization, integration, statistics, and more.
5. **Scikit-learn**: Machine learning—tools for preprocessing, classification, regression, clustering, and model evaluation.
6. **Statsmodels**: Statistical modeling and inference (focuses on explanation, not just prediction).

**How to use these libraries:** 1. **Install the libraries** : Covered by Chapter 2 and 3 2. **Import the libraries** in your Python script or Jupyter notebook. 3. **Use their functions and classes** to analyze data, visualize results, and build models.

These libraries form the foundation of modern data science workflows in Python. You will gain plenty of hands-on experience with each of them throughout this course sequence.

### 4.4.1 Importing a Library

Use the `import` keyword to bring a library into your Python code.

Example:

```
import numpy as np
```

Aliases like `np` make code shorter and easier to read.

```
import numpy as np
np.arange(8)

array([0, 1, 2, 3, 4, 5, 6, 7])
```

## Importing in Python: Key Styles

- Import a whole module:

```
import math
```

- Import specific items:

```
from random import randint
```

- Use an alias:

```
import pandas as pd
```

- Rename an imported item:

```
from os.path import join as join_path
```

Pick the style that fits your needs and keeps your code readable.

## 4.5 User-defined functions

A function is a reusable set of instructions that takes one or more inputs, performs some operations, and often returns an output. Indeed, while python's standard library and ecosystem libraries offer a wealth of pre-defined functions for a wide range of tasks, there are situations where defining your own functions is not just beneficial but necessary.

### 4.5.1 Creating and using functions

You can define a new function using the def keyword.

```
def say_hello():
    print('Hello there!')
    print('How are you?')
```

Note the round brackets or parentheses () and colon : after the function's name. Both are essential parts of the syntax. The function's *body* contains an indented block of statements. The statements inside a function's body are not executed when the function is defined. To execute the statements, we need to *call* or *invoke* the function.

```
say_hello()
```

```
Hello there!  
How are you?
```

```
def say_hello_to(name):  
    print('Hello ', name)  
    print('How are you?')
```

```
say_hello_to('Lizhen')
```

```
Hello Lizhen  
How are you?
```

```
name = input ('Please enter your name: ')  
say_hello_to(name)
```

```
Please enter your name: George  
Hello George  
How are you?
```

#### 4.5.2 Variable scope: Local and global Variables

**Local variable:** When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function. These types of variables are called local variables. For example,

```
def greet():  
    message = 'Hello' # local variable  
    print('Local', message)  
greet()
```

```
Local Hello
```

```
# print(message) # try to access message variable outside greet() function, uncomment this line
```

As `message` was defined within the function `greet()`, it is local to the function, and cannot be called outside the function.

**Global variable:** A variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created.

```
message = 'Hello' # declare global variable

def greet():
    print('Local', message) # declare local variable

greet()
print('Global', message)
```

```
Local Hello
Global Hello
```

### 4.5.3 Function Arguments

#### 4.5.3.1 Named Arguments

When calling functions with multiple arguments, using *named* arguments improves clarity and reduces mistakes. You can also split long function calls across multiple lines for readability.

**Example:**

```
def greet(name, message):
    print(f'{message}, {name}!')

greet(name="Alice", message="Hello")
```

Named arguments make your code easier to understand and maintain.

Here is an example:

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    return emi
```

```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
)
```

```
emi1
```

```
14567.19753389219
```

#### 4.5.3.2 Optional Arguments

Functions with optional arguments offer more flexibility in how you can use them. You can call the function with or without the argument, and if there is no argument in the function call, then a default value is used.

```
emi2 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12)
```

```
emi2
```

```
19119.4467632335
```

#### 4.5.3.3 \*args and \*\*kwargs

We can pass a variable number of arguments to a function using two special symbols in Python:

- **\*args** for variable-length **positional arguments**
- **\*\*kwargs** for variable-length **keyword arguments**

This is useful when you want your function to accept a variety of arguments.

```
def myFun(*args,**kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)

# Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('John',22,'cs',name="John",age=22,major="cs")
```

```
args: ('John', 22, 'cs')
kwargs: {'name': 'John', 'age': 22, 'major': 'cs'}
```

## 4.6 Control Flow

Control flow statements (like if, for, and while) let you decide how your code runs based on conditions and loops. Control flow in Python includes both conditional statements and iteration loops to manage program logic.

### 4.6.1 Conditional Statements

As in other languages, python has [built-in keywords](#) that provide conditional flow of control in the code.

#### 4.6.1.1 Branching with if, else and elif

One of the most powerful features of programming languages is *branching*: the ability to make decisions and execute a different set of statements based on whether one or more conditions are true.

##### The if statement

In Python, branching is implemented using the `if` statement, which is written as follows:

```
if condition:
    statement1
    statement2
```

The **condition** can be a value, variable or expression. If the condition evaluates to **True**, then the statements within the *if block* are executed. Notice the four spaces before **statement1**, **statement2**, etc. The spaces inform Python that these statements are associated with the **if** statement above. This technique of structuring code by adding spaces is called *indentation*.

**Indentation:** Python relies heavily on *indentation* (white space before a statement) to define code structure. This makes Python code easy to read and understand. You can run into problems if you don't use indentation properly. Indent your code by placing the cursor at the start of the line and pressing the **Tab** key once to add 4 spaces. Pressing **Tab** again will indent the code further by 4 more spaces, and press **Shift+Tab** will reduce the indentation by 4 spaces.

For example, let's write some code to check and print a message if a given number is even.

```
a_number = 34

if a_number % 2 == 0:
    print("We're inside an if block")
    print('The given number {} is even.'.format(a_number))
```

```
We're inside an if block
The given number 34 is even.
```

### The **else** statement

We may want to print a different message if the number is not even in the above example. This can be done by adding the **else** statement. It is written as follows:

```
if condition:
    statement1
    statement2
else:
    statement4
    statement5
```

If **condition** evaluates to **True**, the statements in the **if** block are executed. If it evaluates to **False**, the statements in the **else** block are executed.

```
if a_number % 2 == 0:
    print('The given number {} is even.'.format(a_number))
else:
    print('The given number {} is odd.'.format(a_number))
```

The given number 34 is even.

### The `elif` statement

Python also provides an `elif` statement (short for “else if”) to chain a series of conditional blocks. The conditions are evaluated one by one. For the first condition that evaluates to `True`, the block of statements below it is executed. The remaining conditions and statements are not evaluated. So, in an `if, elif, elif...` chain, at most one block of statements is executed, the one corresponding to the first condition that evaluates to `True`.

```
today = 'Wednesday'
```

```
if today == 'Sunday':
    print("Today is the day of the sun.")
elif today == 'Monday':
    print("Today is the day of the moon.")
elif today == 'Tuesday':
    print("Today is the day of Tyr, the god of war.")
elif today == 'Wednesday':
    print("Today is the day of Odin, the supreme diety.")
elif today == 'Thursday':
    print("Today is the day of Thor, the god of thunder.")
elif today == 'Friday':
    print("Today is the day of Frigga, the goddess of beauty.")
elif today == 'Saturday':
    print("Today is the day of Saturn, the god of fun and feasting.")
```

Today is the day of Odin, the supreme diety.

In the above example, the first 3 conditions evaluate to `False`, so none of the first 3 messages are printed. The fourth condition evaluates to `True`, so the corresponding message is printed. The remaining conditions are skipped. Try changing the value of `today` above and re-executing the cells to print all the different messages.

### Using `if, elif, and else` together

You can also include an `else` statement at the end of a chain of `if, elif...` statements. This code within the `else` block is evaluated when none of the conditions hold true.

```
a_number = 49
```

```

if a_number % 2 == 0:
    print('{} is divisible by 2'.format(a_number))
elif a_number % 3 == 0:
    print('{} is divisible by 3'.format(a_number))
elif a_number % 5 == 0:
    print('{} is divisible by 5'.format(a_number))
else:
    print('All checks failed!')
    print('{} is not divisible by 2, 3 or 5'.format(a_number))

```

```

All checks failed!
49 is not divisible by 2, 3 or 5

```

## Non-Boolean Conditions

Note that conditions do not necessarily have to be booleans. In fact, a condition can be any value. The value is converted into a boolean automatically using the `bool` operator. Any value in Python can be converted to a Boolean using the `bool` function.

Only the following values evaluate to `False` (they are often called *falsy* values):

1. The value `False` itself
2. The integer `0`
3. The float `0.0`
4. The empty value `None`
5. The empty text `" "`
6. The empty list `[]`
7. The empty tuple `()`
8. The empty dictionary `{}`
9. The empty set `set()`
10. The empty range `range(0)`

Everything else evaluates to `True` (a value that evaluates to `True` is often called a *truthy* value).

```

if '':
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')

```

```

The condition evaluated to False

```

```
if 'Hello':  
    print('The condition evaluated to True')  
else:  
    print('The condition evaluated to False')
```

The condition evaluated to True

```
if { 'a': 34 }:  
    print('The condition evaluated to True')  
else:  
    print('The condition evaluated to False')
```

The condition evaluated to True

```
if None:  
    print('The condition evaluated to True')  
else:  
    print('The condition evaluated to False')
```

The condition evaluated to False

### Nested conditional statements

The code inside an if block can also include an if statement inside it. This pattern is called **nesting** and is used to check for another condition after a particular condition holds true.

```
a_number = 15
```

```
if a_number % 2 == 0:  
    print("{} is even".format(a_number))  
    if a_number % 3 == 0:  
        print("{} is also divisible by 3".format(a_number))  
    else:  
        print("{} is not divisible by 3".format(a_number))  
else:  
    print("{} is odd".format(a_number))  
    if a_number % 5 == 0:  
        print("{} is also divisible by 5".format(a_number))  
    else:  
        print("{} is not divisible by 5".format(a_number))
```

```
15 is odd
15 is also divisible by 5
```

Notice how the `print` statements are indented by 8 spaces to indicate that they are part of the inner `if/else` blocks.

Nested `if, else` statements are often confusing to read and prone to human error.  
It's good to avoid nesting whenever possible, or limit the nesting to 1 or 2 levels.

### Shorthand if conditional expression

A frequent use case of the `if` statement involves testing a condition and setting a variable's value based on the condition.

Python provides a shorter syntax, which allows writing such conditions in a single line of code. It is known as a *conditional expression*, sometimes also referred to as a *ternary operator*. It has the following syntax:

```
x = true_value if condition else false_value
```

It has the same behavior as the following `if-else` block:

```
if condition:
    x = true_value
else:
    x = false_value
```

Let's try it out for the example above.

```
parity = 'even' if a_number % 2 == 0 else 'odd'

print('The number {} is {}'.format(a_number, parity))
```

```
The number 15 is odd.
```

### The `pass` statement

`if` statements cannot be empty, there must be at least one statement in every `if` and `elif` block. We can use the `pass` statement to do nothing and avoid getting an error.

```
a_number = 9

# please uncomment the code below and see the error message
# if a_number % 2 == 0:

# elif a_number % 3 == 0:
#     print('{} is divisible by 3 but not divisible by 2')
```

As there must be at least one statement within the `if` block, the above code throws an error.

```
if a_number % 2 == 0:
    pass
elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2'.format(a_number))
```

```
9 is divisible by 3 but not divisible by 2
```

## 4.6.2 Loops

### 4.6.2.1 while loops

Another powerful feature of programming languages, closely related to branching, is running one or more statements multiple times. This feature is often referred to as *iteration* or *looping*, and there are two ways to do this in Python: using `while` loops and `for` loops.

`while` loops have the following syntax:

```
while condition:
    statement(s)
```

Statements in the code block under `while` are executed repeatedly as long as the `condition` evaluates to `True`. Generally, one of the statements under `while` makes some change to a variable that causes the condition to evaluate to `False` after a certain number of iterations.

Let's try to calculate the factorial of 100 using a `while` loop. The factorial of a number `n` is the product (multiplication) of all the numbers from 1 to `n`, i.e.,  $1*2*3*\dots*(n-2)*(n-1)*n$ .

```

result = 1
i = 1

while i <= 10:
    result = result * i
    i = i+1

print('The factorial of 100 is: {}'.format(result))

```

The factorial of 100 is: 3628800

#### 4.6.2.2 Infinite Loops

Suppose the condition in a `while` loop always holds true. In that case, Python repeatedly executes the code within the loop forever, and the execution of the code never completes. This situation is called an infinite loop. It generally indicates that you've made a mistake in your code. For example, you may have provided the wrong condition or forgotten to update a variable within the loop, eventually falsifying the condition.

If your code is *stuck* in an infinite loop during execution, just press the “Stop” button on the toolbar (next to “Run”) or select “Kernel > Interrupt” from the menu bar. This will *interrupt* the execution of the code. The following two cells both lead to infinite loops and need to be interrupted.

```

# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i <= 100:
    result = result * i
    # forgot to increment i

```

```

# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i > 0 : # wrong condition
    result *= i
    i += 1

```

#### 4.6.2.3 break and continue statements

In Python, `break` and `continue` statements can alter the flow of a normal loop.

We can use the `break` statement within the loop's body to immediately stop the execution and *break* out of the loop. with the `continue` statement. If the condition evaluates to `True`, then the loop will move to the next iteration.

```
i = 1
result = 1

while i <= 100:
    result *= i
    if i == 42:
        print('Magic number 42 reached! Stopping execution..')
        break
    i += 1

print('i:', i)
print('result:', result)
```

```
Magic number 42 reached! Stopping execution..
i: 42
result: 1405006117752879898543142606244511569936384000000000
```

```
i = 1
result = 1

while i < 8:
    i += 1
    if i % 2 == 0:
        print('Skipping {}'.format(i))
        continue
    print('Multiplying with {}'.format(i))
    result = result * i

print('i:', i)
print('result:', result)
```

```
Skipping 2
Multiplying with 3
Skipping 4
```

```
Multiplying with 5
Skipping 6
Multiplying with 7
Skipping 8
i: 8
result: 105
```

In the example above, the statement `result = result * i` inside the loop is skipped when `i` is even, as indicated by the messages printed during execution.

**Logging:** The process of adding `print` statements at different points in the code (*often within loops and conditional statements*) for inspecting the values of variables at various stages of execution is called logging. As our programs get larger, they naturally become prone to human errors. Logging can help in verifying the program is working as expected. In many cases, `print` statements are added while writing & testing some code and are removed later.

Task: Guess the output and explain it.

```
# Use of break statement inside the loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

```
s
t
r
The end
```

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

```
s  
t  
r  
n  
g  
The end
```

#### 4.6.2.4 for loops

A `for` loop is used for iterating or looping over sequences, i.e., lists, tuples, dictionaries, strings, and *ranges*. For loops have the following syntax:

```
for value in sequence:  
    statement(s)
```

The statements within the loop are executed once for each element in `sequence`. Here's an example that prints all the elements of a list.

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']  
  
for day in days:  
    print(day)
```

```
Monday  
Tuesday  
Wednesday  
Thursday  
Friday
```

```
# Looping over a string  
for char in 'Monday':  
    print(char)
```

```
M  
o  
n  
d  
a  
y
```

```
# Looping over a dictionary
person = {
    'name': 'John Doe',
    'sex': 'Male',
    'age': 32,
    'married': True
}

for key, value in person.items():
    print("Key:", key, ", ", "Value:", value)
```

Key: name , Value: John Doe  
 Key: sex , Value: Male  
 Key: age , Value: 32  
 Key: married , Value: True

#### 4.6.2.5 Iterating using range

The `range` function is used to create a sequence of numbers that can be iterated over using a `for` loop. It can be used in 3 ways:

- `range(n)` - Creates a sequence of numbers from 0 to `n-1`
- `range(a, b)` - Creates a sequence of numbers from `a` to `b-1`
- `range(a, b, step)` - Creates a sequence of numbers from `a` to `b-1` with increments of `step`

Let's try it out.

```
for i in range(4):
    print(i)
```

0  
 1  
 2  
 3

```
for i in range(3, 8):
    print(i)
```

```
3
4
5
6
7

for i in range(3, 14, 4):
    print(i)
```

```
3
7
11
```

Ranges are used for iterating over lists when you need to track the index of elements while iterating.

```
a_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for i in range(len(a_list)):
    print('The value at position {} is {}'.format(i, a_list[i]))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

## 4.7 Object Oriented Programming

Python is an object-oriented programming language. In layman terms, it means that every number, string, data structure, function, class, module, etc., exists in the python interpreter as a python object. An object may have attributes and methods associated with it. For example, let us define a variable that stores an integer:

```
var = 2
```

The variable `var` is an object that has attributes and methods associated with it. For example a couple of its attributes are `real` and `imag`, which store the real and imaginary parts respectively, of the object `var`:

```
print("Real part of 'var': ",var.real)
print("Real part of 'var': ",var.imag)
```

```
Real part of 'var':  2
Real part of 'var':  0
```

**Attribute:** An attribute is a value associated with an object, defined within the class of the object.

**Method:** A method is a function associated with an object, defined within the class of the object, and has access to the attributes associated with the object.

For looking at attributes and methods associated with an object, say `obj`, press tab key after typing `obj..`

Consider the example below of a class `example_class`:

```
class example_class:
    class_name = 'My Class'
    def my_method(self):
        print('Hello World!')

e = example_class()
```

In the above class, `class_name` is an attribute, while `my_method` is a method.

#### 4.7.0.1 Call by Reference in Python

Python uses *call by object reference* (also called *call by sharing*). When you assign an object to a variable, the variable points to the object in memory—not a copy.

Changes made through one reference affect the original object. This is important to remember when working with mutable types like lists and dictionaries.

```
x = [5,3]
```

The variable name `x` is a reference to the memory location where the object `[5, 3]` is stored. Now, suppose we assign `x` to a new variable `y`:

```
y = x
```

In the above statement the variable name `y` now refers to the same object `[5,3]`. The object `[5,3]` does **not** get copied to a new memory location referred by `y`. To prove this, let us add an element to `y`:

```
y.append(4)  
print(y)
```

`[5, 3, 4]`

```
print(x)
```

`[5, 3, 4]`

When we changed `y`, note that `x` also changed to the same object, showing that `x` and `y` refer to the same object, instead of referring to different copies of the same object.

# 5 Data structures

```
<IPython.core.display.Image object>
```

In data science, you'll shape raw data **before modeling**. These built-ins are the foundation that explain why `Series`/`DataFrame` (pandas) and `ndarray` (NumPy) behave the way they do.

## 5.1 Primitives vs. Containers

- Primitive (single value): `int`, `float`, `bool`, `None`
- Containers (hold multiple values): `str`, `list`, `tuple`, `set`, `dict`

Tip: Check any object's type with `type(x)`.

## 5.2 Core Built-in Data Structures

### 5.2.1 Sequences (ordered, indexable)

Types: `list`, `tuple`, `str`

```
# list (mutable)
nums = [10, 20, 20, 30]
nums.append(40)          # mutate in place
nums[0] = 11             # item assignment
nums_slice = nums[1:3]   # slicing

# tuple (immutable)
pt = (42.0, -1.5)      # cannot reassign pt[0]

# string (immutable sequence of characters)
s = "data"
s2 = s.upper()          # returns a new string; s unchanged
```

### When to use:

- `list`: grow/shrink, frequent edits, ordered data
- `tuple`: fixed-size records, function returns, hashable as dict keys
- `str`: text processing (we'll also use `nltk` later)

### 5.2.2 Sets (unordered, unique)

Types: `set`

```
a = {1, 2, 2, 3}      # {1, 2, 3}
b = {3, 4}
a | b                # union -> {1, 2, 3, 4}
a & b                # intersection -> {3}
a - b                # difference -> {1, 2}
```

{1, 2}

### When to use:

- Deduplication, membership tests, fast set algebra.

### 5.2.3 Mappings (key–value)

Types: `dict`

```
student = {"name": "Alex", "year": 3, "major": "Stats"}
student["year"] = 4          # update
student["gpa"] = 3.7        # insert
for k, v in student.items(): # iterate keys & values
    print(k, "->", v)
```

name -> Alex  
year -> 4  
major -> Stats  
gpa -> 3.7

### When to use:

- Labeled data, lookups, configuration/state.

## 5.3 Iterables vs. Iterators

In Python, an **iterable** is any object capable of returning its members one at a time, such as lists, tuples, strings, and dictionaries. You can loop over iterables using a **for** loop.

An **iterator** is an object that represents a stream of data; it produces the next value when you call **next()** on it. Iterators are created from iterables using the **iter()** function.

**Key differences:** - Iterables can be looped over, but do not remember their position. - Iterators remember their position and can only be advanced one item at a time.

**Example:**

```
my_list = [1, 2, 3]
my_iter = iter(my_list)
print(next(my_iter)) # 1
print(next(my_iter)) # 2
print(next(my_iter)) # 3
```

Understanding the difference helps you write efficient loops and work with data streams in Python.

## 5.4 Common, Efficient Operations

### 5.4.1 Comprehensions

Comprehensions are a concise way to create lists, sets, or dictionaries from iterables by applying an expression to each item in an iterable (such as a list, tuple, or range) and optionally filtering the items based on a condition. They are a powerful and efficient way to generate new collections without the need for explicit loops.

**Basic Syntax:**

```
new_list = [expression for item in iterable if condition]
```

- **expression:** What you want to include in the new list (or set/dict).
- **item:** Represents each element in the iterable as the comprehension iterates through it.
- **iterable:** The source of elements (list, tuple, range, etc.).
- **condition (optional):** A filter to control which items are included. If omitted, all items are included.

**Why use comprehensions?** - More readable and concise than loops. - Often faster than equivalent for-loops. - Preferred for simple transformations and filtering.

### List comprehension example:

Create a list that has squares of natural numbers from 5 to 15.

```
sqrt_natural_no_5_15 = [(x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

[25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]

Create a list of tuples, where each tuple consists of a natural number and its square, for natural numbers ranging from 5 to 15.

```
sqrt_natural_no_5_15 = [(x,x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

[(5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11, 121), (12, 144), (13, 169), (14, 196), (15, 225)]

Creating a list of words that start with the letter 'a' in a given list of words.

```
words = ['apple', 'banana', 'avocado', 'grape', 'apricot']
a_words = [word for word in words if word.startswith('a')]
print(a_words)
```

['apple', 'avocado', 'apricot']

### Set comprehension:

```
unique_lengths = {len(word) for word in ['cat', 'dog', 'mouse']}
uniq_initials = {name[0].upper() for name in ["amy", "ann", "bob", "amy"]}
print(uniq_initials)
print(unique_lengths)
```

{'B', 'A'}
{3, 5}

### Dictionary comprehension:

```
word_lengths = {word: len(word) for word in ['cat', 'dog', 'mouse']}
print(word_lengths)
```

```
{'cat': 3, 'dog': 3, 'mouse': 5}
```

Comprehensions are preferred for simple transformations and filtering.

### 5.4.2 Membership & Lookups

Membership operations let you check if an item exists in a collection (like a list, set, or dictionary). Use `in` and `not in` for these checks. Lookups retrieve values by key or index, and are fastest in sets and dictionaries.

**Examples:** - `3 in [1, 2, 3] → True` (checks if 3 is in the list) - `'a' in {'a': 1, 'b': 2} → True` (checks if 'a' is a key in the dictionary) - `5 not in {1, 2, 3} → True` (checks if 5 is not in the set)

**Tip:** - Dictionary and set lookups are very fast (constant time). - List and tuple membership checks are slower (linear time). - For safe dictionary lookups, use `.get(key)` to avoid errors if the key is missing.

### 5.4.3 Unpacking

Unpacking lets you assign elements of a collection (like a list, tuple, or string) to multiple variables in a single step. This makes your code more readable and concise, especially when working with structured data.

**Basic Syntax:**

```
pt = (3, 4)
x, y = pt # x=3, y=4

a, b, c = [1, 2, 3] # a=1, b=2, c=3
```

You can also use unpacking in loops and with function arguments.

**Extended Unpacking:** Python allows you to use the `*` operator to capture multiple elements:

```
# Extended unpacking:  
first, *rest = [10, 20, 30, 40] # first=10, rest=[20, 30, 40]  
  
a, *mid, z = [1,2,3,4]  
  
print(rest)  
print(mid)
```

[20, 30, 40]

[2, 3]

### Unpacking in Loops:

```
pairs = [(1, 'a'), (2, 'b'), (3, 'c')]  
for num, char in pairs:  
    print(num, char)
```

1 a  
2 b  
3 c

### Unpacking with Dictionaries:

```
student = {"name": "Alex", "year": 3}  
for key, value in student.items():  
    print(key, value)
```

name Alex  
year 3

### Why use unpacking?

- Makes code cleaner and more readable
- Quick variable assignment
- Useful for working with structured data (e.g., tuples, lists, dicts)
- Avoids manual indexing

**Tip:** Unpacking works with any iterable, including lists, tuples, strings, and even the results of functions that return multiple values.

#### 5.4.4 Sorting in Python Iterables

Sorting is a common task when working with data. Python provides flexible ways to order lists, tuples, and other iterables.

##### 5.4.4.1 `sorted()` (built-in function)

- Returns a **new sorted list** from any iterable.
- Works with lists, tuples, strings, sets, and more.
- Does **not modify** the original iterable.

```
nums = [3, 1, 4, 1, 5]
print(sorted(nums))      # [1, 1, 3, 4, 5]
print(nums)              # [3, 1, 4, 1, 5] (unchanged)

word = "python"
print(sorted(word))     # ['h', 'n', 'o', 'p', 't', 'y']
```

```
[1, 1, 3, 4, 5]
[3, 1, 4, 1, 5]
['h', 'n', 'o', 'p', 't', 'y']
```

##### 5.4.4.2 `.sort()` (list method)

- **In-place sort**: modifies the list itself.
- Only available for lists (not other iterables).
- Returns `None`.

```
nums = [3, 1, 4, 1, 5]
nums.sort()
print(nums)    # [1, 1, 3, 4, 5]
```

```
[1, 1, 3, 4, 5]
```

#### 5.4.4.3 Reverse Sorting

Both `sorted()` and `.sort()` accept a `reverse` argument.

```
nums = [3, 1, 4, 1, 5]
print(sorted(nums, reverse=True))
```

```
[5, 4, 3, 1, 1]
```

#### 5.4.4.4 Sorting with a key

The `key` parameter lets you customize sorting logic.

```
# Sort by string length
words = ["pear", "apple", "banana", "kiwi"]
print(sorted(words, key=len))
# ['kiwi', 'pear', 'apple', 'banana']

# Sort by last character
print(sorted(words, key=lambda w: w[-1]))
# ['banana', 'pear', 'apple', 'kiwi']
```

```
['pear', 'kiwi', 'apple', 'banana']
['banana', 'apple', 'kiwi', 'pear']
```

#### 5.4.4.5 Sorting Complex Data

For lists of tuples or dicts, use `key`.

```
# Sort by the second element of each tuple
pairs = [("a", 3), ("b", 1), ("c", 2)]
print(sorted(pairs, key=lambda t: t[1]))
# [('b', 1), ('c', 2), ('a', 3)]

# Sort list of dicts by a field
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 92},
    {"name": "Chen", "grade": 78}
]
print(sorted(students, key=lambda s: s["grade"]))
```

```
[('b', 1), ('c', 2), ('a', 3)]  
[{'name': 'Chen', 'grade': 78}, {'name': 'Alice', 'grade': 85}, {'name': 'Bob', 'grade': 92}]
```

Sorting strings alphabetically vs. numerically:

```
nums = ["10", "2", "1"]  
print(sorted(nums))  
print(sorted(nums, key=int))
```

```
['1', '10', '2']  
['1', '2', '10']
```

#### 5.4.4.6 Mini Exercises

- Sort [10, 2, 33, 25, 7] in descending order.
- Given words = [“data”, “python”, “AI”, “science”], sort alphabetically ignoring case.
- Sort [(“Ann”, 22), (“Bob”, 19), (“Chen”, 22)] by age, preserving name order when ages match.

### 5.4.5 Lambda Functions in Python

Sometimes you only need a **tiny one-off function** for a specific task (like sorting by length or filtering items). Writing a full `def` feels heavy.  
That’s where **lambda functions** help.

#### Syntax

```
lambda parameters: expression
```

- Creates an anonymous function (no name required).
- Must contain a single expression (not multiple statements).
- Automatically returns the value of the expression.

```
square = lambda x: x**2  
print(square(5)) # 25
```

25

Using Lambda Functions

- With `sorted()`

```
words = ["pear", "apple", "banana", "kiwi"]

# Sort by word length
print(sorted(words, key=lambda w: len(w)))

# Sort by last character
print(sorted(words, key=lambda w: w[-1]))
```

`['pear', 'kiwi', 'apple', 'banana']  
['banana', 'apple', 'kiwi', 'pear']`

- With `map()` and `filter()`

```
nums = [1, 2, 3, 4, 5]

# Square each number
print(list(map(lambda x: x**2, nums)))

# Keep only even numbers
print(list(filter(lambda x: x % 2 == 0, nums)))
```

`[1, 4, 9, 16, 25]  
[2, 4]`

- With `reduce()` (from `functools`)

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, nums)
print(product) # 120
```

120

### Key Takeaways:

- `lambda` = quick, throwaway function for one-liners
- use `def` if the function
  - has multiple steps
  - is reused often

#### 5.4.5.1 Mini Exercises

- Use `lambda` with `sorted()` to order the list:

```
nums = [-3, 1, -2, 5, 0]
```

- Use `lambda` with `filter()` to keep only names starting with a vowel:

```
names = ["Alice", "Bob", "Eve", "Uma", "Sam"]
```

- Use `lambda` with `map()` to convert Celsius to Fahrenheit:

```
temps_c = [0, 20, 37, 100]
```

#### 5.4.6 enumerate()

The `enumerate()` function is a built-in Python tool that adds a counter to any iterable, returning pairs of (index, item) as you loop. This is especially useful when you need both the item and its position in a loop.

##### Syntax:

```
for index, value in enumerate(iterable, start=0):
    # use index and value
```

- `iterable`: Any sequence (list, tuple, string, etc.)
- `start`: Optional, sets the starting index (default is 0)

##### Why use `enumerate()`?

- Makes code cleaner and less error-prone
- Avoids manual index tracking with a separate variable
- Works with lists, tuples, strings, and more

##### Example:

```
fruits = ['apple', 'banana', 'cherry']
for idx, fruit in enumerate(fruits):
    print(idx, fruit)
```

```
0 apple
1 banana
2 cherry
```

**Tip:** You can set the starting index with the `start` argument, e.g. `enumerate(my_list, start=1)`.

Use `enumerate()` for readable, efficient loops when you need both index and value.

#### 5.4.7 `zip()`

The `zip()` function combines two or more iterables (like lists, tuples, or strings) into tuples, pairing elements by their position. This is useful for parallel iteration, creating pairs, or merging data from multiple sources.

**Syntax:**

```
zip(iterable1, iterable2, ...)
```

- Each tuple contains one element from each iterable, matched by position.
- Stops at the shortest iterable.

**Why use `zip()`?** - Parallel iteration over multiple sequences - Pairing related data (e.g., names and scores) - Creating dictionaries from two lists

**Example:**

```
names = ['Alice', 'Bob', 'Chen']
scores = [85, 92, 78]
for name, score in zip(names, scores):
    print(name, score)
```

```
Alice 85
Bob 92
Chen 78
```

```
# Creating a dictionary from two lists:
gradebook = dict(zip(names, scores))
print(gradebook)
```

```
{'Alice': 85, 'Bob': 92, 'Chen': 78}
```

**Tip:** - You can use `zip(*zipped)` to unzip a list of tuples back into separate lists. - If the input iterables are different lengths, `zip()` stops at the shortest one.

#### 5.4.8 Common Functions for Iterables

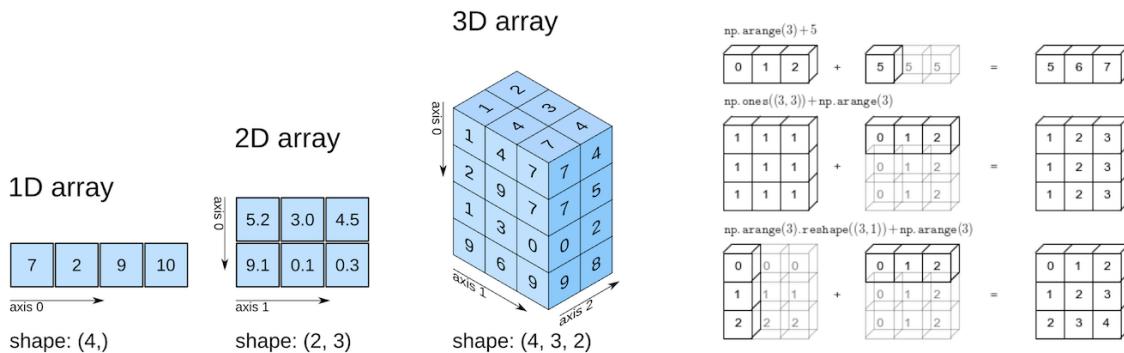
Python provides a variety of built-in functions to operate on iterables, making it easy to manipulate, process, and analyze collections like lists, tuples, strings, sets, and dictionaries. Below is a list of commonly used built-in functions specifically designed for iterables.

Function	Description	Example
<code>len()</code>	Returns the number of elements in an iterable.	<code>len([1, 2, 3]) → 3</code>
<code>min()</code>	Returns the smallest element in an iterable.	<code>min([3, 1, 4]) → 1</code>
<code>max()</code>	Returns the largest element in an iterable.	<code>max([3, 1, 4]) → 4</code>
<code>sum()</code>	Returns the sum of elements in an iterable (numeric types only).	<code>sum([1, 2, 3]) → 6</code>
<code>sorted()</code>	Returns a sorted list from an iterable (does <b>not</b> modify the original).	<code>sorted([3, 1, 2]) → [1, 2, 3]</code>
<code>reversed()</code>	Returns an iterator that accesses the elements of an iterable in reverse.	<code>list(reversed([1, 2, 3])) → [3, 2, 1]</code>
<code>enumerate()</code>	Returns an iterator of tuples containing indices and elements of the iterable.	<code>list(enumerate(['a', 'b', 'c'])) → [(0, 'a'), (1, 'b'), (2, 'c')]</code>
<code>all()</code>	Returns <code>True</code> if all elements of the iterable are true (or if empty).	<code>all([True, 1, 'a']) → True</code>
<code>any()</code>	Returns <code>True</code> if <b>any</b> element of the iterable is true.	<code>any([False, 0, 'b']) → True</code>
<code>str.join()</code>	Joins elements of an iterable (e.g., list, tuple) into a single string, using the given string as a separator.	<code>' '.join(['a', 'b', 'c']) → 'abc'</code>

## **Part III**

# **Core library foundations**

# 6 NumPy



**NumPy** is a foundational library in Python, providing support for large, multi-dimensional arrays and matrices, along with a variety of mathematical functions. It's a critical tool in data science and machine learning because it enables efficient numerical computations, data manipulation, and linear algebra operations. Many machine learning algorithms rely on these operations to process data and perform complex calculations quickly. Moreover, popular libraries like Pandas, SciPy, and TensorFlow are built on top of NumPy, making it essential to understand for implementing and optimizing machine learning models.

## 6.1 Learning Objectives

By the end of this lecture, students should be able to:

1. Understand the basic structure and functionality of NumPy
2. Create and manipulate NumPy arrays.
3. Perform mathematical and statistical operations on arrays.
4. Utilize advanced concepts such as slicing, broadcasting, and vectorization.
5. Apply NumPy operations to real-world data science problems.

## 6.2 Getting Started

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

If you encounter a ‘ModuleNotFoundError’, please ensure that the module is installed in your current environment before attempting to import it

```
#Using the NumPy function array() to define a NumPy array
numpy_array = np.array([[1,2],[3,4]])
numpy_array
```

```
array([[1, 2],
       [3, 4]])
```

```
type(numpy_array)
```

```
numpy.ndarray
```

Numpy arrays can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the `.shape` property of an array.

```
numpy_array.ndim
```

```
2
```

## 6.3 Data Types in NumPy

Unlike lists and tuples, NumPy arrays are designed to store elements of the same type, enabling more efficient memory usage and faster computations. The data type of the elements in a NumPy array can be accessed using the `.dtype` attribute

```
numpy_array.dtype
```

```
dtype('int64')
```

NumPy supports a wide range of data types, each with a specific memory size. Here is a list of common NumPy data types and the memory they consume:

Note that: these data type correspond directly to C data types, since NumPy uses C for the core computational operations. This C implementation allows NumPy to perform array operations much faster and more efficiently than native Python data structures.

Data Type	Memory Size
<code>np.int8</code>	1 byte
<code>np.int16</code>	2 bytes
<code>np.int32</code>	4 bytes
<code>np.int64</code>	8 bytes
<code>np.uint8</code>	1 byte
<code>np.uint16</code>	2 bytes
<code>np.uint32</code>	4 bytes
<code>np.uint64</code>	8 bytes
<code>np.float16</code>	2 bytes
<code>np.float32</code>	4 bytes
<code>np.float64</code>	8 bytes
<code>np.complex64</code>	8 bytes
<code>np.complex128</code>	16 bytes
<code>np.bool_</code>	1 byte
<code>np.string_</code>	1 byte per character
<code>np.unicode_</code>	4 bytes per character
<code>np.object_</code>	Variable
<code>np.datetime64</code>	8 bytes
<code>np.timedelta64</code>	8 bytes

### 6.3.1 Upcasting

When creating a NumPy array with elements of different data types, NumPy automatically attempts to **upcast** the elements to a compatible data type that can accommodate all of them. This process is known as **type coercion** or **type promotion**. The rules for upcasting follow a hierarchy of data types to ensure no data is lost.

Below are two common types of upcasting with examples:

**Numeric Upcasting:** If you mix integers and floats, NumPy will convert the entire array to floats.

```
arr = np.array([1, 2.5, 3])
print(arr.dtype)
```

```
float64
```

**String Upcasting:** If you mix numbers and strings, NumPy will upcast all elements to strings.

```
arr = np.array([1, 'hello', 3.5])
print(arr.dtype)
```

```
<U32
```

<U21 is a NumPy data type that stands for a Unicode string with a maximum length of 21 characters.

## 6.4 Why do we need NumPy arrays?

NumPy arrays can store data similarly to lists and tuples, and the same computations can be performed across these structures. However, NumPy is preferred because it is significantly more efficient, particularly when handling large datasets, due to its optimized memory usage and computational speed.

### 6.4.1 Numpy arrays are memory efficient: Homogeneity and Contiguous Memory Storage

A NumPy array is a collection of elements of the same data type, stored in contiguous memory locations. In contrast, data structures like lists can hold elements of different data types, stored in non-contiguous memory locations. This homogeneity and contiguous storage allow NumPy arrays to be densely packed, leading to lower memory consumption. The following example demonstrates how NumPy arrays are more memory-efficient compared to other data structures.

```
import sys

# Create a NumPy array, Python list, and tuple with the same elements
array = np.arange(1000)
py_list = list(range(1000))
py_tuple = tuple(range(1000))

# Calculate memory usage
array_memory = array.nbytes
```

```

list_memory = sys.getsizeof(py_list) + sum(sys.getsizeof(item) for item in py_list)
tuple_memory = sys.getsizeof(py_tuple) + sum(sys.getsizeof(item) for item in py_tuple)

# Display the memory usage
memory_usage = {
    "NumPy Array (in bytes)": array_memory,
    "Python List (in bytes)": list_memory,
    "Python Tuple (in bytes)": tuple_memory
}

memory_usage

```

```

{'NumPy Array (in bytes)': 8000,
 'Python List (in bytes)': 36056,
 'Python Tuple (in bytes)': 36040}

# each element in the array is a 64-bit integer
array.dtype

```

```

dtype('int64')

```

#### 6.4.2 NumPy arrays are fast

With NumPy arrays, mathematical computations can be performed faster, as compared to other data structures, due to the following reasons:

1. As the NumPy array is **densely packed** with homogenous data, it helps retrieve the data faster as well, thereby making computations faster.
2. With NumPy, **vectorized computations** can replace the relatively more expensive python **for** loops. The NumPy package breaks down the vectorized computations into multiple fragments and then processes all the fragments parallelly. However, with a **for** loop, computations will be one at a time.
3. The NumPy package **integrates C**, and **C++** codes in Python. These programming languages have very little execution time as compared to Python.

We'll see the faster speed on NumPy computations in the example below.

**Example:** This example shows that computations using NumPy arrays are typically much faster than computations with other data structures.

**Q:** Multiply whole numbers up to 1 million by an integer, say 2. Compare the time taken for the computation if the numbers are stored in a NumPy array vs a list.

Use the numpy function `arange()` to define a one-dimensional NumPy array.

```
#Examples showing NumPy arrays are more efficient for numerical computation
import time as tm

# List comprehension to multiply each element by 2
start_time = tm.time()
list_ex = list(range(1000000)) # List containing whole numbers up to 1 million
a = [x * 2 for x in list_ex] # Multiply each element by 2
print("Time taken to multiply numbers in a list = ", tm.time() - start_time)

# Tuple - converting to list for multiplication, then back to tuple
start_time = tm.time()
tuple_ex = tuple(range(1000000)) # Tuple containing whole numbers up to 1 million
a = tuple(x * 2 for x in tuple_ex) # Multiply each element by 2
print("Time taken to multiply numbers in a tuple = ", tm.time() - start_time)

# NumPy array element-wise multiplication
start_time = tm.time()
numpy_ex = np.arange(1000000) # NumPy array containing whole numbers up to 1 million
a = numpy_ex * 2 # Multiply each element by 2
print("Time taken to multiply numbers in a NumPy array = ", tm.time() - start_time)

Time taken to multiply numbers in a list =  0.049832820892333984
Time taken to multiply numbers in a tuple =  0.08172726631164551
Time taken to multiply numbers in a NumPy array =  0.010966300964355469
```

## 6.5 Basics of NumPy Arrays

### 6.5.1 Creating NumPy Arrays

#### 6.5.1.1 Creating Arrays for specific use cases:

You can create a NumPy array using various methods, such as:

- `np.array()`: Creates an array from a list or iterable.
- `np.zeros()`, `np.ones()`: Creates arrays filled with zeros or ones.
- `np.arange()`, `np.linspace()`: Creates arrays with evenly spaced values.

- `np.random` module: Generates arrays with random values.

Each method is designed for different use cases. I encourage you to explore and experiment with these functions to see the types of arrays they produce and how they can be used in different scenarios.

### 6.5.1.2 Loading data from file into a NumPy array

- `numpy.loadtxt()`: Reading simple numerical text files
- `np.genfromtxt()`: Reading more complex text files
- `df.to_numpy()`: Reading tabular data into pandas dataframe, then using ‘`to_numpy()`’ convert it to NumPy array

## 6.5.2 Array Attributes:

Let us define a NumPy array in order to access its attributes:

```
numpy_ex = np.array([[1,2,3],[4,5,6]])
numpy_ex
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

The attributes of `numpy_ex` can be seen by typing `numpy_ex` followed by a `.`, and then pressing the *tab* key.

Some of the basic attributes of a NumPy array are the following:

### 6.5.2.1 `ndim`

Shows the number of dimensions (or axes) of the array.

```
numpy_ex.ndim
```

2

### 6.5.2.2 `shape`

This is a tuple of integers indicating the size of the array in each dimension. For a matrix with  $n$  rows and  $m$  columns, the shape will be  $(n,m)$ . The length of the shape tuple is therefore the rank, or the number of dimensions, `ndim`.

```
numpy_ex.shape
```

(2, 3)

### 6.5.2.3 `size`

This is the total number of elements of the array, which is the product of the elements of `shape`.

```
numpy_ex.size
```

6

### 6.5.2.4 `dtype`

This is an object describing the type of the elements in the array. One can create or specify `dtype`'s using standard Python types. NumPy provides many, for example `bool_`, `character`, `int_`, `int8`, `int16`, `int32`, `int64`, `float_`, `float8`, `float16`, `float32`, `float64`, `complex_`, `complex64`, `object_`.

```
numpy_ex.dtype
```

`dtype('int64')`

## 6.6 Array Indexing and Slicing

### 6.6.1 Array Indexing

Similar to Python lists, NumPy uses zero-based indexing, meaning the first element of an array is accessed using index 0. You can use positive or negative indices to access elements

```
array = np.array([10, 20, 30, 40, 50])

print(array[0])
print(array[4])
print(array[-1])
print(array[-3])
```

```
10
50
50
30
```

In multi-dimensional arrays, indices are separated by commas. The first index refers to the row, and the second index refers to the column in a 2D array.

```
# 2D array (3 rows, 3 columns)
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(array_2d)
print(array_2d[0, 1])
print(array_2d[1, -1])
print(array_2d[-1, -1])
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
2
6
9
```

You can use boolean arrays to filter or select elements based on a condition

```
array = np.array([10, 20, 30, 40, 50])
mask = array > 30 # Boolean mask for elements greater than 30
print(array[mask]) # Output: [40 50]
```

```
[40 50]
```

## 6.6.2 Array Slicing

Slicing is used to extract a sub-array from an existing array.

The Syntax for slicing is ‘array[start:stop:step]

```
array = np.array([10, 20, 30, 40, 50])

print(array[1:4])
print(array[:3])
print(array[2:])
print(array[::-2])
print(array[::-1])
```

```
[20 30 40]
[10 20 30]
[30 40 50]
[10 30 50]
[50 40 30 20 10]
```

For slicing in Multi-Dimensional Arrays, use commas to separate slicing for different dimensions

```
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Extract a sub-array: elements from the first two rows and the first two columns
sub_array = array_2d[:2, :2]
print(sub_array)

# Extract all rows for the second column
col = array_2d[:, 1]
print(col)

# Extract the last two rows and last two columns
sub_array = array_2d[-2:, -2:]
print(sub_array)
```

```
[[1 2]
 [4 5]]
[2 5 8]
[[5 6]
 [8 9]]
```

The `step` parameter can be used to select elements at regular intervals.

```
array = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  
  
print(array[1:8:2])  
print(array[::-2])
```

```
[1 3 5 7]  
[9 7 5 3 1]
```

### 6.6.3 Modify Sub-Arrays through Slicing

Slices are views of the original array, not copies. Modifying a slice will change the original array.

```
array = np.array([10, 20, 30, 40, 50])  
array[1:4] = 100 # Replace elements from index 1 to 3 with 100  
print(array) # Output: [ 10 100 100 100 50]
```

```
[ 10 100 100 100 50]
```

### 6.6.4 Combining Indexing and Slicing

You can combine indexing and slicing to extract specific elements or sub-arrays

```
# Create a 3D array  
array_3d = np.array([[1, 2, 3], [4, 5, 6], [[7, 8, 9], [10, 11, 12]]])  
  
# Select specific elements and slices  
print(array_3d[0, :, 1]) # Output: [2 5] (second element from each row in the first sub-array)  
print(array_3d[1, 1, :2]) # Output: [10 11] (first two elements in the last row of the second sub-array)
```

```
[2 5]  
[10 11]
```

### 6.6.5 `np.where()` and `np.select()`

You can also use `np.where` and `np.select` for array slicing and conditional selection.

```
array_3d
```

```
array([[ [ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

# Using np.where to create a mask and select elements greater than 5
greater_than_5 = np.where(array_3d > 5, array_3d, 0)
print("Elements greater than 5:\n", greater_than_5)

# Using np.select to categorize elements into three categories
conditions = [array_3d < 4, (array_3d >= 4) & (array_3d <= 8), array_3d > 8]
choices = ['low', 'medium', 'high']
categorized_array = np.select(conditions, choices, default='unknown')
print("\nCategorized array:\n", categorized_array)
```

Elements greater than 5:

```
[[[ 0  0  0]
  [ 0  0  6]]]
```

```
[[ 7  8  9]
 [10 11 12]]]
```

Categorized array:

```
[[['low' 'low' 'low']
  ['medium' 'medium' 'medium']]

[['medium' 'medium' 'high']
 ['high' 'high' 'high']]]
```

This example shows how `np.where` and `np.select` can be used to filter, manipulate, and categorize elements within a 3D array based on specific conditions.

## 6.6.6 `np.argmin()` and `np.argmax()`

You can use `np.argmin` and `np.argmax` to quickly find the index of the minimum or maximum value in an array along a specified axis. You'll see their usage in the practice example below

## 6.7 Array Operations

### 6.7.1 Arithmetic operations

Numpy arrays support arithmetic operators like `+`, `-`, `*`, etc. We can perform an arithmetic operation on an array either with a single number (also called scalar) or with another array of the same shape. However, we cannot perform an arithmetic operation on an array with an array of a different shape.

Below are some examples of arithmetic operations on arrays.

```
#Defining two arrays of the same shape
arr1 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
arr2 = np.array([[11, 12, 13, 14],
                 [15, 16, 17, 18],
                 [19, 11, 12, 13]])
```

```
#Element-wise summation of arrays
arr1 + arr2
```

```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 12, 14, 16]])
```

```
# Element-wise subtraction
arr2 - arr1
```

```
array([[10, 10, 10, 10],
       [10, 10, 10, 10],
       [10, 10, 10, 10]])
```

```
# Adding a scalar to an array adds the scalar to each element of the array
arr1 + 3
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12,  4,  5,  6]])
```

```

# Dividing an array by a scalar divides all elements of the array by the scalar
arr1 / 2

array([[0.5, 1. , 1.5, 2. ],
       [2.5, 3. , 3.5, 4. ],
       [4.5, 0.5, 1. , 1.5]])

# Element-wise multiplication
arr1 * arr2

array([[ 11,  24,  39,  56],
       [ 75,  96, 119, 144],
       [171,  11,  24,  39]])

# Modulus operator with scalar
arr1 % 4

array([[1, 2, 3, 0],
       [1, 2, 3, 0],
       [1, 1, 2, 3]])

```

### 6.7.2 Comparison and Logical Operation

Numpy arrays support comparison operations like `==`, `!=`, `>` etc. The result is an array of booleans.

```

arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])

arr1 == arr2

array([[False,  True,  True],
       [False, False,  True]])

arr1 != arr2

array([[ True, False, False],
       [ True,  True, False]])

```

```
arr1 >= arr2
```

```
array([[False,  True,  True],  
       [ True,  True,  True]])
```

```
arr1 < arr2
```

```
array([[ True, False, False],  
       [False, False, False]])
```

Array comparison is frequently used to count the number of equal elements in two arrays using the `sum` method. Remember that `True` evaluates to 1 and `False` evaluates to 0 when booleans are used in arithmetic operations.

```
(arr1 == arr2).sum()
```

```
np.int64(3)
```

### 6.7.3 Aggregate Functions: `np.sum()`, `np.mean()`, `np.min()`, `np.max()`

#### 6.7.3.1 Overall Aggregate Calculations:

- `np.sum(array)`: Calculates the sum of all elements in the array.
- `np.mean(array)`: Calculates the mean of all elements in the array.
- `np.min(array)`: Finds the minimum value in the entire array.
- `np.max(array)`: Finds the maximum value in the entire array.

#### 6.7.3.2 Row-Wise Calculations (`axis=1`):

- `np.sum(array, axis=1)`: Computes the sum of elements in each row.
- `np.mean(array, axis=1)`: Computes the mean of elements in each row.
- `np.min(array, axis=1)`: Finds the minimum value in each row.
- `np.max(array, axis=1)`: Finds the maximum value in each row.

### 6.7.3.3 Column-Wise Calculations (axis=0):

- `np.sum(array, axis=0)`: Computes the sum of elements in each column.
- `np.mean(array, axis=0)`: Computes the mean of elements in each column.
- `np.min(array, axis=0)`: Finds the minimum value in each column.
- `np.max(array, axis=0)`: Finds the maximum value in each column.

```
# Create a 3x4 array of integers
array = np.array([[4, 7, 1, 3],
                  [5, 8, 2, 6],
                  [9, 3, 5, 2]])

# Display the original array
print("Original Array:\n", array)

# Calculate the sum, mean, minimum, and maximum for the entire array
total_sum = np.sum(array)
mean_value = np.mean(array)
min_value = np.min(array)
max_value = np.max(array)

print(f"\nSum of all elements: {total_sum}")
print(f"Mean of all elements: {mean_value}")
print(f"Minimum value in the array: {min_value}")
print(f"Maximum value in the array: {max_value}")

# Calculate the sum, mean, minimum, and maximum along each row (axis=1)
row_sum = np.sum(array, axis=1)
row_mean = np.mean(array, axis=1)
row_min = np.min(array, axis=1)
row_max = np.max(array, axis=1)

print("\nSum along each row:", row_sum)
print("Mean along each row:", row_mean)
print("Minimum value along each row:", row_min)
print("Maximum value along each row:", row_max)

# Calculate the sum, mean, minimum, and maximum along each column (axis=0)
col_sum = np.sum(array, axis=0)
col_mean = np.mean(array, axis=0)
col_min = np.min(array, axis=0)
col_max = np.max(array, axis=0)
```

```
print("\nSum along each column:", col_sum)
print("Mean along each column:", col_mean)
print("Minimum value along each column:", col_min)
print("Maximum value along each column:", col_max)
```

Original Array:

```
[[4 7 1 3]
 [5 8 2 6]
 [9 3 5 2]]
```

Sum of all elements: 55

Mean of all elements: 4.583333333333333

Minimum value in the array: 1

Maximum value in the array: 9

Sum along each row: [15 21 19]

Mean along each row: [3.75 5.25 4.75]

Minimum value along each row: [1 2 2]

Maximum value along each row: [7 8 9]

Sum along each column: [18 18 8 11]

Mean along each column: [6. 6. 2.66666667 3.66666667]

Minimum value along each column: [4 3 1 2]

Maximum value along each column: [9 8 5 6]

## 6.8 Array Reshaping

Certain functions and machine learning models require input data in a specific shape or format. For instance, operations like matrix multiplication and broadcasting depend on the alignment of array dimensions. Many deep learning models expect input data to be in a 4D array format (batch size, height, width, channels). Reshaping enables us to convert data into the necessary shape, ensuring compatibility without altering the underlying values.

Below are some methods to reshape NumPy arrays:

### 6.8.1 `reshape()`

The `reshape` method in NumPy allows you to change the shape of an existing array without changing its data.

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
```

```
[[1 2 3]
 [4 5 6]]
```

Using -1 for automatic dimension inference

```
arr = np.arange(12)
reshaped_arr = arr.reshape(3, -1)
print(reshaped_arr)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Here, -1 means “calculate this dimension based on the remaining dimensions and the total size of the array”. So, (3, -1) becomes (3, 4).

### 6.8.2 flatten() and ravel()

The `flatten` method returns a copy of the array collapsed into one dimension. It is useful when you need to perform operations that require 1D input or need to pass the array data as a linear sequence. `order` specifies the order in which elements are read, options include but not limited to: \* C (default): Row-major (C-style). \* F: Column-major (Fortran-style).

In contrast, `ravel()` returns a flattened view of the original array whenever possible, without creating a copy.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
flattened_arr = arr.flatten()
print(flattened_arr)
```

```
[1 2 3 4 5 6]
```

```
# using ravel() to flatten the array
flattened_arr = arr.ravel()
print(flattened_arr)
```

```
[1 2 3 4 5 6]
```

### 6.8.3 `resize()`

Changes the shape and size of an array in place. Unlike `reshape()`, it can modify the array and fill additional elements with zeros if necessary.

```
arr = np.array([1, 2, 3, 4])
arr.resize(2, 3)
print(arr)
```

```
[[1 2 3]
 [4 0 0]]
```

### 6.8.4 `transpose()` or `T`

Both can be used to transpose the NumPy array. This is often used to make matrices (2-dimensional arrays) compatible for multiplication.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
transposed_arr = arr.transpose() # Output: [[1 4], [2 5], [3 6]]
print(transposed_arr)

T_arr = arr.T # Output: [[1 4], [2 5], [3 6]]
print(T_arr)
```

```
[[1 4]
 [2 5]
 [3 6]]
[[1 4]
 [2 5]
 [3 6]]
```

## 6.9 Arrays Concatenating

NumPy provides several functions to concatenate arrays along different axes.

### 6.9.1 np.concatenate()

Arrays can be concatenated along an axis with NumPy's `concatenate` function. The `axis` argument specifies the dimension for concatenation. The arrays should have the same number of dimensions, and the same length along each axis except the axis used for concatenation.

The examples below show concatenation of arrays.

```
arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])
print("Array 1:\n",arr1)
print("Array 2:\n",arr2)
```

```
Array 1:
[[1 2 3]
 [3 4 5]]
Array 2:
[[2 2 3]
 [1 2 5]]
```

```
#Concatenating the arrays along the default axis: axis=0
np.concatenate((arr1,arr2))
```

```
array([[1, 2, 3],
       [3, 4, 5],
       [2, 2, 3],
       [1, 2, 5]])
```

```
#Concatenating the arrays along axis = 1
np.concatenate((arr1,arr2),axis=1)
```

```
array([[1, 2, 3, 2, 2, 3],
       [3, 4, 5, 1, 2, 5]])
```

Here's a visual explanation of `np.concatenate` along `axis=1` (can you guess what `axis=0` results in?):

Let us concatenate the array below (`arr3`) with `arr1`, along axis = 0.

```
arr3 = np.array([2, 2, 3])  
  
np.concatenate((arr1,arr3),axis=0)
```

```
ValueError: all the input arrays must have same number of dimensions, but the array at index  
-----  
ValueError Traceback (most recent call last)  
Cell In[52], line 1  
----> 1 np.concatenate((arr1,arr3),axis=0)  
ValueError: all the input arrays must have same number of dimensions, but the array at index
```

Note the above error, which indicates that `arr3` has only one dimension. Let us check the shape of `arr3`.

```
arr3.shape
```

```
(3,)
```

We can reshape `arr3` to a shape of (1,3) to make it compatible for concatenation with `arr1` along axis = 0.

```
arr3_reshaped = arr3.reshape(1,3)  
arr3_reshaped  
  
array([[2, 2, 3]])
```

Now we can concatenate the reshaped `arr3` with `arr1` along axis = 0.

```
np.concatenate((arr1,arr3_reshaped),axis=0)  
  
array([[1, 2, 3],  
       [3, 4, 5],  
       [2, 2, 3]])
```

### 6.9.2 `np.vstack()` and `np.hstack()`

- `np.vstack()`: Stacks arrays vertically (along rows).
- `np.hstack()`: Stacks arrays horizontally (along columns).

```

# Vertical stacking
vstack = np.vstack((arr1, arr2))
print("\nVertical Stack:\n", vstack)

# Horizontal stacking
hstack = np.hstack((arr1, arr2))
print("\nHorizontal Stack:\n",.hstack)

```

Vertical Stack:

```

[[1 2 3]
 [3 4 5]
 [2 2 3]
 [1 2 5]]

```

Horizontal Stack:

```

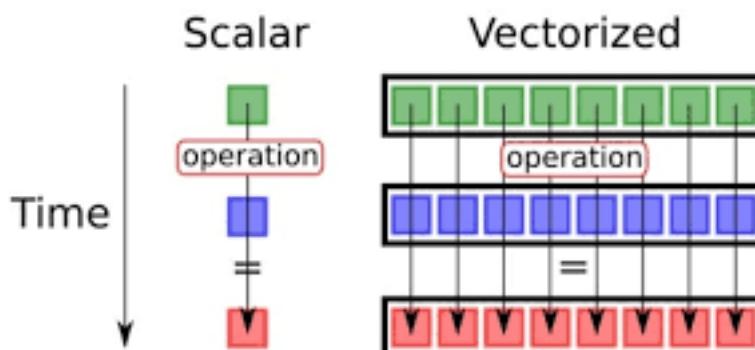
[[1 2 3 2 2 3]
 [3 4 5 1 2 5]]

```

## 6.10 Vectorization in NumPy

### 6.10.1 Introduction to Vectorization

Vectorization is the process of applying operations to entire arrays or matrices simultaneously rather than iterating over individual elements using loops. This approach allows NumPy to utilize highly optimized C libraries for faster computation.



Comparison between scalar (classical) computation and vectorization.

Benefits of Vectorization:

- Performance: Significantly faster than using for-loops in Python.
- Conciseness: Shorter and more readable code.
- Efficiency: Reduces the overhead of Python loops and leverages lower-level optimizations.

### 6.10.2 Understanding Vectorized Operations with Examples

```
# Create two arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([10, 20, 30, 40, 50])

# Element-wise addition (vectorized operation)
sum_arr = arr1 + arr2
print("Sum Array:", sum_arr)
```

Sum Array: [11 22 33 44 55]

### 6.10.3 NumPy Vectorization Vs Python for Loop

`np.dot` is a vectorized operation in NumPy that performs matrix multiplication or dot product between arrays. It efficiently computes the element-wise multiplications and then sums them up. To better understand its efficiency, let's first implement the dot product using for loops, and then compare its performance with `np.dot` to see the benefits of vectorization.

```
import time

# Function to calculate dot product using for loops
def dot_product_loops(arr1, arr2):
    result = 0
    for i in range(len(arr1)):
        result += arr1[i] * arr2[i]
    return result

# Create sample arrays
arr1 = np.random.rand(1000000)
arr2 = np.random.rand(1000000)

# Measure time for the loop-based implementation
start_time = time.time()
loop_result = dot_product_loops(arr1, arr2)
loop_time = time.time() - start_time
```

```

# Measure time for np.dot
start_time = time.time()
numpy_result = np.dot(arr1, arr2)
numpy_time = time.time() - start_time

# Display results
print(f"Loop-based implementation result: {loop_result:.5f}, Time: {loop_time:.5f} seconds")
print(f"NumPy np.dot result: {numpy_result:.5f}, Time: {numpy_time:.5f} seconds")

```

```

Loop-based implementation result: 250068.46992, Time: 0.19136 seconds
NumPy np.dot result: 250068.46992, Time: 0.00199 seconds

```

The `np.dot` function significantly outperforms the manual loop-based implementation because it leverages NumPy's vectorized operations, which is written in highly efficient C code. This example highlights why vectorized operations like `np.dot` are preferred for large-scale numerical computations in NumPy.

#### 6.10.4 Broadcasting and Its Role in Vectorization

Broadcasting allows NumPy to perform operations between arrays of different shapes by automatically expanding their dimensions. This is essential for vectorized operations involving arrays with varying shapes.

Let's look at an example to see how it works

```

arr2 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])

arr4 = np.array([4, 5, 6, 7])

arr2 + arr4

```

```

array([[ 5,  7,  9, 11],
       [ 9, 11, 13, 15],
       [13,  6,  8, 10]])

```

When the expression `arr2 + arr4` is evaluated, `arr4` (which has the shape `(4,)`) is replicated three times to match the shape `(3, 4)` of `arr2`. Numpy performs the replication without actually creating three copies of the smaller dimension array, thus improving performance and using lower memory.

Broadcasting only works if one of the arrays can be replicated to match the other array's shape.

- Dimensions of size 1 will broadcast (as if the value was repeated).
- Otherwise, the dimension must have the same shape.

```
arr5 = np.array([7, 8])
```

```
arr2 + arr5
```

```
ValueError: operands could not be broadcast together with shapes (3,4) (2,)
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[62], line 1  
----> 1 arr2 + arr5  
ValueError: operands could not be broadcast together with shapes (3,4) (2,)
```

In the above example, even if `arr5` is replicated three times, it will not match the shape of `arr2`. Hence `arr2 + arr5` cannot be evaluated successfully. See the [broadcasting documentation](#) to learn more about it.

### 6.10.5 Matrix Multiplication in NumPy with Vectorization

Matrix multiplication is one of the most common and computationally intensive operations in numerical computing and deep learning. NumPy offers efficient and highly optimized methods for performing matrix multiplication, which leverage vectorization to handle large matrices quickly and accurately.

**Note that:** NumPy matrix operations follow the standard rules of linear algebra, so it's important to ensure that the shapes of the matrices are compatible. If they are not, consider reshaping the matrices before performing multiplication

There are two commonly methods for matrix multiplication

#### 6.10.5.1 Method 1: Matrix Multiplication Using `np.dot()`

```

# Define two 2D arrays (matrices)
matrix1 = np.array([[1, 2, 3],
                   [4, 5, 6]])
matrix2 = np.array([[7, 8],
                   [9, 10],
                   [11, 12]])

# Matrix multiplication using np.dot
result_dot = np.dot(matrix1, matrix2)
print("Matrix Multiplication using np.dot:\n", result_dot)

# Another way to perform np.dot for matrix multiplication
result_dot2 = matrix1.dot(matrix2)
print("\nMatrix Multiplication using dot method:\n", result_dot2)

```

Matrix Multiplication using np.dot:

```

[[ 58  64]
 [139 154]]

```

Matrix Multiplication using dot method:

```

[[ 58  64]
 [139 154]]

```

#### 6.10.5.2 Method 2: Matrix Multiplication using np.matmul() or @

```

# Matrix multiplication using np.matmul or @ operator
result_matmul = np.matmul(matrix1, matrix2)
result_operator = matrix1 @ matrix2
print("\nMatrix Multiplication using np.matmul:\n", result_matmul)
print("\nMatrix Multiplication using @ operator:\n", result_operator)

```

Matrix Multiplication using np.matmul:

```

[[ 58  64]
 [139 154]]

```

Matrix Multiplication using @ operator:

```

[[ 58  64]
 [139 154]]

```

Note that \* operator in numpy is element-wise multiplication

```
# using * operator for element-wise multiplication, please uncomment the code below to run
# element_wise = matrix1 * matrix2
# print("\nElement-wise Multiplication:\n", element_wise)

# reshape the array for element-wise multiplication
matrix2_reshaped = matrix2.reshape(2, 3)
element_wise = matrix1 * matrix2_reshaped
print("\nElement-wise Multiplication after reshaping:\n", element_wise)
```

Element-wise Multiplication after reshaping:

```
[[ 7 16 27]
 [40 55 72]]
```

## 6.11 Converting Between NumPy Arrays and pandas DataFrames

### 6.11.1 Why Conversion Matters

While both pandas and NumPy are integral to data science, they serve different purposes:

1. pandas:

- Focuses on **data manipulation** and analysis
- Provided intuitive tools to work with **tabular data**(e.g., labeled rows and columns).

2. NumPy:

- Optimized for **numerical computations**.
- Offers mathematical functions like matrix multiplication, matrix decomposition, and eigenvalue computations, which pandas does not directly support.

### 6.11.2 Converting a NumPy Array to a pandas DataFrame

You have numerical data stored in a NumPy array and want to apply pandas functionalities like labeling rows/columns or analyzing tabular data.

```

# Create a NumPy array
data = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

# Convert to a DataFrame
df = pd.DataFrame(data, columns=['A', 'B', 'C'])

# Display the DataFrame
print(df)

```

	A	B	C
0	10	20	30
1	40	50	60
2	70	80	90

### Key Points

- Use the `pd.DataFrame()` constructor to create a DataFrame from a NumPy array.
- You can specify:
  - **Column names** using the `columns` parameter.
  - **Row labels** using the `index` parameter.
- If `columns` or `index` are not specified, pandas assigns default labels:
  - Columns: 0, 1, 2, ...
  - Rows: 0, 1, 2, ...

### 6.11.3 Converting a pandas DataFrame to a NumPy Array

You have a pandas DataFrame and want to perform NumPy operations for optimized numerical processing.

```

# Convert DataFrame to NumPy array
array = df.to_numpy()

# Display the NumPy array
print(array)

```

```

[[10 20 30]
 [40 50 60]
 [70 80 90]]

```

**Key Points** \* Use `.to_numpy()` for conversion. \* `.values` is an older method but still works (not recommended for newer code).

#### 6.11.4 Preserving Index and Column Information

When converting a DataFrame to an array and back, ensure row/column labels are retained.

```
# Convert DataFrame to NumPy array
array_with_labels = df.to_numpy()

# Convert back to a DataFrame
df_restored = pd.DataFrame(array_with_labels, columns=df.columns, index=df.index)

# Display the restored DataFrame
print(df_restored)
```

```
      A    B    C
0  10  20  30
1  40  50  60
2  70  80  90
```

Note that you need to manually pass `columns` and `index` back when re-creating the DataFrame.

#### 6.11.5 Practice User Cases

##### 6.11.5.1 Case 1: Adding labels to a NumPy Array

```
# NumPy array without labels
data = np.random.rand(4, 3)

# Add labels by converting to a DataFrame
df_with_labels = pd.DataFrame(data, columns=['Feature1', 'Feature2', 'Feature3'])

print(df_with_labels)
```

```
      Feature1  Feature2  Feature3
0  0.867644  0.478449  0.684584
1  0.378309  0.860889  0.096055
2  0.168311  0.981326  0.685720
3  0.459235  0.340368  0.396904
```

#### 6.11.5.2 Case 2: Perform Numerical Computations

```
# DataFrame for tabular manipulation
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Convert to NumPy for computations
array = df.to_numpy()

# Perform a NumPy operation
array_squared = np.square(array)

# Convert back to a DataFrame
df_squared = pd.DataFrame(array_squared, columns=df.columns)

print(df_squared)
```

```
A    B
0    1   16
1    4   25
2    9   36
```

#### 6.11.6 Best Practices

- **Check Data Types:** Ensure the data in your DataFrame is numerical before converting to NumPy for computations.

```
print(df.dtypes)
```

```
A    int64
B    int64
dtype: object
```

- **Handle Missing Values:** Fill or drop missing values (NaN) in DataFrames before conversion to avoid errors in NumPy.

```
df = df.fillna(0)
```

- **Optimize for Performance:** Use NumPy for heavy numerical computations and pandas for data manipulation.
- **Convert Back to pandas for Analysis:** After computations, convert the NumPy array back to pandas if you need labeled data for further analysis or visualization.

### 6.11.7 Comparison Table

Operation	pandas DataFrame	NumPy Array
Row/Column Labels	Yes	No
Indexing	Label-based ( <code>loc</code> , <code>iloc</code> )	Integer-based only
Mathematical Ops	Slower (handles metadata)	Faster
Missing Data	Supports ( <code>NaN</code> )	Does not support ( <code>NaN</code> )

## 6.12 Random Number Generation in NumPy

NumPy provides a variety of functions for generating random numbers through its `numpy.random` module. These functions can generate random numbers for different distributions and array shapes, making them essential for simulations, statistical sampling, and machine learning applications.

### 6.12.1 Key Functions for Random Number Generation in NumPy

1. `np.random.rand()`: Generates random numbers from a uniform distribution between 0 and 1.

```
# Generate a 2x3 array of random values between 0 and 1
rand_array = np.random.rand(2, 3)
print(rand_array)
```

```
[[0.59160333 0.1864713  0.70446818]
 [0.46476414 0.75172527 0.2225734 ]]
```

2. `np.random.randn()`: Generates random numbers from a standard normal distribution (mean = 0, standard deviation = 1). It is useful for simulating Gaussian distributed data

```
normal_array = np.random.randn(3, 3)
print(normal_array)
```

```
[[ 0.96287539 -0.57932461 -0.25728454]
 [ 0.33379165  0.01331971 -0.19328112]
 [-0.79966965 -1.20325527  1.50081961]]
```

NumPy's `random` module can be used to generate arrays of random numbers from several different probability distributions. For example, a 3x5 array of uniformly distributed random numbers can be generated using the `uniform` function of the `random` module.

3. `np.random.randint()`: Generate random integers within a specific range, you can specify `low` and `high` values and the shape of the output array

```
# Generate a 4x4 matrix of random integers between 10 and 20
int_array = np.random.randint(10, 20, (4, 4))
print(int_array)
```

```
[[13 17 15 18]
 [10 11 13 15]
 [13 12 12 13]
 [17 14 11 16]]
```

4. `np.random.choice()`: Random selects elements from an input array

```
# Select 5 random elements from the array [1, 2, 3, 4, 5] with replacement
choice_array = np.random.choice([1, 2, 3, 4, 5], size=5, replace=True)
print(choice_array)
```

```
[4 3 5 2 2]
```

5. `np.random.uniform()`: Generates random floating-point numbers between a specified range (`low`, `high`) that follow uniform distribution

```
# Generate 6 random numbers from a normal distribution with mean=10 and std=2
custom_normal = np.random.normal(10, 2, size=6)
print(custom_normal)
```

```
[ 8.96337061  9.80411528  7.1623284   9.58883996 11.82330829 10.53508984]
```

```
# Generate a 1D array of 5 random numbers between -5 and 5
uniform_array = np.random.uniform(-5, 5, size=5)
print(uniform_array)
```

```
[-4.76424391 -2.90776829  4.56871733  2.3230159  -4.72083461]
```

6. `np.random.normal()`: Generates random numbers from a normal distribution with a specified mean and standard deviation.

```
np.random.uniform(size = (3,5))
```

```
array([[0.66198756, 0.76365694, 0.79924355, 0.80723229, 0.0661366 ],
       [0.5777333 , 0.49288606, 0.02914087, 0.61883579, 0.04236321],
       [0.2049758 , 0.9819383 , 0.65983787, 0.87025466, 0.39354816]])
```

For a full list, please check the [official website](#)

Random numbers can also be generated by Python's built-in `random` module. However, it generates one random number at a time, which makes it much slower than NumPy's random module.

## 6.13 Independent Practice:

### 6.13.1 Practice exercise 1

#### 6.13.1.1

Read the coordinates of the capital cities of the world from <https://gist.github.com/ofou/df09a6834a8421b4f376c>.

Task 1: Use NumPy to print the name and coordinates of the capital city closest to the US capital - Washington DC.

Note that:

1. The *Country Name* for US is given as *United States of America* in the data.
2. The ‘closeness’ of capital cities from the US capital is based on the Euclidean distance of their coordinates to those of the US capital.

#### Hints:

1. Use the `to_numpy()` function of the *Pandas DataFrame* class to convert a DataFrame to a Numpy array
2. Use *broadcasting* to compute the euclidean distance of capital cities from Washington DC.
3. Use `np.argmin` to locate the index of the minimum value in an array
4. Exclude the capital itself to avoid trivial zero values in the results.

#### Solution:

```

capital_cities = pd.read_csv('./datasets/country-capital-lat-long-population.csv')
coordinates_capital_cities = capital_cities[['Latitude', 'Longitude']].to_numpy()
us_coordinates = capital_cities.loc[capital_cities['Country']=='United States of America', ['Latitude', 'Longitude']]

#Broadcasting
distance_from_DC = np.sqrt(np.sum((us_coordinates-coordinates_capital_cities)**2, axis=1))

#Assigning a high value of distance to DC, otherwise it will itself be selected as being closest
distance_from_DC[distance_from_DC==0]=9999
closest_capital_index = np.argmin(distance_from_DC)
print("Closest capital city is: ", capital_cities.loc[closest_capital_index, 'Capital City'])
print("Coordinates of the closest capital city are: ", coordinates_capital_cities[closest_capital_index])

```

Closest capital city is: Ottawa-Gatineau  
Coordinates of the closest capital city are: [ 45.4166 -75.698 ]

Task 2: Use NumPy to:

1. Print the names of the countries of the top 10 capital cities closest to the US capital - Washington DC.
2. Create and print a NumPy array containing the coordinates of the top 10 cities.

**Hint:** Use the *concatenate()* function from the *NumPy* library to stack the coordinates of the top 10 cities.

```

top10_cities_coordinates = coordinates_capital_cities[closest_capital_index,:].reshape(1,2)
print("Top 10 countries closest to Washington DC are:\n Canada")
for i in range(9):
    distance_from_DC[closest_capital_index]=9999
    closest_capital_index = np.argmin(distance_from_DC)
    print(capital_cities.loc[closest_capital_index, 'Country'])
    top10_cities_coordinates=np.concatenate((top10_cities_coordinates,coordinates_capital_cities[closest_capital_index]),axis=0)
print("Coordinates of the top 10 cities closest to US are: \n",top10_cities_coordinates)

```

Top 10 countries closest to Washington DC are:  
Canada  
Cuba  
Turks and Caicos Islands  
Cayman Islands  
Haiti

```

Jamaica
Dominican Republic
Saint Pierre and Miquelon
Puerto Rico
United States Virgin Islands
Coordinates of the top 10 cities closest to US are:
[[ 32.2915 -64.778 ]
 [ 23.1195 -82.3785]
 [ 21.4612 -71.1419]
 [ 19.2866 -81.3744]
 [ 18.5392 -72.335 ]
 [ 17.997 -76.7936]
 [ 18.4896 -69.9018]
 [ 46.7738 -56.1815]
 [ 18.4663 -66.1057]
 [ 18.3419 -64.9307]]

```

### 6.13.2 Practice exercise 2:

This exercise will show vectorized computations with NumPy. Vectorized computations help perform computations more efficiently, and also make the code concise.

**Q:** Read the (1) quantities of roll, bun, cake and bread required by 3 people - Ben, Barbara & Beth, from *food\_quantity.csv*, (2) price of these food items in two shops - Target and Kroger, from *price.csv*. Find out which shop should each person go to minimize their expenses.

```
#Reading the datasets on food quantity and price
import pandas as pd
food_qty = pd.read_csv('../data/food_quantity.csv',index_col=0)
price = pd.read_csv('../data/price.csv',index_col=0)
```

```
food_qty
```

	roll	bun	cake	bread
Person				
Ben	6	5	3	1
Barbara	3	6	2	2
Beth	3	4	3	1

```
price
```

Item	Target	Kroger
roll	1.5	1.0
bun	2.0	2.5
cake	5.0	4.5
bread	16.0	17.0

First, let's start from a simple problem. We'll compute the expenses of Ben if he prefers to buy all food items from Target

```
%%time
# write a for loop to calculate the total cost of Ben's food if he shops at the Target store
total_cost = 0
for food in food_qty.columns:

    total_cost += food_qty.loc['Ben',food]*price.loc[food,'Target']
total_cost
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

```
np.float64(50.0)
```

```
%%time
# using numpy
total_cost = np.sum(food_qty.loc['Ben',]*price.loc[:, 'Target'])
total_cost
```

```
CPU times: total: 0 ns
Wall time: 1 ms
```

```
np.float64(50.0)
```

Ben will spend \$50 if he goes to Target

Now, let's add another layer of complication. We'll compute Ben's expenses for both stores - Target and Kroger

```

%%time
# using loops to calculate the total cost of food for Ben for all stores
total_cost = {}
for store in price.columns:
    total_cost[store] = 0
    for food in food_qty.columns:
        total_cost[store] += food_qty.loc['Ben', food]*price.loc[food, store]
total_cost

```

CPU times: total: 0 ns  
Wall time: 1 ms

{'Target': np.float64(50.0), 'Kroger': np.float64(49.0)}

```

%%time
# using numpy
total_cost = np.dot(food_qty.loc['Ben',:],price.loc[:,:])
total_cost

```

CPU times: total: 0 ns  
Wall time: 0 ns

array([50., 49.])

Ben will spend \$50 if he goes to Target, and \$49 if he goes to Kroger. Thus, he should choose Kroger.

Now, let's add the final layer of complication, and solve the problem. We'll compute everyone's expenses for both stores - Target and Kroger

```

%%timeit
store_expense = pd.DataFrame(0.0, columns=price.columns, index = food_qty.index)
for person in store_expense.index:
    for store in store_expense.columns:
        for food in food_qty.columns:
            store_expense.loc[person, store] += food_qty.loc[person, food]*price.loc[food, store]

store_expense

```

1.51 ms ± 15.3 s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

```

%%timeit
# using matrix multiplication in numpy
pd.DataFrame(np.dot(food_qty.values, price.values), columns=price.columns, index=food_qty.index)

11 s ± 102 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

```

Based on the above table, Ben should go to Kroger, Barbara to Target and Beth can go to either store.

As the complexity of operations increases, the number of nested for-loops tends to grow, making the code more cumbersome and difficult to manage. In contrast, leveraging NumPy arrays allows for concise and straightforward implementation, regardless of the complexity. Vectorized computations are not only cleaner but also significantly faster, offering a more efficient solution. To take advantage of this, you first need to convert a pandas DataFrame to a NumPy array for matrix multiplication. Once the computation is complete, you can convert the results back to a pandas DataFrame for further analysis or display

### 6.13.3 Practice exercise 3

Use matrix multiplication to find the average IMDB rating and average Rotten tomatoes rating for each genre - comedy, action, drama and horror. Use the data: *movies\_cleaned.csv*. Which is the most preferred genre for IMDB users, and which is the least preferred genre for Rotten Tomatoes users?

- Hint:**
1. Create two matrices - one containing the IMDB and Rotten Tomatoes ratings, and the other containing the genre flags (comedy/action/drama/horror).
  2. Multiply the two matrices created in 1.
  3. Divide each row/column of the resulting matrix by a vector having the number of ratings in each genre to get the average rating for the genre.

**Solution:**

```

data = pd.read_csv('../Data/movies_cleaned.csv')
data.head()

```

	Title	IMDB Rating	Rotten Tomatoes Rating	Running Time min	Release Date	US G
0	Broken Arrow	5.8	55	108	Feb 09 1996	7064
1	Brazil	8.0	98	136	Dec 18 1985	9929
2	The Cable Guy	5.8	52	95	Jun 14 1996	6024

	Title	IMDB Rating	Rotten Tomatoes Rating	Running Time min	Release Date	US G
3	Chain Reaction	5.2	13	106	Aug 02 1996	2122
4	Clash of the Titans	5.9	65	108	Jun 12 1981	3000

```
# Getting ratings of all movies
drating = data[['IMDB Rating','Rotten Tomatoes Rating']]
drating_num = drating.to_numpy() #Converting the data to NumPy array
drating_num
```

```
array([[ 5.8, 55. ],
       [ 8. , 98. ],
       [ 5.8, 52. ],
       ...,
       [ 7. , 65. ],
       [ 5.7, 26. ],
       [ 6.7, 82. ]])
```

```
# Getting the matrix indicating the genre of all movies
dgenre = data.iloc[:,8:12]
dgenre_num = dgenre.to_numpy() #Converting the data to NumPy array
dgenre_num
```

```
array([[0, 1, 0, 0],
       [1, 0, 0, 0],
       [1, 0, 0, 0],
       ...,
       [1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 1, 0, 0]])
```

We'll first find the total IMDB and Rotten tomatoes ratings for all movies of each genre, and then divide them by the number of movies of the corresponding genre to find the average rating for the genre.

For finding the total IMDB and Rotten tomatoes ratings, we'll multiply `drating_num` with `dgenre_num`. However, before multiplying, we'll check if their shapes are compatible for matrix multiplication.

```
#Shape of drating_num  
drating_num.shape
```

```
(980, 2)
```

```
#Shape of dgenre_num  
dgenre_num.shape
```

```
(980, 4)
```

Note that the above shapes are not compatible for matrix multiplication. We'll transpose `dgenre_num` to make the shapes compatible.

```
#Total IMDB and Rotten tomatoes ratings for each genre  
ratings_sum_genre = drating_num.T.dot(dgenre_num)  
ratings_sum_genre
```

```
array([[ 1785.6,  1673.1,  1630.3,   946.2],  
       [14119. , 13725. , 14535. ,  6533. ]])
```

```
#Number of movies in the data will be stored in 'rows', and number of columns stored in 'cols'  
rows, cols = data.shape
```

```
#Getting number of movies in each genre  
movies_count_genre = dgenre_num.T.dot(np.ones(rows))  
movies_count_genre
```

```
array([302., 264., 239., 154.])
```

```
#Finding the average IMDB and average Rotten tomatoes ratings for each genre  
ratings_sum_genre/movies_count_genre
```

```
array([[ 5.91258278,  6.3375     ,  6.82133891,  6.14415584],  
       [46.75165563, 51.98863636, 60.81589958, 42.42207792]])
```

```
pd.DataFrame(ratings_sum_genre/movies_count_genre,columns = ['comedy','Action','drama','horror'],  
             index = ['IMDB Rating','Rotten Tomatoes Rating'])
```

	comedy	Action	drama	horror
IMDB Rating	5.912583	6.337500	6.821339	6.144156
Rotten Tomatoes Rating	46.751656	51.988636	60.815900	42.422078

IMDB users prefer *drama*, and are amused the least by *comedy* movies, on an average. However, Rotten tomatoes critics would rather watch *comedy* than *horror* movies, on an average.

#### 6.13.4 Practice exercise 4:

##### Random number generation using NumPy

Suppose 500 people eat at Food cart 1, and another 500 eat at Food cart 2, everyday.

The waiting time at Food cart 2 has a normal distribution with mean 8 minutes and standard deviation 3 minutes, while the waiting time at Food cart 1 has a uniform distribution with minimum 5 minutes and maximum 25 minutes.

Simulate a dataset containing waiting times for 500 ppl for 30 days in each of the food joints. Assume that the waiting times are measured simultaneously at a certain time in both places, i.e., the observations are paired.

**On how many days is the average waiting time at Food cart 2 higher than that at Food cart 1?**

**What percentage of times the waiting time at Food cart 2 was higher than the waiting time at Food cart 1?**

Try both approaches: (1) Using loops to generate data, (2) numpy array to generate data. Compare the time taken in both approaches.

```
import time as tm

#Method 1: Using loops
start_time = tm.time() #Current system time

#Initializing waiting times for 500 ppl over 30 days
waiting_times_FoodCart1 = pd.DataFrame(0,index=range(500),columns=range(30)) #FoodCart1
waiting_times_FoodCart2 = pd.DataFrame(0,index=range(500),columns=range(30)) #FoodCart2
import random as rm
for i in range(500): #Iterating over 500 ppl
    for j in range(30): #Iterating over 30 days
        waiting_times_FoodCart2.iloc[i,j] = rm.gauss(8,3) #Simulating waiting time in FoodCart2
        waiting_times_FoodCart1.iloc[i,j] = rm.uniform(5,25) #Simulating waiting time in FoodCart1
```

```

time_diff = waiting_times_FoodCart2-waiting_times_FoodCart1

print("On ",sum(time_diff.mean()>0)," days, the average waiting time at FoodCart2 higher than
print("Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 = ",)
end_time = tm.time() #Current system time
print("Time taken = ", end_time-start_time)

```

On 0 days, the average waiting time at FoodCart2 higher than that at FoodCart1  
Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 = 16.226660000000001  
Time taken = 4.521248817443848

```

#Method 2: Using NumPy arrays
start_time = tm.time()
waiting_time_FoodCart2 = np.random.normal(8,3,size = (500,30)) #Simultaneously generating the waiting times for 500 customers at FoodCart2
waiting_time_FoodCart1 = np.random.uniform(5,25,size = (500,30)) #Simultaneously generating the waiting times for 500 customers at FoodCart1
time_diff = waiting_time_FoodCart2-waiting_time_FoodCart1
print("On ",(time_diff.mean()>0).sum()," days, the average waiting time at FoodCart2 higher than
print("Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 = ",)
end_time = tm.time()
print("Time taken = ", end_time-start_time)

```

On 0 days, the average waiting time at FoodCart2 higher than that at FoodCart1  
Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 = 16.52 %  
Time taken = 0.008000850677490234

The approach with NumPy is much faster than the one with loops.

### 6.13.5 Practice exercise 5

**Bootstrapping:** Find the 95% confidence interval of mean profit for ‘Action’ movies, using Bootstrapping.

Bootstrapping is a non-parametric method for obtaining confidence interval. Use the algorithm below to find the confidence interval:

1. Find the profit for each of the ‘Action’ movies. Suppose there are  $N$  such movies. We will have a *Profit* column with  $N$  values.
2. Randomly sample  $N$  values with replacement from the *Profit* column

3. Find the mean of the  $N$  values obtained in (b)
4. Repeat steps (b) and (c)  $M=1000$  times
5. The 95% Confidence interval is the range between the 2.5% and 97.5% percentile values of the 1000 means obtained in (c)  
Use the *movies\_cleaned.csv* dataset.

**Solution:**

```
#Reading data
movies = pd.read_csv('./Datasets/movies_cleaned.csv')

#Filtering action movies
movies_action = movies.loc[movies['Action']==1,:]

#Computing profit of movies
movies_action.loc[:, 'Profit'] = movies_action.loc[:, 'Worldwide Gross'] - movies_action.loc[:, 'Budget']

#Subsetting the profit column
profit_vec = movies_action['Profit']

#Creating a matrix of 1000 samples with replacement from the profit column
bootstrap_samples=np.random.choice(profit_vec,size = (1000,len(profit_vec)))

#Computing the mean of each of the 1000 samples
bootstrap_sample_means = bootstrap_samples.mean(axis=1)

#The confidence interval is the 2.5th and 97.5th percentile of the mean of the 1000 samples
print("Confidence interval = [${}"+str(np.round(np.percentile(bootstrap_sample_means,2.5)/1e6,2))+" million, ${}"+str(np.round(np.percentile(bootstrap_sample_means,97.5)/1e6,2))+" million]
```

Confidence interval = [\$132.53 million, \$182.69 million]

# 7 Pandas

## 7.1 Introduction

Pandas is an essential tool in the data scientist or data analyst's toolkit due to its ability to handle and transform data with ease and efficiency.

Built on top of the NumPy package, Pandas extends the capabilities of NumPy by offering support for working with tabular or heterogeneous data. While NumPy is optimized for working with homogeneous numerical arrays, Pandas is specifically designed for handling structured, labeled data, commonly represented in two-dimensional tables known as DataFrames. There are some similarities between the two libraries. Like NumPy, Pandas provides basic mathematical functionalities such as addition, subtraction, conditional operations, and broadcasting. However, while NumPy focuses on multi-dimensional arrays, Pandas excels in offering the powerful 2D `DataFrame` object for data manipulation.

Data in Pandas is often used to feed statistical analyses in SciPy, visualization functions in Matplotlib, and machine learning algorithms in Scikit-learn.

Typically, the Pandas library is used for:

- Data reading/writing from different sources (CSV, Excel, SQL, etc.).
- Data manipulation, cleaning, and transformation.
- Computing data distribution and summary statistics
- Grouping, aggregation, and pivoting for advanced data analysis.
- Merging, concatenating, and reshaping data.
- Handling time series and datetime data.
- Data visualization and plotting.

Let's import the Pandas library to use its methods and functions.

```
import pandas as pd
```

## 7.2 Pandas data structures - Series and DataFrame

There are two core components of the Pandas library - Series and DataFrame.

A DataFrame is a two-dimensional object - comprising of tabular data organized in rows and columns, where individual columns can be of different value types (numeric / string / boolean etc.). A DataFrame has row labels (also called row indices) which refer to individual rows, and column labels (also called column names) that refer to individual columns. By default, the row indices are integers starting from zero. However, both the row indices and column names can be customized by the user.

Let us read the spotify data - *spotify\_data.csv*, using the Pandas function `read_csv()`.

```
spotify_data = pd.read_csv('../Data/spotify_data.csv')
spotify_data.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popularity
0	16996777	rap	Juice WRLD	96	All Girls Are The Same	0
1	16996777	rap	Juice WRLD	96	Lucid Dreams	0
2	16996777	rap	Juice WRLD	96	Hear Me Calling	0
3	16996777	rap	Juice WRLD	96	Robbery	0
4	5988689	rap	Roddy Ricch	88	Big Stepper	0

The object `spotify_data` is a pandas DataFrame:

```
type(spotify_data)
```

```
pandas.core.frame.DataFrame
```

A Series is a one-dimensional array-like object in pandas that contains a sequence of values, where each value is associated with an index. All the values in a Series must have the same data type. Each column of a DataFrame is Series as shown in the example below.

```
#Extracting song titles from the spotify_songs DataFrame
spotify_songs = spotify_data['track_name']
spotify_songs
```

```
0          All Girls Are The Same
1          Lucid Dreams
2          Hear Me Calling
3          Robbery
4          Big Stepper
...
243185      Stardust
243186      Knockin' A Jug - 78 rpm Version
243187      When It's Sleepy Time Down South
243188      On The Sunny Side Of The Street - Part 2
243189      My Sweet
Name: track_name, Length: 243190, dtype: object
```

```
#The object spotify_songs is a Series
type(spotify_songs)
```

```
pandas.core.series.Series
```

A Series is essentially a column, and a DataFrame is a two-dimensional table made up of a collection of Series

## 7.3 Creating a Pandas Series / DataFrame

- From a Python List or Dictionary
- By Reading Data from a file

### 7.3.1 Creating a Series/DataFrame from Python List or Dictionary

#### 7.3.1.1 Creating a series:

- **From a Python List:** You can create a pandas Series by passing a list of values.

```
#Defining a Pandas Series
series_example = pd.Series(['these','are','english','words'])
series_example
```

```
0      these
1      are
2    english
3    words
dtype: object
```

Note that the default row indices are integers starting from 0. However, the index can be specified with the `index` argument if desired by the user:

```
#Defining a Pandas Series with custom row labels
series_example = pd.Series(['these','are','english','words'], index = range(101,105))
series_example
```

```
101      these
102      are
103    english
104    words
dtype: object
```

- **From a Dictionary:** A dictionary can also be used to create a Series, where keys become index labels.

```
#Dictionary consisting of the GDP per capita of the US from 1960 to 2021 with some missing values
GDP_per_capita_dict = {'1960':3007,'1961':3067,'1962':3244,'1963':3375,'1964':3574,'1965':382}
```

```
#Example 2: Creating a Pandas Series from a Dictionary
GDP_per_capita_series = pd.Series(GDP_per_capita_dict)
GDP_per_capita_series.head()
```

```
1960    3007
1961    3067
1962    3244
1963    3375
1964    3574
dtype: int64
```

### 7.3.1.2 Creating a DataFrame

- **From a list of Python Dictionary:** You can create a DataFrame where keys are column names and values are lists representing column data.

```
#List of dictionary consisting of 52 playing cards of the deck
deck_list_of_dictionaries = [{value:i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

```
#Example 3: Creating a Pandas DataFrame from a List of dictionaries
deck_df = pd.DataFrame(deck_list_of_dictionaries)
deck_df.head()
```

	value	suit
0	2	spades
1	3	spades
2	4	spades
3	5	spades
4	6	spades

- **From a Python Dictionary:** You can create a DataFrame where keys are column names and values are lists representing column data.

```
#Example 4: Creating a Pandas DataFrame from a Dictionary
dict_data = {'A': [1, 2, 3, 4, 5],
             'B': [20, 10, 50, 40, 30],
             'C': [100, 200, 300, 400, 500]}

dict_df = pd.DataFrame(dict_data)
dict_df
```

	A	B	C
0	1	20	100
1	2	10	200
2	3	50	300
3	4	40	400
4	5	30	500

## 7.4 Creating a Series/DataFrame by Reading Data from a File

In the real world, a Pandas DataFrame will typically be created by loading the datasets from existing storage such as SQL Database, CSV file, Excel file, text files, HTML files, etc., as we learned in the previous chapter of the book on Reading data.

- **Using `read_csv()`:** This is one of the most common methods to read data from a CSV file into a pandas DataFrame.
- **Using `read_excel()`:** You can also read data from Excel files.
- **Using `read_json()`:** You can also read data from json files.

## 7.5 Attributes and Methods of a Pandas DataFrame

All attributes and methods of a Pandas DataFrame object can be viewed with the python's built-in `dir()` function.

```
#List of attributes and methods of a Pandas DataFrame
#This code is not executed as the list is too long
# dir(spotify_data)
```

Although we'll see examples of attributes and methods of a Pandas DataFrame, please note that most of these attributes and methods are also applicable to the Pandas Series object.

### 7.5.1 Attributes of a Pandas DataFrame

Some of the attributes of the Pandas DataFrame class are the following.

#### 7.5.1.1 `dtypes`

This attribute is a Series consisting the datatypes of columns of a Pandas DataFrame.

```
spotify_data.dtypes
```

<code>artist_followers</code>	<code>int64</code>
<code>genres</code>	<code>object</code>
<code>artist_name</code>	<code>object</code>
<code>artist_popularity</code>	<code>int64</code>
<code>track_name</code>	<code>object</code>
<code>track_popularity</code>	<code>int64</code>
<code>duration_ms</code>	<code>int64</code>
<code>explicit</code>	<code>int64</code>
<code>release_year</code>	<code>int64</code>
<code>danceability</code>	<code>float64</code>
<code>energy</code>	<code>float64</code>
<code>key</code>	<code>int64</code>

```

loudness          float64
mode              int64
speechiness       float64
acousticness      float64
instrumentalness float64
liveness          float64
valence           float64
tempo              float64
time_signature    int64
dtype: object

```

The table below describes the datatypes of columns in a Pandas DataFrame.

Pandas Type	Native Python Type	Description
object	string	The most general dtype. This datatype is assigned to a column if the column has mixed types (numbers and strings)
int64	int	This datatype is for integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or for integers having a maximum size of 64 bits
float64	float	This datatype is for real numbers. If a column contains integers and NaNs, Pandas will default to float64. This is because the missing values may be a real number
datetime64, timedelta[ns]	N/A (but see the <a href="#">datetime</a> module in Python's standard library)	Values meant to hold time data. This datatype is useful for time series analysis

### 7.5.1.2 [columns](#)

This attribute consists of the column labels (or column names) of a Pandas DataFrame.

```
spotify_data.columns
```

```
Index(['artist_followers', 'genres', 'artist_name', 'artist_popularity',
       'track_name', 'track_popularity', 'duration_ms', 'explicit',
       'release_year', 'danceability', 'energy', 'key', 'loudness', 'mode',
       'speechiness', 'acousticness', 'instrumentalness', 'liveness',
       'valence', 'tempo', 'time_signature'],
      dtype='object')
```

#### 7.5.1.3 `index`

This attribute consists of the row labels (or row indices) of a Pandas DataFrame.

```
spotify_data.index
```

```
RangeIndex(start=0, stop=243190, step=1)
```

#### 7.5.1.4 `axes`

This is a list of length two, where the first element is the row labels, and the second element is the columns labels. In other words, this attribute combines the information in the attributes - `index` and `columns`.

```
spotify_data.axes
```

```
[RangeIndex(start=0, stop=243190, step=1),
 Index(['artist_followers', 'genres', 'artist_name', 'artist_popularity',
        'track_name', 'track_popularity', 'duration_ms', 'explicit',
        'release_year', 'danceability', 'energy', 'key', 'loudness', 'mode',
        'speechiness', 'acousticness', 'instrumentalness', 'liveness',
        'valence', 'tempo', 'time_signature'],
      dtype='object')]
```

#### 7.5.1.5 `ndim`

As in NumPy, this attribute specifies the number of dimensions. However, unlike NumPy, a Pandas DataFrame has a fixed dimension of 2, and a Pandas Series has a fixed dimension of 1.

```
spotify_data.ndim
```

2

#### 7.5.1.6 size

This attribute specifies the number of elements in a DataFrame. Its value is the product of the number of rows and columns.

```
spotify_data.size
```

5106990

#### 7.5.1.7 shape

This is a tuple consisting of the number of rows and columns in a Pandas DataFrame.

```
spotify_data.shape
```

(243190, 21)

#### 7.5.1.8 values

This provides a NumPy representation of a Pandas DataFrame.

```
spotify_data.values
```

```
array([[16996777, 'rap', 'Juice WRLD', ..., 0.203, 161.991, 4],  
       [16996777, 'rap', 'Juice WRLD', ..., 0.218, 83.903, 4],  
       [16996777, 'rap', 'Juice WRLD', ..., 0.499, 88.933, 4],  
       ...,  
       [2256652, 'jazz', 'Louis Armstrong', ..., 0.37, 105.093, 4],  
       [2256652, 'jazz', 'Louis Armstrong', ..., 0.576, 101.279, 4],  
       [2256652, 'jazz', 'Louis Armstrong', ..., 0.816, 105.84, 4]],  
      dtype=object)
```

## 7.5.2 Methods of a Pandas DataFrame

Some of the commonly used methods of the Pandas DataFrame class are the following.

### 7.5.2.1 `head()`

Prints the first  $n$  rows of a DataFrame.

```
spotify_data.head(2)
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popularity
0	16996777	rap	Juice WRLD	96	All Girls Are The Same	0
1	16996777	rap	Juice WRLD	96	Lucid Dreams	0

### 7.5.2.2 `tail()`

Prints the last  $n$  rows of a DataFrame.

```
spotify_data.tail(3)
```

	artist_followers	genres	artist_name	artist_popularity	track_name
243187	2256652	jazz	Louis Armstrong	74	When It's Sleepy Time Down South
243188	2256652	jazz	Louis Armstrong	74	On The Sunny Side Of The Street
243189	2256652	jazz	Louis Armstrong	74	My Sweet

### 7.5.2.3 `describe()`

Print summary statistics of a Pandas DataFrame, as seen in chapter 3 on Reading Data.

```
spotify_data.describe()
```

	artist_followers	artist_popularity	track_popularity	duration_ms	explicit	release_year
count	2.431900e+05	243190.000000	243190.000000	2.431900e+05	243190.000000	243190.000000
mean	1.960931e+06	65.342633	36.080772	2.263209e+05	0.050039	1992.475258
std	5.028746e+06	10.289182	16.476836	9.973214e+04	0.218026	18.481463

	artist_followers	artist_popularity	track_popularity	duration_ms	explicit	release_year
min	2.300000e+01	51.000000	0.000000	3.344000e+03	0.000000	1923.000000
25%	1.832620e+05	57.000000	25.000000	1.776670e+05	0.000000	1980.000000
50%	5.352520e+05	64.000000	36.000000	2.188670e+05	0.000000	1994.000000
75%	1.587332e+06	72.000000	48.000000	2.645465e+05	0.000000	2008.000000
max	7.890023e+07	100.000000	99.000000	4.995083e+06	1.000000	2021.000000

#### 7.5.2.4 `max()`/`min()`

Returns the max/min values of numeric columns. If the function is applied on non-numeric columns, it will return the maximum/minimum value based on the order of the alphabet.

```
#The max() method applied on a Series
spotify_data['artist_popularity'].max()
```

```
np.int64(100)
```

```
#The max() method applied on a DataFrame
spotify_data.max()
```

artist_followers	78900234
genres	rock
artist_name	OSN
artist_popularity	100
track_name	days gone by
track_popularity	99
duration_ms	4995083
explicit	1
release_year	2021
danceability	0.988
energy	1.0
key	11
loudness	3.744
mode	1
speechiness	0.969
acousticness	0.996
instrumentalness	1.0
liveness	1.0
valence	1.0

```
tempo           243.507
time_signature      5
dtype: object
```

### 7.5.2.5 `mean()`/`median()`

Returns the mean/median values of numeric columns.

```
# apply median to the numeric columns

spotify_data.select_dtypes(include='number').median()
```

```
artist_followers      535252.000000
artist_popularity     64.000000
track_popularity      36.000000
duration_ms           218867.000000
explicit                0.000000
release_year          1994.000000
danceability           0.579000
energy                  0.591000
key                     5.000000
loudness                 -8.645000
mode                     1.000000
speechiness              0.043100
acousticness             0.325000
instrumentalness        0.000011
liveness                  0.141000
valence                   0.560000
tempo                      118.002000
time_signature            4.000000
dtype: float64
```

### 7.5.2.6 `std()`

Returns the standard deviation of numeric columns.

```
spotify_data.select_dtypes(include='number').std()
```

```
artist_followers      5.028746e+06
artist_popularity     1.028918e+01
```

```
track_popularity      1.647684e+01
duration_ms           9.973214e+04
explicit              2.180260e-01
release_year          1.848146e+01
danceability          1.594436e-01
energy                 2.366309e-01
key                     3.532546e+00
loudness               4.449731e+00
mode                    4.698771e-01
speechiness            1.980684e-01
acousticness           3.211417e-01
instrumentalness       2.095551e-01
liveness                1.980759e-01
valence                  2.500172e-01
tempo                   2.986422e+01
time_signature          4.580822e-01
dtype: float64
```

#### 7.5.2.7 `sample(n)`

Returns  $n$  random observations from a Pandas DataFrame.

```
spotify_data.sample(4)
```

	artist_followers	genres	artist_name	artist_popularity
95984	1179716	pop & rock	mor ve ötesi	65
201066	1582426	pop & rock	Andrés Calamaro	73
15002	666680	rock	Indio Solari y los Fundamentalistas del Aire A...	61
186925	295211	jazz	Stan Getz	68

#### 7.5.2.8 `dropna()`

Drops all observations with at least one missing value.

```
#This code is not executed to avoid printing a large table
spotify_data.dropna()
```

	artist_followers	genres	artist_name	artist_popularity	track_name
0	16996777	rap	Juice WRLD	96	All Girls Are The Same
1	16996777	rap	Juice WRLD	96	Lucid Dreams
2	16996777	rap	Juice WRLD	96	Hear Me Calling
3	16996777	rap	Juice WRLD	96	Robbery
4	5988689	rap	Roddy Ricch	88	Big Stepper
...	...	...	...	...	...
243185	2256652	jazz	Louis Armstrong	74	Stardust
243186	2256652	jazz	Louis Armstrong	74	Knockin' A Jug - 78 rpm Version
243187	2256652	jazz	Louis Armstrong	74	When It's Sleepy Time Down South
243188	2256652	jazz	Louis Armstrong	74	On The Sunny Side Of The Street
243189	2256652	jazz	Louis Armstrong	74	My Sweet

### 7.5.2.9 unique()

This function provides the unique values of a Series. For example, let us find the number of unique genres of songs in the spotify dataset:

```
spotify_data.genres.unique()
```

```
array(['rap', 'pop', 'miscellaneous', 'metal', 'hip hop', 'rock',
       'pop & rock', 'hoerspiel', 'folk', 'electronic', 'jazz', 'country',
       'latin'], dtype=object)
```

### 7.5.2.10 value\_counts()

This function provides the number of observations of each value of a Series. For example, let us find the number of songs of each genre in the spotify dataset:

```
spotify_data.genres.value_counts()
```

genres	
pop	70441
rock	49785
pop & rock	43437
miscellaneous	35848
jazz	13363
hoerspiel	12514
hip hop	7373

```

folk           2821
latin          2125
rap            1798
metal          1659
country        1236
electronic     790
Name: count, dtype: int64

```

More than half the songs in the dataset are *pop*, *rock* or *pop & rock*.

### 7.5.2.11 `isin()`

This function provides a boolean Series indicating the position of certain values in a Series. The function is helpful in sub-setting data. For example, let us subset the songs that are either *latin*, *rap*, or *metal*:

```

latin_rap_metal_songs = spotify_data.loc[spotify_data.genres.isin(['latin','rap','metal'])]
latin_rap_metal_songs.head()

```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popularity
0	16996777	rap	Juice WRLD	96	All Girls Are The Same	0
1	16996777	rap	Juice WRLD	96	Lucid Dreams	0
2	16996777	rap	Juice WRLD	96	Hear Me Calling	0
3	16996777	rap	Juice WRLD	96	Robbery	0
4	5988689	rap	Roddy Ricch	88	Big Stepper	0

## 7.6 Data manipulations with Pandas

### 7.6.1 Sub-setting data

In the chapter on reading data, we learned about different ways for subsetting data, specifically:

- Selecting columns: `df['column_name']`, `df[['col1', 'col2']]`
- Selecting rows by label: `df.loc[]`
- Selecting rows by integer position: `df.iloc[]`
- Selecting by condition (Boolean indexing): `df[df['column_name'] > value]`
- Selecting by multiple conditions: `df[(df['column_name'] > value) & (df['other_column'] == another_value)]`

### 7.6.2 Setting and Resetting indices

In pandas, the `index` of a dataframe can be accessed by using `index` attribute, it represents the row labels of a DataFrame. When you create a DataFrame without specifying an index, pandas automatically assigns a RangeIndex, which is a sequence of integers starting from 0.

```
# read the txt data
bestseller_data = pd.read_csv('../Data/bestseller_books.txt', sep=';')
bestseller_data.head()
```

	Unnamed: 0.1	Unnamed: 0	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

```
# accessing the index of the DataFrame
bestseller_data.index
```

```
RangeIndex(start=0, stop=550, step=1)
```

Alternatively, we can use the `index_col` parameter in the `pd.read_csv()` function to set a specific column as the index when reading a CSV file in pandas. This allows us to directly specify which column should be used as the index of the resulting DataFrame.

```
bestseller_index_data = pd.read_csv('../Data/bestseller_books.txt', sep=';', index_col='Unnamed: 0')
bestseller_index_data.head()
```

	Unnamed: 0	Name	Author	User Rating
0	0	10-Day Green Smoothie Cleanse	JJ Smith	4.7
1	1	11/22/63: A Novel	Stephen King	4.6
2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7
3	3	1984 (Signet Classics)	George Orwell	4.7
4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8

```
bestseller_index_data.index
```

```
Index([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
       ...
      540, 541, 542, 543, 544, 545, 546, 547, 548, 549],
     dtype='int64', length=550)
```

The `index` is crucial when subsetting your dataset using methods like `loc[]`, as it identifies the rows you want to select based on their labels.

```
# get the book that User rating is greater than 4.5
bestseller_index_data.loc[bestseller_index_data['User Rating']== 4.9].head()
```

	Unnamed: 0	Name	Author	User Rating	Rev
40	40	Brown Bear, Brown Bear, What Do You See?	Bill Martin Jr.	4.9	143
41	41	Brown Bear, Brown Bear, What Do You See?	Bill Martin Jr.	4.9	143
81	81	Dog Man and Cat Kid: From the Creator of Captain Underpants	Dav Pilkey	4.9	506
82	82	Dog Man: A Tale of Two Kitties: From the Creator of Captain Underpants	Dav Pilkey	4.9	478
83	83	Dog Man: Brawl of the Wild: From the Creator of Captain Underpants	Dav Pilkey	4.9	723

Additionally, you can modify the index at any point after the DataFrame is created by using the `set_index()` function to re-index the rows based on existing column(s) of the DataFrame.

```
bestseller_data_reindexed = bestseller_data.set_index('Unnamed: 0.1')
bestseller_data_reindexed.index
```

```
Index([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
       ...
      540, 541, 542, 543, 544, 545, 546, 547, 548, 549],
     dtype='int64', name='Unnamed: 0.1', length=550)
```

Note that, the index column does not have to uniquely identify each row. For example, if we need to subset data by `Author` frequently in our analysis, we can set it as our index

```
bestseller_author_index = bestseller_data.set_index('Author')
bestseller_author_index.loc['Jen Sincero']
```

	Unnamed: 0.1	Unnamed: 0	Name	User Rating
Author				
Jen Sincero	546	546	You Are a Badass: How to Stop Doubting Your Gr...	4.7
Jen Sincero	547	547	You Are a Badass: How to Stop Doubting Your Gr...	4.7
Jen Sincero	548	548	You Are a Badass: How to Stop Doubting Your Gr...	4.7
Jen Sincero	549	549	You Are a Badass: How to Stop Doubting Your Gr...	4.7

```
bestseller_author_index.index
```

```
Index(['JJ Smith', 'Stephen King', 'Jordan B. Peterson', 'George Orwell',
       'National Geographic Kids', 'George R. R. Martin',
       'George R. R. Martin', 'Amor Towles', 'James Comey', 'Fredrik Backman',
       ...
       'R. J. Palacio', 'R. J. Palacio', 'R. J. Palacio', 'R. J. Palacio',
       'R. J. Palacio', 'Jeff Kinney', 'Jen Sincero', 'Jen Sincero',
       'Jen Sincero', 'Jen Sincero'],
      dtype='object', name='Author', length=550)
```

You can revert it back to a default index using `reset_index()`. It converts the current index into a regular column and resets the index to a default integer sequence.

```
bestseller_author_index.reset_index()
```

	Author	Unnamed: 0.1	Unnamed: 0	Name
0	JJ Smith	0	0	10-Day Green Smoothie Cleanse
1	Stephen King	1	1	11/22/63: A Novel
2	Jordan B. Peterson	2	2	12 Rules for Life: An Antidote to Chaos
3	George Orwell	3	3	1984 (Signet Classics)
4	National Geographic Kids	4	4	5,000 Awesome Facts (About Everything!) (I
...	...	...	...	...
545	Jeff Kinney	545	545	Wrecking Ball (Diary of a Wimpy Kid Book
546	Jen Sincero	546	546	You Are a Badass: How to Stop Doubting Y
547	Jen Sincero	547	547	You Are a Badass: How to Stop Doubting Y
548	Jen Sincero	548	548	You Are a Badass: How to Stop Doubting Y
549	Jen Sincero	549	549	You Are a Badass: How to Stop Doubting Y

In summary, the index is a powerful feature of pandas for organizing and accessing data, and it can be set, changed, or reset as needed to suit your data analysis.

Later, you will learn about **hierarchical indexing**, which allows you to create multi-level indices by passing multiple columns.

### 7.6.3 Dropping a column

In pandas, you can drop a column from a DataFrame using the `drop()` method. This is useful when you want to remove columns that are redundant or when you want to simplify your DataFrame by excluding unnecessary data.

```
df.drop(columns='column_name')
df.drop(columns= ['col_1', 'col_2'])
```

```
# drop the repeated rows in the bestseller dataframe
bestseller_df = bestseller_index_data.drop(columns = 'Unnamed: 0')
bestseller_df.head()
```

	Name	Author	User Rating	Reviews
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350
1	11/22/63: A Novel	Stephen King	4.6	2052
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979
3	1984 (Signet Classics)	George Orwell	4.7	21424
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665

```
# another way to drop the column
bestseller_index_data.drop('Unnamed: 0', axis=1).head()
```

	Name	Author	User Rating	Reviews
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350
1	11/22/63: A Novel	Stephen King	4.6	2052
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979
3	1984 (Signet Classics)	George Orwell	4.7	21424
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665

**Note:** The `df.drop()` function can be used to remove rows by specifying row labels, index values, or based on conditions. It includes an `axis` parameter, which defaults to dropping rows (`axis=0`). Feel free to explore its additional options.

#### 7.6.4 Adding a column

In a DataFrame, it's common to create new columns based on existing columns for analysis or prediction purposes. If the column name you specify does not already exist, a new column will be added to the DataFrame. If the column name already exists, the new values will overwrite the existing column.

Let's classify books based on their user rating

```
def classify_rating(rating):
    if rating >= 4.5:
        return 'Highly Rated'
    elif 3.0 <= rating < 4.5:
        return 'Moderately Rated'
    else:
        return 'Low Rated'

bestseller_df['Rating_Class'] = bestseller_df['User Rating'].apply(classify_rating)

bestseller_df.head()
```

	Name	Author	User Rating	Reviews
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350
1	11/22/63: A Novel	Stephen King	4.6	2052
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979
3	1984 (Signet Classics)	George Orwell	4.7	21424
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665

#### 7.6.5 Renaming a column

To rename a column in a pandas DataFrame, you can use the `rename()` method. . .

- Renaming a single column

```
df.rename(columns={'old_name': 'new_name'}, inplace=False)

# rename Year to Publication Year
bestseller_df.rename(columns={'Year':'Publication Year'}, inplace=True)
bestseller_df.head()
```

	Name	Author	User Rating	Reviews
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350
1	11/22/63: A Novel	Stephen King	4.6	2052
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979
3	1984 (Signet Classics)	George Orwell	4.7	21424
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665

- Renaming multiple columns

Specifying a dictionary that maps old column names to new one

```
df_renamed_multiple = df.rename(columns={'col1': 'new_col1', 'col2': 'new_col2'})
```

```
# rename Name to Book Name, and Author to Writer
bestseller_df.rename(columns={'Name':'Book Name', 'Author':'Writer'}, inplace=True)
bestseller_df.head()
```

	Book Name	Writer	User Rating	Reviews
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350
1	11/22/63: A Novel	Stephen King	4.6	2052
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979
3	1984 (Signet Classics)	George Orwell	4.7	21424
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665

### 7.6.6 Sorting data

Sorting dataset is a very common operation. The `sort_values()` function of Pandas can be used to sort a Pandas DataFrame or Series. Let us sort the spotify data in decreasing order of *track\_popularity*:

```
bestseller_sorted = bestseller_index_data.sort_values(by = 'User Rating', ascending = False)
bestseller_sorted.head()
```

	Unnamed: 0	Name	Author	User Rating	Reviews	Price
521	521	Unfreedom of the Press	Mark R. Levin	4.9	5956	11
420	420	The Legend of Zelda: Hyrule Historia	Patrick Thorpe	4.9	5396	20
248	248	Oh, the Places You'll Go!	Dr. Seuss	4.9	21834	8

	Unnamed: 0	Name	Author	User Rating	Reviews	Price
247	247	Oh, the Places You'll Go!	Dr. Seuss	4.9	21834	8
246	246	Oh, the Places You'll Go!	Dr. Seuss	4.9	21834	8

### 7.6.7 Ranking data

With the `rank()` function, we can rank the observations.

For example, let us add a new column to the bestseller data that provides the rank of the User Rating column:

```
bestseller_ranked = bestseller_df.copy()
bestseller_ranked['Rating_rank']=bestseller_ranked['User Rating'].rank()
bestseller_ranked.head()
```

	Book Name	Writer	User Rating	Reviews
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350
1	11/22/63: A Novel	Stephen King	4.6	2052
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979
3	1984 (Signet Classics)	George Orwell	4.7	21424
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665

Note the column `Rating_rank`. Why does it contain floating point numbers? Check the `rank()` documentation to find out!

## 7.7 Arithmetic operations within and between DataFrames

Pandas is built on top of NumPy, and for arithmetic operations, it inherits many of NumPy's powerful features. Pandas provides an intuitive and efficient way to perform arithmetic operations both within a DataFrame (on columns or rows) and between multiple DataFrames. These operations are element-wise, meaning they are applied to corresponding elements of the DataFrame or Series.

### 7.7.1 Arithmetic operations within a DataFrame

You can easily perform arithmetic operations on columns or rows within a DataFrame

- Arithmetic Between Columns

```
# Create a DataFrame
data = {'A': [1, 2, 3],
        'B': [4, 5, 6],
        'C': [7, 8, 9]}
df = pd.DataFrame(data)

# Adding two columns
df['A_plus_B'] = df['A'] + df['B']

# Multiplying two columns
df['A_times_C'] = df['A'] * df['C']

df
```

	A	B	C	A_plus_B	A_times_C
0	1	4	7	5	7
1	2	5	8	7	16
2	3	6	9	9	27

- Arithmetic on Entire Rows

```
# Summing values across columns for each row
df['row_sum'] = df.sum(axis=1)
df
```

	A	B	C	A_plus_B	A_times_C	row_sum
0	1	4	7	5	7	24
1	2	5	8	7	16	38
2	3	6	9	9	27	54

### 7.7.2 Arithmetic operations between DataFrames

When performing operations between two DataFrames, pandas aligns the data based on both the `index` and `column labels`. If the indices or columns do not match, pandas automatically fills the missing values with NaN (not-a-number) unless a fill value is provided.

Let us create two toy DataFrames:

```
#Creating two toy DataFrames
toy_df1 = pd.DataFrame([(1,2),(3,4),(5,6)], columns=['a','b'])
toy_df2 = pd.DataFrame([(100,200),(300,400),(500,600)], columns=['a','b'])
```

```
#DataFrame 1
toy_df1
```

	a	b
0	1	2
1	3	4
2	5	6

```
#DataFrame 2
toy_df2
```

	a	b
0	100	200
1	300	400
2	500	600

Element by element operations between two DataFrames can be performed with the operators `+`, `-`, `*`, `/`, `**`, and `%`. Below is an example of element-by-element addition of two DataFrames:

```
# Element-by-element arithmetic addition of the two DataFrames
toy_df1 + toy_df2
```

	a	b
0	101	202
1	303	404

	a	b
2	505	606

Note that these operations create problems when the row indices and/or column names of the two DataFrames do not match. See the example below:

```
#Creating another toy example of a DataFrame
toy_df3 = pd.DataFrame([(100,200),(300,400),(500,600)], columns=['a','b'], index=[1,2,3])
toy_df3
```

	a	b
1	100	200
2	300	400
3	500	600

```
#Adding DataFrames with some unmatching row indices
toy_df1 + toy_df3
```

	a	b
0	NaN	NaN
1	103.0	204.0
2	305.0	406.0
3	NaN	NaN

Note that the rows whose indices match between the two DataFrames are added up. The rest of the values are missing (or `NaN`) because only one of the DataFrames has that index.

As in the case of row indices, missing values will also appear in the case of unmatching column names, as shown in the example below.

```
toy_df4 = pd.DataFrame([(100,200),(300,400),(500,600)], columns=['b','c'])
toy_df4
```

	b	c
0	100	200
1	300	400

	b	c
2	500	600

```
#Adding DataFrames with some unmatching column names
toy_df1 + toy_df4
```

	a	b	c
0	NaN	102	NaN
1	NaN	304	NaN
2	NaN	506	NaN

### 7.7.3 Arithmetic Between DataFrame and Series (Broadcasting)

Pandas supports **broadcasting**, where a Series can be broadcasted to match the dimensions of a DataFrame during an arithmetic operation.

```
# Broadcasting: The row [1,2] (a Series) is added on every row in df2
toy_df1.loc[0,:] + toy_df2
```

	a	b
0	101	202
1	301	402
2	501	602

Note that the `+` operator is used to add values of a Series to a DataFrame based on column names. For adding a Series to a DataFrame based on row indices, we cannot use the `+` operator. Instead, we'll need to use the `add()` function as explained below.

**Broadcasting based on row/column labels:** We can use the `add()` function to broadcast a Series to a DataFrame. By default the Series adds based on column names, as in the case of the `+` operator.

```
# Add the first row of df1 (a Series) to every row in df2
toy_df2.add(toy_df1.loc[0,:])
```

	a	b
0	101	202
1	301	402
2	501	602

For broadcasting based on row indices, we use the `axis` argument of the `add()` function.

```
# The second column of df1 (a Series) is added to every col in df2
toy_df2.add(toy_df1.loc[:, 'b'], axis='index')
```

	a	b
0	102	202
1	304	404
2	506	606

#### 7.7.4 Converting a DataFrame to a NumPy Array

In pandas, you often need to convert a DataFrame or Series into a NumPy array to perform specific operations or to interface with libraries that require NumPy arrays.

There are two common methods for converting pandas objects to NumPy arrays: `.values` and `.to_numpy()`.

##### 7.7.4.1 Using `.values`

- The `.values` attribute returns the underlying NumPy array representation of a DataFrame or Series.
- It provides a direct conversion, but it has been deprecated in favor of the newer `.to_numpy()` method because `.values` does not always handle all DataFrame types consistently.

```
toy_df1
```

	a	b
0	1	2
1	3	4
2	5	6

a	b

```
toy_df1.values
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

#### 7.7.4.2 Using .to\_numpy()

- The `.to_numpy()` method is a recommended way to convert a DataFrame to a NumPy array.
- It handles different types of data, such as nullable types or mixed data types, more consistently than `.values`.
- Additional parameters: `.to_numpy()` can take parameters like `dtype` (to specify the data type) and `copy` (to create a copy or not).

```
toy_df1.to_numpy()
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In the NumPy chapter, we explored the below case study that demonstrated the benefits of using matrix multiplication in NumPy over python loops. However, this same operation cannot be directly performed in a pandas DataFrame in the same way.

```
food_qty = pd.read_csv('../data/food_quantity.csv', index_col=0)
price = pd.read_csv('../data/price.csv', index_col=0)
```

```
%%time
store_expense = pd.DataFrame(0.0, columns=price.columns, index = food_qty.index)
for person in store_expense.index:
    for store in store_expense.columns:
        for food in food_qty.columns:
            store_expense.loc[person, store] += food_qty.loc[person, food]*price.loc[food, s]

store_expense
```

```
CPU times: total: 0 ns
Wall time: 5.01 ms
```

Person	Target	Kroger
Ben	50.0	49.0
Barbara	58.5	61.0
Beth	43.5	43.5

```
%%time
# using matrix multiplication in numpy
import numpy as np
np.dot(food_qty.values, price.values)
```

```
CPU times: total: 0 ns
Wall time: 996 s
```

```
array([[50. , 49. ],
       [58.5, 61. ],
       [43.5, 43.5]])
```

```
# converting the result to a DataFrame
pd.DataFrame(np.dot(food_qty.values, price.values), columns=price.columns, index=food_qty.index)
```

Person	Target	Kroger
Ben	50.0	49.0
Barbara	58.5	61.0
Beth	43.5	43.5

## 7.8 Advanced DataFrame Operations

Pandas provides powerful methods for applying functions to DataFrame rows, columns, or individual elements. These methods allow for flexible data transformations, making them essential for advanced data manipulation.

### 7.8.1 apply(): Row-wise, or Column-wise

The `apply()` function in pandas is used to apply a function along an axis of a DataFrame.

- Row-wise: You can apply a function to each row using `axis=1`.
- Column-wise: You can apply a function to each column using `axis=0` (default behavior).

Let's create a DataFrame and apply a custom function to each row.

```
# Create a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})
```

```
df
```

	A	B	C
0	1	4	7
1	2	5	8
2	3	6	9

```
# Function to compute the row sum
def row_sum(row):
    return row.sum()
```

```
# Apply the function row-wise (axis=1)
df['Row_Sum'] = df.apply(row_sum, axis=1)
df
```

	A	B	C	Row_Sum
0	1	4	7	12
1	2	5	8	15
2	3	6	9	18

Let's apply the same function to each column next

```
# Apply the function column-wise (axis=0)
column_sum = df.apply(sum, axis=0)
column_sum
```

```
A      6
B     15
C    24
Row_Sum 45
dtype: int64
```

### 7.8.2 map(): Element-wise

`map()` is typically used for element-wise transformations on a Series.

```
# write the function to compute square of a number
def square(x):
    return x ** 2

# use the map() function to apply a function element-wise
df['A_squared'] = df['A'].map(square)
df
```

	A	B	C	Row_Sum	A_squared
0	1	4	7	12	1
1	2	5	8	15	4
2	3	6	9	18	9

Element-wise transformation on an entire DataFrame.

```
df.map(square)
```

	A	B	C	Row_Sum	A_squared
0	1	16	49	144	1
1	4	25	64	225	16
2	9	36	81	324	81

When applying a function to an entire DataFrame, it's important to ensure that all columns have the same data type. Otherwise, you might encounter errors or unexpected behavior.

```

# Create a DataFrame
df_str = pd.DataFrame({
    'A': [1, 2, 3],
    'B': ['I', 'major', 'in'],
    'C': [7, 8, 9]
})
# uncomment the following line to run the code
# df_str.map(square)

```

### 7.8.3 Key Differences Between `map()` and `apply()`

Feature	<code>map()</code>	<code>apply()</code>
Applies to Functionality	Both Series and DataFrame Element-wise transformations	Both Series and DataFrame Can apply more complex functions (row-wise or column-wise for DataFrame)
Input	Function, dictionary, or another Series	Function
Use Case	Used for simpler element-wise replacements or mappings	Used for more complex operations that can act on rows, columns, or individual elements
Axis Option	N/A	Can specify axis in DataFrame (row-wise or column-wise)

#### 7.8.3.1 Summary:

- `map()`: Primarily used for element-wise operations. It can take functions, dictionaries, or Series as input for substitution or transformation.
- `apply()`: Apply complex functions across rows or columns.

## 7.9 Introduction to Lambda Functions in Pandas

A lambda function is a small, anonymous function defined using the `lambda` keyword in Python. It can take any number of arguments but can only have one expression. In pandas, lambda functions are often used for quick and concise operations on data.

### 7.9.1 Syntax of Lambda Functions:

```
lambda arguments: expression
```

### 7.9.2 Key Features

- **Anonymous:** Lambda functions are defined without a name.
- **Concise:** They are typically used for small, simple functions that are not reused elsewhere.
- **Single Expression:** They can only contain one expression, which is evaluated and returned.

Lambda functions are commonly used with `apply()` and `map()` to efficiently transform data in pandas, allowing for quick, inline function definitions without the need to explicitly define separate functions.

Let's use a lambda function to rewrite the `apply()` and `map()` operations from the previous example

```
df['A_squared_lamdba'] = df['A'].map(lambda x: x ** 2)  
df
```

	A	B	C	Row_Sum	A_squared	A_squared_lamdba
0	1	4	7	12	1	1
1	2	5	8	15	4	4
2	3	6	9	18	9	9

```
df_squared = df.apply(lambda x: x ** 2)  
df_squared
```

	A	B	C	Row_Sum	A_squared	A_squared_lamdba
0	1	16	49	144	1	1
1	4	25	64	225	16	16
2	9	36	81	324	81	81

Let's look at other examples using lambda

Example 1: adding a new column based on existing columns

```

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

# Use a lambda function to create a new column 'C' as the sum of 'A' and 'B'
df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
print(df)

```

	A	B	C
0	1	4	5
1	2	5	7
2	3	6	9

Example 2: Filter rows based on a condition

```

# Filter rows where values in column 'A' are greater than 1
filtered_df = df[df['A'].apply(lambda x: x > 1)]
print(filtered_df)

```

	A	B	C
1	2	5	7
2	3	6	9

## 7.10 Understanding `inplace=True` and `inplace=False`

Above, we covered multiple methods that allow you to modify a DataFrame or Series. The `inplace` parameter of these methods determines whether the operation modifies the original object or returns a new object.

- `inplace=True`: Modifies the original DataFrame or Series directly and returns None. The changes are made in place, meaning the original object is altered.
- `inplace=False`: Returns a new DataFrame or Series with the modifications, leaving the original object unchanged. This is the default behavior for most methods.

**Note:** The default value for the `inplace` parameter is False for most methods.

Usage Example:

```

# Create a DataFrame
inplace_df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

# Drop a column without modifying the original DataFrame
df_dropped = inplace_df.drop('A', axis=1)
print(df_dropped) # New DataFrame without column 'A'
print(inplace_df) # Original DataFrame remains unchanged

# Drop a column in place
inplace_df.drop('A', axis=1, inplace=True)
print(inplace_df) # Original DataFrame is now modified

```

```

      B
0  4
1  5
2  6
      A   B
0  1  4
1  2  5
2  3  6
      B
0  4
1  5
2  6

```

## 7.11 Case study

To see the application of arithmetic operations on DataFrames, let us see the case study below.

**Song recommendation:** Spotify recommends songs based on songs listened by the user. Suppose you have listened to the song *drivers license*. Spotify intends to recommend you 5 songs that are *similar* to *drivers license*. Which songs should it recommend?

Let us see the available song information that can help us identify songs similar to *drivers license*. The [columns](#) attribute of DataFrame will display all the columns names. The description of some of the column names relating to audio features is [here](#).

```
spotify_data.columns
```

```
Index(['artist_followers', 'genres', 'artist_name', 'artist_popularity',
       'track_name', 'track_popularity', 'duration_ms', 'explicit',
       'release_year', 'danceability', 'energy', 'key', 'loudness', 'mode',
       'speechiness', 'acousticness', 'instrumentalness', 'liveness',
       'valence', 'tempo', 'time_signature'],
      dtype='object')
```

**Solution approach:** We have several features of a song. Let us find songs similar to *drivers license* in terms of *danceability*, *energy*, *key*, *loudness*, *mode*, *speechiness*, *acousticness*, *instrumentalness*, *liveness*, *valence*, *time\_signature* and *tempo*. Note that we are considering only audio features for simplicity.

To find the songs most similar to *drivers license*, we need to define a measure that quantifies the similarity. Let us define similarity of a song with *drivers license* as the Euclidean distance of the song from *drivers license*, where the coordinates of a song are: (*danceability*, *energy*, *key*, *loudness*, *mode*, *speechiness*, *acousticness*, *instrumentalness*, *liveness*, *valence*, *time\_signature*, *tempo*). Thus, similarity can be formulated as:

$$\text{Similarity}_{DL-S} = \sqrt{(danceability_{DL} - danceability_S)^2 + (energy_{DL} - energy_S)^2 + \dots + (tempo_{DL} - tempo_S)^2}$$

where the subscript *DL* stands for *drivers license* and *S* stands for any song. The top 5 songs with the least value of *Similarity*<sub>*DL-S*</sub> will be the most similar to *drivers lincense* and should be recommended.

Let us subset the columns that we need to use to compute the Euclidean distance.

```
audio_features = spotify_data[['danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness',
                               'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo', 'time_signature']]
```

```
audio_features.head()
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo	time_signature
0	0.673	0.529	0	-7.226	1	0.3060	0.0769	0.000338	0.0856	0.000000	140.0	1.0
1	0.511	0.566	6	-7.230	0	0.2000	0.3490	0.000000	0.3400	0.000000	140.0	1.0
2	0.699	0.687	7	-3.997	0	0.1060	0.3080	0.000036	0.1210	0.000000	140.0	1.0
3	0.708	0.690	2	-5.181	1	0.0442	0.3480	0.000000	0.2220	0.000000	140.0	1.0
4	0.753	0.597	8	-8.469	1	0.2920	0.0477	0.000000	0.1970	0.000000	140.0	1.0

```
danceability    energy    key    loudness    mode    speechiness    acousticness    instrumentalness    liveness    v
```

```
#Distribution of values of audio_features  
audio_features.describe()
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	v
count	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000
mean	0.568357	0.580633	5.240326	-9.432548	0.670928	0.111984	0.000000	0.000000	0.000000	0.000000
std	0.159444	0.236631	3.532546	4.449731	0.469877	0.198068	0.000000	0.000000	0.000000	0.000000
min	0.000000	0.000000	0.000000	-60.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.462000	0.405000	2.000000	-11.990000	0.000000	0.033200	0.000000	0.000000	0.000000	0.000000
50%	0.579000	0.591000	5.000000	-8.645000	1.000000	0.043100	0.000000	0.075300	0.000000	0.000000
75%	0.685000	0.776000	8.000000	-6.131000	1.000000	0.075300	0.000000	0.000000	0.000000	0.000000
max	0.988000	1.000000	11.000000	3.744000	1.000000	0.969000	0.000000	0.000000	0.000000	0.000000

Note that the audio features differ in terms of scale. Some features like *key* have a wide range of [0,11], while others like *danceability* have a very narrow range of [0,0.988]. If we use them directly, features like *danceability* will have a much higher influence on  $Similarity_{DL-S}$  as compared to features like *key*. Assuming we wish all the features to have equal weight in quantifying a song's similarity to *drivers license*, we should scale the features, so that their values are comparable.

Let us scale the value of each column to a standard uniform distribution:  $U[0, 1]$ .

For scaling the values of a column to  $U[0, 1]$ , we need to subtract the minimum value of the column from each value, and divide by the range of values of the column. For example, *danceability* can be standardized as follows:

```
#Scaling danceability to U[0,1]  
danceability_value_range = audio_features.danceability.max()-audio_features.danceability.min()  
danceability_std = (audio_features.danceability-audio_features.danceability.min())/danceability_value_range  
danceability_std
```

```
0          0.681174  
1          0.517206  
2          0.707490  
3          0.716599  
4          0.762146  
...  
243185    0.621457
```

```

243186    0.797571
243187    0.533401
243188    0.565789
243189    0.750000
Name: danceability, Length: 243190, dtype: float64

```

However, it will be cumbersome to repeat the above code for each audio feature. We can instead write a function that scales values of a column to  $U[0, 1]$ , and apply the function on all the audio features.

```

#Function to scale a column to U[0,1]
def scale_uniform(x):
    return (x-x.min())/(x.max()-x.min())

```

We will use the Pandas function `apply()` to apply the above function to the DataFrame `audio_features`.

```

#Scaling all audio features to U[0,1]
audio_features_scaled = audio_features.apply(scale_uniform)

```

The above two blocks of code can be concisely written with the `lambda` function as:

```

audio_features_scaled = audio_features.apply(lambda x: (x-x.min())/(x.max()-x.min()))

#All the audio features are scaled to U[0,1]
audio_features_scaled.describe()

```

	danceability	energy	key	loudness	mode	speechiness	acousticness
count	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000
mean	0.575260	0.580633	0.476393	0.793290	0.670928	0.115566	0.044479
std	0.161380	0.236631	0.321141	0.069806	0.469877	0.204405	0.099999
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.467611	0.405000	0.181818	0.753169	0.000000	0.034262	0.000000
50%	0.586032	0.591000	0.454545	0.805644	1.000000	0.044479	0.000000
75%	0.693320	0.776000	0.727273	0.845083	1.000000	0.077709	0.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Since we need to find the Euclidean distance from the song *drivers license*, let us find the index of the row containing features of *drivers license*.

```
#Index of the row consisting of drivers license can be found with the index attribute  
drivers_license_index = spotify_data[spotify_data.track_name=='drivers license'].index[0]
```

Note that the object returned by the `index` attribute is of type `pandas.core.indexes.numeric.Int64Index`. The elements of this object can be retrieved like the elements of a python list. That is why the object is sliced with `[0]` to return the first element of the object. As there is only one observation with the `track_name` as *drivers license*, we sliced the first element. If there were multiple observations with `track_name` as *drivers license*, we will obtain the indices of all those observations with the `index` attribute.

```
audio_features_scaled.loc[drivers_license_index,:]
```

```
danceability      0.592105  
energy            0.436000  
key               0.909091  
loudness          0.803825  
mode              1.000000  
speechiness       0.062023  
acousticness      0.723896  
instrumentalness  0.000013  
liveness          0.105000  
valence           0.132000  
tempo              0.590841  
time_signature    0.800000  
Name: 2398, dtype: float64
```

Now, we'll subtract the audio features of *drivers license* from all other songs (broadcasting):

```
#Audio features of drivers license are being subtracted from audio features of all songs by 1  
songs_minus_DL = audio_features_scaled-audio_features_scaled.loc[drivers_license_index,:]
```

Now, let us square the difference computed above. We'll use the in-built python function `pow()` to square the difference:

```
songs_minus_DL_sq = songs_minus_DL.pow(2)  
songs_minus_DL_sq.head()
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	live
0	0.007933	0.008649	0.826446	0.000580	0.0	0.064398	0.418204	1.055600e-07	0.00
1	0.005610	0.016900	0.132231	0.000577	1.0	0.020844	0.139498	1.716100e-10	0.05
2	0.013314	0.063001	0.074380	0.005586	1.0	0.002244	0.171942	5.382400e-10	0.00
3	0.015499	0.064516	0.528926	0.003154	0.0	0.000269	0.140249	1.716100e-10	0.01
4	0.028914	0.025921	0.033058	0.000021	0.0	0.057274	0.456981	1.716100e-10	0.00

Now, we'll sum the squares of differences from all audio features to compute the similarity of all songs to *drivers license*.

```
distance_squared = songs_minus_DL_sq.sum(axis = 1)
distance_squared.head()
```

```
0    1.337163
1    1.438935
2    1.516317
3    1.004043
4    0.920316
dtype: float64
```

Now, we'll sort these distances to find the top 5 songs closest to drivers's license.

```
distances_sorted = distance_squared.sort_values()
distances_sorted.head()
```

```
2398    0.000000
81844   0.008633
4397    0.011160
130789   0.015018
143744   0.015058
dtype: float64
```

Using the indices of the top 5 distances, we will identify the top 5 songs most similar to *drivers license*:

```
spotify_data.loc[distances_sorted.index[0:6], :]
```

	artist_followers	genres	artist_name	artist_popularity	track_name
2398	1444702	pop	Olivia Rodrigo	88	drivers license
81844	2264501	pop	Jay Chou	74	
4397	25457	pop	Terence Lam	60	in Bb major
130789	176266	pop	Alan Tam	54	
143744	396326	pop & rock	Laura Branigan	64	How Am I Supposed to Live W
35627	1600562	pop	Tiziano Ferro	68	Non Me Lo So Spiegare

We can see the top 5 songs most similar to *drivers license* in the *track\_name* column above. Interestingly, three of the five songs are Asian! These songs indeed sound similar to *drivers license!*

## 7.12 Independent Practice:

### 7.12.1 Practice exercise 1

#### 7.12.1.1

Read the file *Top 10 Albums By Year.csv*. This file contains the top 10 albums for each year from 1990 to 2021. Each row corresponds to a unique album.

#### 7.12.1.2

Print the summary statistics of the data, and answer the following questions:

1. What proportion of albums have 15 or lesser tracks? Mention a range for the proportion.
2. What is the mean length of a track (in minutes)?

#### 7.12.1.3

Why is *Worldwide Sales* not included in the summary statistics table printed in the above step?

#### **7.12.1.4**

Update the DataFrame so that `Worldwide Sales` is included in the summary statistics table.  
Print the summary statistics table.

**Hint:** Sometimes it may not be possible to convert an object to `numeric()`. For example, the object ‘hi’ cannot be converted to a `numeric()` by the python compiler. To avoid getting an error, use the `errors` argument of `to_numeric()` to force such conversions to `NaN` (missing value).

#### **7.12.1.5**

Create a new column called `mean_sales_per_year` that computes the average worldwide sales per year for each album, assuming that the worldwide sales are as of 2022. Print the first 5 rows of the updated DataFrame.

#### **7.12.1.6**

Find the album having the highest worldwide sales per year, and its artist.

#### **7.12.1.7**

Subset the data to include only Hip-Hop albums. How many Hip\_Hop albums are there?

#### **7.12.1.8**

Which album amongst hip-hop has the higest mean sales per year per track, and who is its artist?

### **7.12.2 Practice exercise 2**

#### **7.12.2.1**

Read the file *STAT303-1 survey for data analysis.csv*.

#### **7.12.2.2**

How many observations and variables are there in the data?

### 7.12.2.3

Rename all the columns of the data, except the first two columns, with the shorter names in the list `new_col_names` given below. The order of column names in the list is the same as the order in which the columns are to be renamed starting with the third column from the left.

### 7.12.2.4

Rename the following columns again:

1. Rename `do_you_smoke` to `smoke`.
2. Rename `are_you_an_introvert_or_extrovert` to `introvert_extrovert`.

**Hint:** Use the function [rename\(\)](#)

### 7.12.2.5

Find the proportion of people going to more than 4 parties per month. Use the variable `parties_per_month`.

### 7.12.2.6

Among the people who go to more than 4 parties a month, what proportion of them are introverts?

### 7.12.2.7

Find the proportion of people in each category of the variable `how_happy`.

### 7.12.2.8

Among the people who go to more than 4 parties a month, what proportion of them are either `Pretty happy` or `Very happy`?

### 7.12.2.9

Examine the column `num_insta_followers`. Some numbers in the column contain a comma(,) or a tilde(~). Remove both these characters from the numbers in the column.

**Hint:** You may use the function [str.replace\(\)](#) of the Pandas Series class.

### **7.12.2.10**

Convert the column `num_insta_followers` to numeric. Coerce the errors.

### **7.12.2.11**

What is the mean `internet_hours_per_day` for the top 46 people in terms of number of instagram followers?

### **7.12.2.12**

What is the mean `internet_hours_per_day` for the remaining people?

## **7.12.3 Practice exercise 3**

### **7.12.3.1**

Use the updated dataset from Practice exercise 2.

The last four variables in the dataset are:

1. `cant_change_math_ability`
2. `can_change_math_ability`
3. `math_is_genetic`
4. `much_effort_is_lack_of_talent`

Each of the above variables has values - `Agree` / `Disagree`. Replace `Agree` with 1 and `Disagree` with 0.

**Hint :** You can do it with any one of the following methods:

1. Use the `map()` function
2. Use the `apply()` function with the `lambda` function
3. Use the `replace()` function

Two of the above methods avoid a `for`-loop. Which ones?

# 8 Reading Data

```
<IPython.core.display.Image object>
```

## 8.1 Types of data - structured and unstructured

Reading data is the first step to extract information from it. Data can exist broadly in two formats:

- (1) Structured data, and
- (2) Unstructured data.

Structured data is typically stored in a tabular form, where rows in the data correspond to “observations” and columns correspond to “variables”. For example, the following dataset contains 5 observations, where each observation (or row) consists of information about a movie. The variables (or columns) contain different pieces of information about a given movie. As all variables for a given row are related to the same movie, the data below is also called relational data.

	Title	US Gross	Production Budget	Release Date	Major Genre
0	The Shawshank Redemption	28241469	25000000	Sep 23 1994	Drama
1	Inception	285630280	160000000	Jul 16 2010	Horror/Thriller
2	One Flew Over the Cuckoo's Nest	108981275	4400000	Nov 19 1975	Comedy
3	The Dark Knight	533345358	185000000	Jul 18 2008	Action/Adventure
4	Schindler's List	96067179	25000000	Dec 15 1993	Drama

Unstructured data is data that is not organized in any pre-defined manner. Examples of unstructured data can be text files, audio/video files, images, Internet of Things (IoT) data, etc. Unstructured data is relatively harder to analyze as most of the analytical methods and tools are oriented towards structured data. However, an unstructured data can be used to obtain structured data, which in turn can be analyzed. For example, an image can be converted to an array of pixels - which will be structured data. Machine learning algorithms can then be used on the array to classify the image as that of a dog or a cat.

In this course, we will focus on analyzing structured data.

[Pandas](#) is a popular Python library used for working in tabular data (similar to the data stored in a spreadsheet). Pandas provides helper functions to read data from various file formats like CSV, Excel spreadsheets, HTML tables, JSON, SQL, and more.

## 8.2 Reading a *csv* file with *Pandas*

Structured data can be stored in a variety of formats. The most popular format is *data\_file\_name.csv*, where the extension *csv* stands for comma separated values. The variable values of each observation are separated by a comma in a *.csv* file. In other words, the **delimiter** is a comma in a *csv* file. However, the comma is not visible when a *.csv* file is opened with Microsoft Excel.

The below csv file contains day-wise Covid-19 data for Italy:

```
date,new_cases,new_deaths,new_tests
2020-04-21,2256.0,454.0,28095.0
2020-04-22,2729.0,534.0,44248.0
2020-04-23,3370.0,437.0,37083.0
2020-04-24,2646.0,464.0,95273.0
2020-04-25,3021.0,420.0,38676.0
2020-04-26,2357.0,415.0,24113.0
2020-04-27,2324.0,260.0,26678.0
2020-04-28,1739.0,333.0,37554.0
...
...
```

**CSVs:** A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. (Wikipedia)

First, let's import the Pandas library. As a convention, it is imported with the alias pd.

```
import pandas as pd
import os
```

### 8.2.1 Using the `read_csv` function

The `pd.read_csv` function can be used to read a CSV file into a pandas `DataFrame`: a spreadsheet-like object for analyzing and processing data.

```
movie_ratings = pd.read_csv('./datasets/movie_ratings.csv')
```

The built-in python function `type` can be used to check the datatype of an object:

```
type(movie_ratings)
```

```
pandas.core.frame.DataFrame
```

We'll learn more about `DataFrame` in a future lesson.

Note that I use the `relative path` to specify the file path for `movie_ratings.csv`, you may need to change it based on where you store the data file.

### 8.2.2 Data Overview

Once the data has been read, we may want to see what the data looks like. We'll use another Pandas function `head()` to view the first few rows of the data.

```
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
1	Major Dundee	14873	14873	3800000	Apr 07 1965	PG/PG-13
2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	PG/PG-13

#### Row Indices and column names (axis labels)

By default, when you create a pandas DataFrame (or Series) without specifying an index, pandas will automatically assign integer-based row indices starting from 0. These indices serve as the row labels and uniquely identify each row in the DataFrame. For example, the index 2 corresponds to the row of the movie The Informers. By default, the indices are integers starting from 0. However, they can be changed (to even non-integer values) if desired by the user.

The bold text on top of the DataFrame refers to column names. For example, the column **US Gross** consists of the gross revenue of a movie in the US.

Collectively, the indices and column names are referred as **axis labels**.

**Basic information** We can view some basic information about the data frame using the `.info` method.

```
movie_ratings.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2228 entries, 0 to 2227
Data columns (total 11 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   Title             2228 non-null    object  
 1   US Gross          2228 non-null    int64  
 2   Worldwide Gross   2228 non-null    int64  
 3   Production Budget 2228 non-null    int64  
 4   Release Date      2228 non-null    object  
 5   MPAA Rating       2228 non-null    object  
 6   Source            2228 non-null    object  
 7   Major Genre       2228 non-null    object  
 8   Creative Type     2228 non-null    object  
 9   IMDB Rating       2228 non-null    float64 
 10  IMDB Votes        2228 non-null    int64  
dtypes: float64(1), int64(4), object(6)
memory usage: 191.6+ KB
```

The `shape` property of a pandas DataFrame provides a tuple that represents the dimensions of the DataFrame:

- The first value in the tuple is the number of rows.
- The second value in the tuple is the number of columns.

```
movie_ratings.shape
```

```
(2228, 11)
```

The `columns` property contains the list of columns within the data frame.

```
movie_ratings.columns
```

```
Index(['Title', 'US Gross', 'Worldwide Gross', 'Production Budget',
       'Release Date', 'MPAA Rating', 'Source', 'Major Genre', 'Creative Type',
       'IMDB Rating', 'IMDB Votes'],
      dtype='object')
```

You can view statistical information for numerical columns (mean, standard deviation, minimum/maximum values, and the number of non-empty values) using the `.describe` method.

```
movie_ratings.describe()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes
count	2.228000e+03	2.228000e+03	2.228000e+03	2228.000000	2228.000000
mean	5.076370e+07	1.019370e+08	3.816055e+07	6.239004	33585.154847
std	6.643081e+07	1.648589e+08	3.782604e+07	1.243285	47325.651561
min	0.000000e+00	8.840000e+02	2.180000e+02	1.400000	18.000000
25%	9.646188e+06	1.320737e+07	1.200000e+07	5.500000	6659.250000
50%	2.838649e+07	4.266892e+07	2.600000e+07	6.400000	18169.000000
75%	6.453140e+07	1.200000e+08	5.300000e+07	7.100000	40092.750000
max	7.601676e+08	2.767891e+09	3.000000e+08	9.200000	519541.000000

Functions & Methods we've looked so far

- `pd.read_csv` - Read data from a CSV file into a Pandas DataFrame object
- `.info()` - View basic infomation about rows, columns & data types
- `.shape` - Get the number of rows & columns as a tuple
- `.columns` - Get the list of column names
- `.describe()` - View statistical information about numeric columns

## 8.3 Data Selection and Filtering

### 8.3.1 Extracting Column(s)

The first step when working with a DataFrame is often to extract one or more columns. To do this effectively, it's helpful to understand the internal structure of a DataFrame. Conceptually, you can think of a DataFrame as a dictionary of lists, where the keys are column names, and the values are lists or arrays containing data for the respective columns.

```
# Pandas format is similiar to this
movie_ratings_dict = {
    'Title': ['Opal Dreams', 'Major Dundee', 'The Informers', 'Buffalo Soldiers', 'The Last
    'US Gross': [14443, 14873, 315000, 353743, 388390],
    'Worldwide Gross': [14443, 14873, 315000, 353743, 388390],
    'Production Budget': [9000000, 3800000, 18000000, 15000000, 2200000]
}
```

For dictionary, we use key to retrieve its values

```
movie_ratings_dict['Title']
```

```
['Opal Dreams',  
 'Major Dundee',  
 'The Informers',  
 'Buffalo Soldiers',  
 'The Last Sin Eater']
```

Similar like dictionary, we can extract a column by its column name

```
movie_ratings['Title']
```

```
0           Opal Dreams  
1           Major Dundee  
2           The Informers  
3           Buffalo Soldiers  
4           The Last Sin Eater  
...  
2223          King Arthur  
2224          Mulan  
2225          Robin Hood  
2226  Robin Hood: Prince of Thieves  
2227          Spiceworld  
Name: Title, Length: 2228, dtype: object
```

Each column is a feature of the dataframe, we can also use . operator to extract a single column

```
movie_ratings.Title
```

```
0           Opal Dreams  
1           Major Dundee  
2           The Informers  
3           Buffalo Soldiers  
4           The Last Sin Eater  
...  
2223          King Arthur  
2224          Mulan
```

```

2225          Robin Hood
2226  Robin Hood: Prince of Thieves
2227        Spiceworld
Name: Title, Length: 2228, dtype: object

```

When extracting multiple columns, you need to place the column names inside a list.

```
movie_ratings[['Title', 'US Gross', 'Worldwide Gross']]
```

	Title	US Gross	Worldwide Gross
0	Opal Dreams	14443	14443
1	Major Dundee	14873	14873
2	The Informers	315000	315000
3	Buffalo Soldiers	353743	353743
4	The Last Sin Eater	388390	388390
...	...	...	...
2223	King Arthur	51877963	203877963
2224	Mulan	120620254	303500000
2225	Robin Hood	105269730	310885538
2226	Robin Hood: Prince of Thieves	165493908	390500000
2227	Spiceworld	29342592	56042592

### 8.3.2 Extracting Row(s)

#### 8.3.2.1 Extracting based on a Single Condition or Multiple Conditions

In many cases, we need to filter rows based on specific conditions or a combination of multiple conditions. Next, let's explore how to use these conditions effectively to extract rows that meet our criteria, whether it's a single condition or multiple conditions combined

```
# extracting the rows that have IMDB Rating greater than 8
movie_ratings[movie_ratings['IMDB Rating'] > 8]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Ra
21	Gandhi, My Father	240425	1375194	5000000	Aug 03 2007	Other
56	Ed Wood	5828466	5828466	18000000	Sep 30 1994	R
67	Requiem for a Dream	3635482	7390108	4500000	Oct 06 2000	Other
164	Trainspotting	16501785	24000785	3100000	Jul 19 1996	R

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
181	The Wizard of Oz	28202232	28202232	2777000	Aug 25 2039	G
...	...	...	...	...	...	...
2090	Finding Nemo	339714978	867894287	94000000	May 30 2003	G
2092	Toy Story 3	410640665	1046340665	200000000	Jun 18 2010	G
2094	Avatar	760167650	2767891499	237000000	Dec 18 2009	PG/PG-13
2130	Scarface	44942821	44942821	25000000	Dec 09 1983	Other
2194	The Departed	133311000	290539042	90000000	Oct 06 2006	R

To combine multiple conditions in pandas, you need to use the `&` (AND) and `|` (OR) operators. Make sure to enclose each condition in parentheses () for clarity and to ensure proper evaluation order.

```
# extracting the rows that have IMDB Rating greater than 8 and US Gross less than 1000000
movie_ratings[(movie_ratings['IMDB Rating'] > 8) & (movie_ratings['US Gross'] < 1000000)]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
21	Gandhi, My Father	240425	1375194	5000000	Aug 03 2007	Other
636	Lake of Fire	25317	25317	6000000	Oct 03 2007	Other

### 8.3.2.2 Negating Conditions with the ~ Operator

The tilde (~) is used for negating boolean conditions in pandas, making it a useful tool for excluding specific values or rows.

In some cases, we may want to extract rows that do not meet a specific condition. To accomplish this, we can use the `~` operator, which negates a condition. This operator allows us to filter data by excluding rows that satisfy a particular condition, making it a powerful tool for refining our queries. Let's explore how to use the `~` operator to negate conditions and select rows that do not meet our criteria

```
# Excluding the rows that have IMDB Rating that equals 8 using the tilde ~
movie_ratings[~(movie_ratings['IMDB Rating'] == 8)]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	I
1	Major Dundee	14873	14873	3800000	Apr 07 1965	I
2	The Informers	315000	315000	18000000	Apr 24 2009	PG

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	I
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	I
...	...	...	...	...	...	...
2223	King Arthur	51877963	203877963	90000000	Jul 07 2004	I
2224	Mulan	120620254	303500000	90000000	Jun 19 1998	C
2225	Robin Hood	105269730	310885538	210000000	May 14 2010	I
2226	Robin Hood: Prince of Thieves	165493908	390500000	50000000	Jun 14 1991	I
2227	Spiceworld	29342592	56042592	25000000	Jan 23 1998	I

Another way to exclude rows where a column equals a certain value, you can use `!=` to create the condition.

```
# using the != operator
movie_ratings[movie_ratings['IMDB Rating'] != 8]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	I
1	Major Dundee	14873	14873	3800000	Apr 07 1965	I
2	The Informers	315000	315000	18000000	Apr 24 2009	I
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	I
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	I
...	...	...	...	...	...	...
2223	King Arthur	51877963	203877963	90000000	Jul 07 2004	I
2224	Mulan	120620254	303500000	90000000	Jun 19 1998	C
2225	Robin Hood	105269730	310885538	210000000	May 14 2010	I
2226	Robin Hood: Prince of Thieves	165493908	390500000	50000000	Jun 14 1991	I
2227	Spiceworld	29342592	56042592	25000000	Jan 23 1998	I

### 8.3.3 Extracting Subsets of Rows and Columns

Sometimes we may be interested in working with a subset of rows and columns of the data, instead of working with the entire dataset. The indexing operators `loc` and `iloc` provide a convenient way of selecting a subset of desired rows and columns.

Let us first sort the `movie_ratings` data frame by `IMDB Rating`.

```
movie_ratings_sorted = movie_ratings.sort_values(by = 'IMDB Rating', ascending = False)
movie_ratings_sorted.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
182	The Shawshank Redemption	28241469	28241469	25000000	Sep 23 1994	R
2084	Inception	285630280	753830280	160000000	Jul 16 2010	PG
790	Schindler's List	96067179	321200000	25000000	Dec 15 1993	R
1962	Pulp Fiction	107928762	212928762	8000000	Oct 14 1994	R
561	The Dark Knight	533345358	1022345358	185000000	Jul 18 2008	PG

### 8.3.3.1 Subsetting the DataFrame by loc

The operator `loc` uses axis labels (row indices and column names) to subset the data.

Let's subset the `title`, `worldwide gross`, `production budget`, and `IMDB rating` of top 3 movies.

```
# Subsetting the DataFrame by loc - using axis labels
movies_subset = movie_ratings_sorted.loc[[182, 2084, 2092], [ 'Title', 'IMDB Rating', 'US Gross']]
movies_subset
```

	Title	IMDB Rating	US Gross	Worldwide Gross	Production Budget
182	The Shawshank Redemption	9.2	28241469	28241469	25000000
2084	Inception	9.1	285630280	753830280	160000000
2092	Toy Story 3	8.9	410640665	1046340665	200000000

The `:` symbol in `.loc` is a slicing operator that represents a range or all elements in the specified dimension (rows or columns). Use `:` alone to select all rows/columns, or with start/end points to slice specific parts of the DataFrame.

```
# Subsetting the DataFrame by loc - using axis labels. the colon is used to select all rows
movies_subset = movie_ratings_sorted.loc[:, ['Title', 'Worldwide Gross', 'Production Budget', 'IMDB Rating']]
movies_subset
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
790	Schindler's List	321200000	25000000	8.9
1962	Pulp Fiction	212928762	8000000	8.9
561	The Dark Knight	1022345358	185000000	8.9
...	...	...	...	...

	Title	Worldwide Gross	Production Budget	IMDB Rating
1051	Glitter	4273372	8500000	2.0
1495	Disaster Movie	34690901	20000000	1.7
1116	Crossover	7009668	5600000	1.7
805	From Justin to Kelly	4922166	12000000	1.6
1147	Super Babies: Baby Geniuses 2	9109322	20000000	1.4

```
# Subsetting the DataFrame by loc - using axis labels. the colon is used to select a range of rows
movies_subset = movie_ratings_sorted.loc[182:561,['Title','Worldwide Gross','Production Budget']]
movies_subset
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
790	Schindler's List	321200000	25000000	8.9
1962	Pulp Fiction	212928762	8000000	8.9
561	The Dark Knight	1022345358	185000000	8.9

Combining `.loc` with condition(s) to extract specific rows and columns based on criteria

```
# extracting the rows that have IMDB Rating greater than 8 or US Gross less than 1000000, only
movie_ratings[(movie_ratings['IMDB Rating'] > 8) & (movie_ratings['US Gross'] < 1000000)][['Title','IMDB Rating','US Gross']]

#using loc to extract the rows that have IMDB Rating greater than 8 or US Gross less than 1000000
movie_ratings.loc[(movie_ratings['IMDB Rating'] > 8) & (movie_ratings['US Gross'] < 1000000)]
```

	Title	IMDB Rating
21	Gandhi, My Father	8.1
636	Lake of Fire	8.4

### 8.3.3.2 Subsetting the DataFrame by iloc

while `iloc` uses the position of rows or columns, where position has values 0,1,2,3,...and so on, for rows from top to bottom and columns from left to right. In other words, the first row has position 0, the second row has position 1, the third row has position 2, and so on. Similarly, the first column from left has position 0, the second column from left has position 1, the third column from left has position 2, and so on.

```
<IPython.core.display.Image object>
```

```
# let's check the movie_ratings_sorted DataFrame
movie_ratings_sorted.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA R
182	The Shawshank Redemption	28241469	28241469	25000000	Sep 23 1994	R
2084	Inception	285630280	753830280	160000000	Jul 16 2010	PG
790	Schindler's List	96067179	321200000	25000000	Dec 15 1993	R
1962	Pulp Fiction	107928762	212928762	8000000	Oct 14 1994	R
561	The Dark Knight	533345358	1022345358	185000000	Jul 18 2008	PG

After sorting, the position-based index changes, while the label-based index remains unchanged. Let's pass the position-based index to `iloc` to retrieve the top 2 rows from the `movie_ratings_sorted` DataFrame.

```
movie_ratings_sorted.iloc[0:2,:]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA R
182	The Shawshank Redemption	28241469	28241469	25000000	Sep 23 1994	R
2084	Inception	285630280	753830280	160000000	Jul 16 2010	PG

It is important to note that the endpoint is excluded in an `iloc` slice.

For comparison, let's pass the same argument to `loc` and see what it returns.

```
movie_ratings_sorted.loc[0:2,:]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA R
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
851	Star Trek: Generations	75671262	120000000	38000000	Nov 18 1994	PG/PG-13
140	Tuck Everlasting	19161999	19344615	15000000	Oct 11 2002	PG/PG-13
708	De-Lovely	13337299	18396382	4000000	Jun 25 2004	PG/PG-13
705	Flyboys	13090630	14816379	60000000	Sep 22 2006	PG/PG-13
...	...	...	...	...	...	...
955	The Brothers Solomon	900926	900926	10000000	Sep 07 2007	R
1637	Drumline	56398162	56398162	20000000	Dec 13 2002	PG/PG-13

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
1610	Hollywood Homicide	30207785	51107785	75000000	Jun 13 2003	PG/PG-13
569	Doom	28212337	54612337	70000000	Oct 21 2005	R
2	The Informers	315000	315000	18000000	Apr 24 2009	R

As you can see, `:` is used in the same way as in `loc` to denote a range of the index. All rows between label indices 0 and 2, inclusive of both ends, are returned.

```
# Subsetting the DataFrame by iloc - using index of the position of rows and columns
movies_iloc_subset1 = movie_ratings_sorted.iloc[0:10,[0,2,3,9]]
movies_iloc_subset1
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
2092	Toy Story 3	1046340665	200000000	8.9
1962	Pulp Fiction	212928762	8000000	8.9
790	Schindler's List	321200000	25000000	8.9
561	The Dark Knight	1022345358	185000000	8.9
184	Cidade de Deus	28763397	3300000	8.8
487	The Lord of the Rings: The Fellowship of the Ring	868621686	109000000	8.8
497	The Lord of the Rings: The Return of the King	1133027325	94000000	8.8
1081	C'era una volta il West	5321508	5000000	8.8

Can you use `.iloc` for conditional filtering, why or why not?

To recap, here are the **Key differences between loc and iloc in pandas:**

- **Indexing Type:**

- `loc` uses labels (names) for indexing.
- `iloc` uses integer positions for indexing.

- **Inclusion of Endpoints:**

- In a `loc` slice, both endpoints are included.
- In an `iloc` slice, the endpoint is excluded.

### 8.3.4 Finding minimum/maximum of a column

When working with pandas, there are two main options for locating the minimum or maximum values in a DataFrame column:

- `idxmin()` and `idxmax()`: return the **index label** of the first occurrence of the maximum or minimum value in a specified column.

```
# movie_ratings_sorted.iloc[position_max_wgross,:]  
max_index = movie_ratings_sorted['Worldwide Gross'].idxmax()  
min_index = movie_ratings_sorted['Worldwide Gross'].idxmin()  
print("Max index: ", max_index)  
print("Min index: ", min_index)
```

```
Max index: 2094  
Min index: 896
```

`idxmin()` and `idxmax()` return the index label of the minimum or maximum value in a column. You can use these returned index labels with `.loc` to extract the corresponding row.

```
print(movie_ratings_sorted.loc[max_index, 'Worldwide Gross'])  
print(movie_ratings_sorted.loc[min_index, 'Worldwide Gross'])
```

```
2767891499  
884
```

- `argmax()` and `argmin()`: Return the **integer position** of the first occurrence of the maximum or minimum value in a column. You can use these integer positions with `.iloc` to extract the corresponding row

```
# using argmax and argmin, which return the index of the maximum and minimum values  
max_position = movie_ratings_sorted['Worldwide Gross'].argmax()  
min_position = movie_ratings_sorted['Worldwide Gross'].argmin()  
print("max position:", max_position)  
print("min position:", min_position)  
  
# using iloc to get the row with the maximum and minimum values  
print(movie_ratings_sorted.iloc[max_position, 2])  
print(movie_ratings_sorted.iloc[min_position, 2])
```

```
max position: 48
min position: 2149
2767891499
884
```

#### Tips:

- When working with non-unique or custom indices, it is recommended to use `idxmax()` and `idxmin()` to retrieve index labels, as `argmax()` may be less intuitive in such scenarios.
- For DataFrames, use `.idxmax(axis=1)` or `.idxmin(axis=1)` to find the index labels corresponding to the maximum or minimum values across rows, rather than columns.

#### 8.3.5 Finding the top $n$ minimum/maximum values of a column

To find the top  $n$  minimum or maximum values of a column in a pandas DataFrame, you can use the `nsmallest()` and `nlargest()` methods. Here's how you can do it:

```
# find the top 3 movies with the highest worldwide gross
top_3_movies = movie_ratings.nlargest(3, 'Worldwide Gross')
top_3_movies
```

	Title	US Gross	Worldwide Gross	Production Budget
2094	Avatar	760167650	2767891499	237000000
2093	Titanic	600788188	1842879955	200000000
497	The Lord of the Rings: The Return of the King	377027325	1133027325	94000000

Let's double check the result using the `movie_ratings_sorted` dataframe

```
movie_ratings_sorted['Worldwide Gross'].nlargest(3)
```

```
2094    2767891499
2093    1842879955
497     1133027325
Name: Worldwide Gross, dtype: int64
```

Let's find the 3 movies with the smallest IMDb votes using the `nsmallest` method

```
bottom_3_movies = movie_ratings.nsmallest(3, 'IMDB Votes')
bottom_3_movies
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating	Source
1000	Teeth	347578	2300349	2000000	Jan 18 2008	R	Original S
1261	Birth	5005899	14603001	20000000	Oct 29 2004	R	Original S
1298	CachÈ	3647381	17147381	8000000	Dec 23 2005	R	Original S

Let's check the result using `sort_values` method

```
# using the sort_values method
movie_ratings_sorted['IMDB Votes'].nsmallest(3)
```

```
1000    18
1261    25
1298    26
Name: IMDB Votes, dtype: int64
```

## 8.4 Writing data to a .csv file

The Pandas function `to_csv` can be used to write (or export) data to a csv. Below is an example.

```
#Exporting the data of the top 250 movies to a csv file
movie_ratings.to_csv('../data/movie_rating_exported.csv')

# check if the file has been exported
os.listdir('../data')

['bestseller_books.txt',
 'country-capital-lat-long-population.csv',
 'covid.csv',
 'fifa_data.csv',
 'food_quantity.csv',
 'gas_prices.csv',
 'gdp_lifeExpectancy.csv',
 'LOTR 2.csv',
```

```
'LOTR.csv',
'movies.csv',
'movies_cleaned.csv',
'movie_ratings.csv',
'movie_ratings.txt',
'movie_rating_exported.csv',
'party_nyc.csv',
'price.csv',
'question_json_data.json',
'spotify_data.csv',
'stocks.csv']
```

## 8.5 Reading other data formats - txt, html, json

Although .csv is a very popular format for structured data, data is found in several other formats as well. Some of the other data formats are .txt, .html and .json.

### 8.5.1 Reading .txt files

The *txt* format offers some additional flexibility as compared to the *csv* format. In the *csv* format, the delimiter is a comma (or the column values are separated by a comma). However, in a *txt* file, the delimiter can be anything as desired by the user. Let us read the file *movie\_ratings.txt*, where the variable values are separated by a tab character.

```
movie_ratings_txt = pd.read_csv('../data/movie_ratings.txt', sep='\t')
movie_ratings_txt.head()
```

	Unnamed: 0	Title	US Gross	Worldwide Gross	Production Budget	Release Date	Metascore
0	0	Opal Dreams	14443	14443	9000000	Nov 22 2006	P
1	1	Major Dundee	14873	14873	3800000	Apr 07 1965	P
2	2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	P

We use the function *read\_csv* to read a *txt* file. However, we mention the tab character (`r"\t"`) as a separator of variable values.

Note that there is no need to remember the argument name - *sep* for specifying the delimiter. You can always refer to the [read\\_csv\(\)](#) documentation to find the relevant argument.

### 8.5.2 Reading HTML data

The *Pandas* function `read_html` searches for tabular data, i.e., data contained within the `<table>` tags of an html file. Let us read the tables in the GDP per capita [page](#) on Wikipedia.

```
#Reading all the tables from the Wikipedia page on GDP per capita
tables = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_...
```

All the tables will be read and stored in the variable named as `tables`. Let us find the datatype of the variable `tables`.

```
#Finidng datatype of the variable - tables
type(tables)
```

```
list
```

The variable - `tables` is a list of all the tables read from the HTML data.

```
#Number of tables read from the page
len(tables)
```

```
6
```

The in-built function `len` can be used to find the length of the list - `tables` or the number of tables read from the Wikipedia page. Let us check out the first table.

```
#Checking out the first table. Note that the index of the first table will be 0.
tables[0]
```

0	1	2
0 >\$60,000 \$50,000–\$60,000 \$40,000–\$50,000 \$30,0...	\$20,000–\$30,000 \$10,000–\$20,000 \$5,000–\$10,000...	

The above table doesn't seem to be useful. Let us check out the second table.

```
#Checking out the second table. Note that the index of the first table will be 1.
tables[1]
```

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
0	Monaco	—	—	240862	2022	234317	2021
1	Liechtenstein	—	—	187267	2022	169260	2021
2	Luxembourg	131384	2024	128259	2023	133745	2021
3	Bermuda	—	—	123091	2022	112653	2021
4	Ireland	106059	2024	103685	2023	101109	2021
...	...	...	...	...	...	...	...
218	Malawi	481	2024	673	2023	613	2021
219	South Sudan	422	2024	1072	2015	400	2021
220	Afghanistan	422	2022	353	2022	373	2021
221	Syria	—	—	421	2021	925	2021
222	Burundi	230	2024	200	2023	311	2021

The above table contains the estimated GDP per capita of all countries. This is the table that is likely to be relevant to a user interested in analyzing GDP per capita of countries. Instead of reading all tables of an HTML file, we can focus the search to tables containing certain relevant keywords. Let us try searching all table containing the word ‘Country’.

```
#Reading all the tables from the Wikipedia page on GDP per capita, containing the word 'Country'
tables = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_capita')
```

The *match* argument can be used to specify the keywords to be present in the table to be read.

```
len(tables)
```

```
1
```

Only one table contains the keyword - ‘Country’. Let us check out the table obtained.

```
#Table having the keyword - 'Country' from the HTML page
tables[0]
```

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
0	Monaco	—	—	240862	2022	234317	2021
1	Liechtenstein	—	—	187267	2022	169260	2021

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
2	Luxembourg	131384	2024	128259	2023	133745	2021
3	Bermuda	—	—	123091	2022	112653	2021
4	Ireland	106059	2024	103685	2023	101109	2021
...	...	...	...	...	...	...	...
218	Malawi	481	2024	673	2023	613	2021
219	South Sudan	422	2024	1072	2015	400	2021
220	Afghanistan	422	2022	353	2022	373	2021
221	Syria	—	—	421	2021	925	2021
222	Burundi	230	2024	200	2023	311	2021

The argument *match* helps with a more focussed search, and helps us discard irrelevant tables.

### 8.5.3 Reading JSON data

JSON stands for JavaScript Object Notation, in which the data is stored and transmitted as plain text. A couple of benefits of the JSON format are:

1. Since the format is text only, JSON data can easily be exchanged between web applications, and used by any programming language.
2. Unlike the *csv* format, JSON supports a hierarchical data structure, and is easier to integrate with APIs.

The JSON format can support a hierachical data structure, as it is built on the following two data structures (*Source: [technical documentation](#)*):

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

The *Pandas* function `read_json` converts a JSON string to a Pandas DataFrame. The function `dumps()` of the *json* library converts a Python object to a JSON string.

Lets read the JSON data on Ted Talks.

```
tedtalks_data = pd.read_json('https://raw.githubusercontent.com/cwkenwaysun/TEDmap/master/data.json')
```

```
tedtalks_data.head()
```

	id	speaker	headline	URL	description
0	7	David Pogue	Simplicity sells	http://www.ted.com/talks/view/id/7	New York Times
1	6	Craig Venter	Sampling the ocean's DNA	http://www.ted.com/talks/view/id/6	Genomics
2	4	Burt Rutan	The real future of space exploration	http://www.ted.com/talks/view/id/4	In this presentation
3	3	Ashraf Ghani	How to rebuild a broken state	http://www.ted.com/talks/view/id/3	Ashraf Ghani
4	5	Chris Bangle	Great cars are great art	http://www.ted.com/talks/view/id/5	American designer

```
[{'question': "What is the data type of values in the last column (named 'rates') of the above table?", 'type': 'multiple_choice', 'answers': [{'answer': 'list', 'correct': True, 'feedback': 'Correct!'}, {'answer': 'string', 'correct': False, 'feedback': 'Incorrect. Use the type function on the variable to find its datatype.'}, {'answer': 'numeric', 'correct': False, 'feedback': 'Incorrect. Use the type function on the variable to find its datatype.'}, {'answer': 'dictionary', 'correct': False, 'feedback': 'Incorrect. Use the type function on the variable to find its datatype.'}]]
```

This JSON data contains nested structures, such as lists and dictionaries, which require a deeper understanding to effectively structure. We will address this in future lectures.

#### 8.5.4 Reading data from a URL in Python

This process typically involves using the `requests` library, which allows you to send HTTP requests and handle responses easily.

You'll need to install it using pip:

We'll use the CoinGecko API, which provides cryptocurrency market data. Here's an example of how to retrieve current market data:

```

import requests

# Define the URL of the API
url = 'https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd'

# Send a GET request to the URL
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    # Parse the JSON data
    data = response.json()
    print(data)
else:
    print(f"Failed to retrieve data: {response.status_code}")

```

```

[{'id': 'bitcoin', 'symbol': 'btc', 'name': 'Bitcoin', 'image': 'https://coin-images.coingecko.com/btc.png'}, {'id': 'ethereum', 'symbol': 'eth', 'name': 'Ethereum', 'image': 'https://coin-images.coingecko.com/eth.png'}, {'id': 'tether', 'symbol': 'usdt', 'name': 'Tether', 'image': 'https://coin-images.coingecko.com/usdt.png'}, {'id': 'bnb', 'symbol': 'bnb', 'name': 'BNB', 'image': 'https://coin-images.coingecko.com/bnb.png'}, {'id': 'solana', 'symbol': 'sol', 'name': 'Solana', 'image': 'https://coin-images.coingecko.com/sol.png'}, {'id': 'usdc', 'symbol': 'usdc', 'name': 'USDC', 'image': 'https://coin-images.coingecko.com/usdc.png'}, {"id": "xrp", "symbol": "xrp", "name": "XRP", "image": "https://coin-images.coingecko.com/xrp.png"}, {"id": "lido", "symbol": "lio", "name": "Lido Staked Ether", "image": "https://coin-images.coingecko.com/lio.png"}, {"id": "dogecoin", "symbol": "doge", "name": "Dogecoin", "image": "https://coin-images.coingecko.com/doge.png"}, {"id": "tron", "symbol": "tron", "name": "TRON", "image": "https://coin-images.coingecko.com/tron.png"}, {"id": "toncoin", "symbol": "ton", "name": "Toncoin", "image": "https://coin-images.coingecko.com/ton.png"}, {"id": "cardano", "symbol": "ada", "name": "Cardano", "image": "https://coin-images.coingecko.com/ada.png"}, {"id": "avalanche", "symbol": "avax", "name": "Avalanche", "image": "https://coin-images.coingecko.com/avax.png"}, {"id": "shiba-inu", "symbol": "shiba", "name": "Shiba Inu", "image": "https://coin-images.coingecko.com/shiba.png"}, {"id": "wrapped-steth", "symbol": "wsteth", "name": "Wrapped stETH", "image": "https://coin-images.coingecko.com/wsteth.png"}, {"id": "wrapped-bitcoin", "symbol": "wbtc", "name": "Wrapped Bitcoin", "image": "https://coin-images.coingecko.com/wbtc.png"}, {"id": "weth", "symbol": "weth", "name": "WETH", "image": "https://coin-images.coingecko.com/weth.png"}]

# Loop through the data and print the name and current price
for coin in data:
    name = coin['name']
    price = coin['current_price']
    print(f"Coin: {name}, Price: ${price}")

```

Coin: Bitcoin, Price: \$62490  
 Coin: Ethereum, Price: \$2434.1  
 Coin: Tether, Price: \$0.999711  
 Coin: BNB, Price: \$568.59  
 Coin: Solana, Price: \$144.62  
 Coin: USDC, Price: \$0.99982  
 Coin: XRP, Price: \$0.531761  
 Coin: Lido Staked Ether, Price: \$2433.54  
 Coin: Dogecoin, Price: \$0.109138  
 Coin: TRON, Price: \$0.156189  
 Coin: Toncoin, Price: \$5.23  
 Coin: Cardano, Price: \$0.35311  
 Coin: Avalanche, Price: \$26.86  
 Coin: Shiba Inu, Price: \$1.759e-05  
 Coin: Wrapped stETH, Price: \$2870.49  
 Coin: Wrapped Bitcoin, Price: \$62339  
 Coin: WETH, Price: \$2434.35

Coin: Chainlink, Price: \$11.22  
Coin: Bitcoin Cash, Price: \$325.66  
Coin: Polkadot, Price: \$4.15  
Coin: Dai, Price: \$0.999811  
Coin: Sui, Price: \$2.06  
Coin: NEAR Protocol, Price: \$5.11  
Coin: LEO Token, Price: \$6.01  
Coin: Uniswap, Price: \$7.24  
Coin: Litecoin, Price: \$65.04  
Coin: Bittensor, Price: \$617.07  
Coin: Aptos, Price: \$8.91  
Coin: Pepe, Price: \$9.89e-06  
Coin: Wrapped eETH, Price: \$2554.88  
Coin: Artificial Superintelligence Alliance, Price: \$1.49  
Coin: Internet Computer, Price: \$8.13  
Coin: Kaspa, Price: \$0.137693  
Coin: POL (ex-MATIC), Price: \$0.376926  
Coin: Ethereum Classic, Price: \$18.7  
Coin: Stellar, Price: \$0.091505  
Coin: Monero, Price: \$144.06  
Coin: Stacks, Price: \$1.77  
Coin: First Digital USD, Price: \$0.999501  
Coin: dogwifhat, Price: \$2.56  
Coin: OKB, Price: \$41.75  
Coin: Ethena USDe, Price: \$0.998868  
Coin: Immutable, Price: \$1.49  
Coin: Filecoin, Price: \$3.74  
Coin: Aave, Price: \$146.68  
Coin: Cronos, Price: \$0.078844  
Coin: Optimism, Price: \$1.67  
Coin: Render, Price: \$5.32  
Coin: Injective, Price: \$20.63  
Coin: Arbitrum, Price: \$0.55172  
Coin: Hedera, Price: \$0.052714  
Coin: Mantle, Price: \$0.594723  
Coin: Fantom, Price: \$0.680296  
Coin: VeChain, Price: \$0.02305745  
Coin: Cosmos Hub, Price: \$4.44  
Coin: THORChain, Price: \$5.09  
Coin: WhiteBIT Coin, Price: \$11.61  
Coin: The Graph, Price: \$0.16643  
Coin: Sei, Price: \$0.436364  
Coin: Bitget Token, Price: \$1.075

Coin: Bonk, Price: \$2.134e-05  
Coin: Binance-Peg WETH, Price: \$2434.55  
Coin: FLOKI, Price: \$0.00013831  
Coin: Rocket Pool ETH, Price: \$2719.44  
Coin: Theta Network, Price: \$1.31  
Coin: Popcat, Price: \$1.28  
Coin: Arweave, Price: \$19.03  
Coin: Maker, Price: \$1406.89  
Coin: Mantle Staked Ether, Price: \$2540.6  
Coin: MANTRA, Price: \$1.4  
Coin: Pyth Network, Price: \$0.327718  
Coin: Helium, Price: \$6.89  
Coin: Solv Protocol SolvBTC, Price: \$62563  
Coin: Celestia, Price: \$5.39  
Coin: Gate, Price: \$8.86  
Coin: Jupiter, Price: \$0.772541  
Coin: Algorand, Price: \$0.125351  
Coin: Polygon, Price: \$0.377139  
Coin: Ondo, Price: \$0.711239  
Coin: Worldcoin, Price: \$1.95  
Coin: Quant, Price: \$67.87  
Coin: Lido DAO, Price: \$1.079  
Coin: KuCoin, Price: \$7.95  
Coin: JasmyCoin, Price: \$0.01934656  
Coin: Bitcoin SV, Price: \$45.98  
Coin: Conflux, Price: \$0.198997  
Coin: BitTorrent, Price: \$9.23941e-07  
Coin: Brett, Price: \$0.088115  
Coin: Core, Price: \$0.937112  
Coin: Fasttoken, Price: \$2.6  
Coin: GALA, Price: \$0.02149496  
Coin: ether.fi Staked ETH, Price: \$2424.85  
Coin: Wormhole, Price: \$0.324805  
Coin: Flow, Price: \$0.540995  
Coin: Notcoin, Price: \$0.00801109  
Coin: Beam, Price: \$0.01552574  
Coin: Renzo Restaked ETH, Price: \$2482.56  
Coin: Ethena, Price: \$0.287191  
Coin: Klaytn, Price: \$0.132756  
Coin: Aerodrome Finance, Price: \$1.2

## **8.6 Independent Study**

### **8.6.1 Practice exercise 1: Reading .csv data**

Read the file *Top 10 Albums By Year.csv*. This file contains the top 10 albums for each year from 1990 to 2021. Each row corresponds to a unique album.

#### **8.6.1.1**

Print the first 5 rows of the data.

#### **8.6.1.2**

How many rows and columns are there in the data?

#### **8.6.1.3**

Print the summary statistics of the data, and answer the following questions:

1. What proportion of albums have 15 or lesser tracks? Mention a range for the proportion.
2. What is the mean length of a track (in minutes)?

#### **8.6.1.4**

Find the album having the highest worldwide sales per year, and its artist.

#### **8.6.1.5**

Subset the data to include only Hip-Hop albums. How many Hip\_Hop albums are there?

#### **8.6.1.6**

Which album amongst hip-hop has the higest mean sales per year per track, and who is its artist?

### 8.6.2 Practice exercise 2: Reading .txt data

Read the file *bestseller\_books.txt*. It contains top 50 best-selling books on amazon from 2009 to 2019. Identify the delimiter without opening the file with Notepad or a text-editing software. How many rows and columns are there in the dataset?

**Solution:**

```
#The delimiter seems to be ';' based on the output of the above code
bestseller_books = pd.read_csv('../Data/bestseller_books.txt',sep=';')
bestseller_books.head()
```

	Unnamed: 0.1	Unnamed: 0	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

```
#The file read with ';' as the delimited is correct
print("The file has",bestseller_books.shape[0],"rows and",bestseller_books.shape[1],"columns")
```

The file has 550 rows and 9 columns

Alternatively, you can use the argument `sep = None`, and `engine = 'python'`. The default engine is C. However, the 'python' engine has a 'sniffer' tool which may identify the delimiter automatically.

```
bestseller_books = pd.read_csv('../data/bestseller_books.txt',sep=None, engine = 'python')
bestseller_books.head()
```

	Unnamed: 0.1	Unnamed: 0	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

### 8.6.3 Practice exercise 3: Reading HTML data

Read the table(s) consisting of attendance of spectators in FIFA worlds cup from this [page](#). Read only those table(s) that have the word ‘attendance’ in them. How many rows and columns are there in the table(s)?

```
dfs = pd.read_html('https://en.wikipedia.org/wiki/FIFA_World_Cup',
                   match='reaching')
print(len(dfs))
data = dfs[0]
print("Number of rows =",data.shape[0], "and number of columns=",data.shape[1])
data.head()
```

1

Number of rows = 25 and number of columns= 6

	Team	Titles	Runners-up	Third place	
0	Brazil	5 (1958, 1962, 1970, 1994, 2002)	2 (1950 *, 1998)	2 (1938, 1978)	
1	Germany	4 (1954, 1974 *, 1990, 2014)	4 (1966, 1982, 1986, 2002)	4 (1934, 1970, 2006 *, 2010)	
2	Italy	4 (1934 *, 1938, 1982, 2006)	2 (1970, 1994)	1 (1990 *)	
3	Argentina	3 (1978 *, 1986, 2022)	3 (1930, 1990, 2014)	NaN	
4	France	2 (1998 *, 2018)	2 (2006, 2022)	2 (1958, 1986)	

### 8.6.4 Practice exercise 4: Reading JSON data

Read the movies dataset from [here](#). How many rows and columns are there in the data?

```
movies_data = pd.read_json('https://raw.githubusercontent.com/vega/vega-datasets/master/data/movies.json')
print("Number of rows =",movies_data.shape[0], "and number of columns=",movies_data.shape[1])
```

Number of rows = 3201 and number of columns= 16

# 9 More on Pandas

## 9.1 Types of Data

Each column(feature) in a pandas dataframe has a datatype associated with it. Those datatypes can be grouped into **Numerical**, **Categorical**, and **Dates**.

```
<IPython.core.display.Image object>
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
movie_ratings = pd.read_csv('./datasets/movie_ratings.csv')
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
1	Major Dundee	14873	14873	3800000	Apr 07 1965	PG/PG-13
2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	PG/PG-13

The `dtypes` property is used to find the data types associated with each column in the DataFrame.

```
movie_ratings.dtypes
```

```
Title          object
US Gross       int64
Worldwide Gross    int64
Production Budget    int64
Release Date      object
```

```

MPAA Rating      object
Source          object
Major Genre     object
Creative Type   object
IMDB Rating    float64
IMDB Votes      int64
dtype: object

```

### 9.1.1 Available Data Types and Associated Built-in Functions

In a DataFrame, columns can have different data types. Here are the common data types you'll encounter and some built-in functions associated with each type:

#### 1. Numerical Data (int, float)

- Built-in functions: `mean()`, `sum()`, `min()`, `max()`, `std()`, `median()`, `quantile()`, etc.

#### 2. Object Data (str or mixed types)

- Built-in functions: `str.contains()`, `str.startswith()`, `str.endswith()`, `str.lower()`, `str.upper()`, `str.replace()`, etc.

#### 3. Datetime Data (dates)

- Built-in functions: `dt.year`, `dt.month`, `dt.day`, `dt.strftime()`, `dt.weekday()`, `dt.hour`, etc.

These functions help in exploring and transforming the data effectively depending on the type of data in each column.

### 9.1.2 Data Type Filtering

We can filter the columns based on its data types

```
# select just categorical(object) columns
movie_ratings.select_dtypes(include='object').head()
```

	Title	Release Date	MPAA Rating	Source	Major Genre	Creative Ty
0	Opal Dreams	Nov 22 2006	PG/PG-13	Adapted screenplay	Drama	Fiction
1	Major Dundee	Apr 07 1965	PG/PG-13	Adapted screenplay	Western/Musical	Fiction
2	The Informers	Apr 24 2009	R	Adapted screenplay	Horror/Thriller	Fiction

	Title	Release Date	MPAA Rating	Source	Major Genre	Creative Type
3	Buffalo Soldiers	Jul 25 2003	R	Adapted screenplay	Comedy	Fiction
4	The Last Sin Eater	Feb 09 2007	PG/PG-13	Adapted screenplay	Drama	Fiction

```
# select the numeric columns
movie_ratings.select_dtypes(include='number').head()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes
0	14443	14443	9000000	6.5	468
1	14873	14873	3800000	6.7	2588
2	315000	315000	18000000	5.2	7595
3	353743	353743	15000000	6.9	13510
4	388390	388390	2200000	5.7	1012

### 9.1.3 Data Type Conversion

Often, after the initial reading, we need to convert the datatypes of some of the columns to make them suitable for analysis, as the available functions and operations depend on the column's data type. For example, the datatype of Release Date in the DataFrame `movie_ratings` is object. To perform datetime related computations on this variable, we'll need to convert it to a datetime format. We'll use the Pandas function `to_datetime()` to convert it to a datetime format. Similar functions such as `to_numeric()`, `to_string()` etc., can be used for other conversions.

In Pandas, the `errors='coerce'` parameter is often used in the context of data conversion, specifically when using the `pd.to_numeric` function. This argument tells Pandas to convert values that it can and set the ones it cannot convert to `NaN`. It's a way of gracefully handling errors without raising an exception. Read the textbook for an example

```
# check the datatype of release date column
print(movie_ratings['Release Date'].dtypes)
movie_ratings['Release Date'].head()
```

object

```
0    Nov 22 2006
1    Apr 07 1965
2    Apr 24 2009
3    Jul 25 2003
```

```
4    Feb 09 2007
```

```
Name: Release Date, dtype: object
```

Next, we'll convert the `Release Date` column in the DataFrame to the `datetime` format to facilitate further analysis.

```
movie_ratings['Release Date'] = pd.to_datetime(movie_ratings['Release Date'])

# Let's check the datatype of release data column again
movie_ratings['Release Date'].dtypes

dtype('datetime64[ns]')
```

`dtype('datetime64[ns]')` means a 64-bit datetime object with nanosecond precision stored in little-endian format. This data type is commonly used to represent timestamps in high-resolution time series data.

### 9.1.4 Working with datetime Data

#### 9.1.4.1 the `dt` accessor

The `.dt` accessor is a powerful tool in pandas that allows you to extract and manipulate components of datetime columns in a DataFrame. This is useful for analysis and feature engineering when dealing with time-related data.

You can extract various parts of a datetime column using `.dt`

Attribute	Description	Example
<code>.dt.year</code>	Extracts the year	<code>df['Release Date'].dt.year</code>
<code>.dt.month</code>	Extracts the month (1-12)	<code>df['Release Date'].dt.month</code>
<code>.dt.day</code>	Extracts the day of the month (1-31)	<code>df['Release Date'].dt.day</code>
<code>.dt.hour</code>	Extracts the hour (0-23)	<code>df['Release Date'].dt.hour</code>
<code>.dt.minute</code>	Extracts the minute	<code>df['Release Date'].dt.minute</code>
<code>.dt.second</code>	Extracts the second	<code>df['Release Date'].dt.second</code>
<code>.dt.weekday</code>	Extracts the day of the week (0=Monday, 6=Sunday)	<code>df['Release Date'].dt.weekday</code>
<code>.dt.dayofyear</code>	Extracts the day of the year (1-366)	<code>df['Release Date'].dt.dayofyear</code>
<code>.dt.is_leap_year</code>	Checks if the year is a leap year	<code>df['Release Date'].dt.is_leap_year</code>

Let's add the year for the movie\_ratings dataframe next

```
# Extracting the year from the release date
movie_ratings['Release Year'] = movie_ratings['Release Date'].dt.year
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	2006-11-22	PG/PG-13
1	Major Dundee	14873	14873	3800000	1965-04-07	PG/PG-13
2	The Informers	315000	315000	18000000	2009-04-24	R
3	Buffalo Soldiers	353743	353743	15000000	2003-07-25	R
4	The Last Sin Eater	388390	388390	2200000	2007-02-09	PG/PG-13

In your pandas dataframe, if having start date and end date, you can calculate the time duration between them like below

```
# let's calculate the days since release till Jan 1st 2024
movie_ratings['Days Since Release'] = (pd.Timestamp('2024-01-01') - movie_ratings['Release Date'])
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	2006-11-22	PG/PG-13
1	Major Dundee	14873	14873	3800000	1965-04-07	PG/PG-13
2	The Informers	315000	315000	18000000	2009-04-24	R
3	Buffalo Soldiers	353743	353743	15000000	2003-07-25	R
4	The Last Sin Eater	388390	388390	2200000	2007-02-09	PG/PG-13

**Filtering Data:** Use extracted datetime components to filter rows. Aggregation and grouping using datetime components will be covered in future chapters.

```
# Filter rows where the release month is January
january_releases = movie_ratings[movie_ratings['Release Date'].dt.month == 1]
january_releases.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
15	Thr3e	1008849	1060418	2400000	2007-01-05	PG/PG-13
57	Impostor	6114237	6114237	40000000	2002-01-04	PG/PG-13

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
62	The Last Station	6616974	6616974	18000000	2010-01-15	R
63	The Big Bounce	6471394	6626115	50000000	2004-01-30	PG/PG-13
84	Not Easily Broken	10572742	10572742	5000000	2009-01-09	PG/PG-13

### 9.1.5 Working with object Data

In pandas, the `object` data type is a flexible data type that can store a mix of text (strings), mixed types, or arbitrary Python objects. It is commonly used for string data and is a key part of working with categorical or unstructured data in pandas.

Similar to datetime objects having a `dt` accessor, a number of specialized string methods are available when using the `str` accessor. These methods have in general matching names with the equivalent built-in string methods for single elements, but are applied element-wise on each of the values of the columns.

#### 9.1.5.1 the str accessor in pandas

The `str` accessor in pandas provides a wide range of string methods that allow for efficient and convenient text processing on an entire Series of strings. Here are some commonly used `str` methods:

- String splitting: `str.split()`
- String joining: `str.join()`
- Substrings: `str.slice(start, stop)`, `str[0]`
- String Case Conversion: `str.lower()`, `str.upper()`, `str.capitalize()`
- Whitespace Removal: `str.strip()`, `str.lstrip()`, `str.rstrip()`
- Replacing and Removing: `str.replace('old', 'new')`
- Pattern matching and extraction: `str.contains('pattern')`, `startswith('prefix')`, `endswith('suffix')`
- String length and counting: `str.len()`, `str.count()`

Let's use the well-known titanic dataset to illustrate how to manipulate string columns in pandas dataframe next

```
titanic = pd.read_csv('./Datasets/titanic.csv')
titanic.head()
```

	PassengerId	Survived	Pclass	Name	Age	SibSp	Par
0	1	0	3	Braund, Mr. Owen Harris	22.0	1	0

	PassengerId	Survived	Pclass	Name	Age	SibSp	Par
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina Futrelle, Mrs. Jacques Heath (Lily May Peel) Allen, Mr. William Henry	38.0 26.0 35.0 35.0	1 0 1 0	0 0 0 0
2	3	1	3				
3	4	1	1				
4	5	0	3				

```
titanic.Name
```

```
0 Braund, Mr. Owen Harris
1 Cumings, Mrs. John Bradley (Florence Briggs Th...
2 Heikkinen, Miss. Laina
3 Futrelle, Mrs. Jacques Heath (Lily May Peel)
4 Allen, Mr. William Henry
...
886 Montvila, Rev. Juozas
887 Graham, Miss. Margaret Edith
888 Johnston, Miss. Catherine Helen "Carrie"
889 Behr, Mr. Karl Howell
890 Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

The `Name` column varies in length and contains passengers' last names, titles, and first names. By extracting the title from the `Name` column, we could infer the `sex` of the passenger and add it as a new feature. This could potentially be a significant predictor of survival, as the “ladies first” principle was often applied during the Titanic evacuation. Adding this feature may enhance the model’s ability to predict whether a passenger survived.

```
# Let's check the length of the name of each passenger
titanic["Name"].str.len()
```

```
0    23
1    51
2    22
3    44
4    24
...
886   21
887   28
888   40
889   21
```

```
890      19
Name: Name, Length: 891, dtype: int64
```

```
# check what is the maximum length of the name
titanic.loc[titanic["Name"].str.len().idxmax(), "Name"]
```

```
'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y Vallejo)'
```

```
# get the observations that contains the word 'Mrs'
titanic[titanic.Name.str.contains('Mrs.')]
```

	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... er)	38.0	1	0
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel) er)	35.0	1	0
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg) er)	27.0	0	2
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem) er)	14.0	1	0
15	16	1	2	Hewlett, Mrs. (Mary D Kingcome) er)	55.0	0	0
...	...	...	...	...	...	...	...
871	872	1	1	Beckwith, Mrs. Richard Leonard (Sallie Monypeny) er)	47.0	1	0
874	875	1	2	Abelson, Mrs. Samuel (Hannah Wizosky) er)	28.0	1	0
879	880	1	1	Potter, Mrs. Thomas Jr (Lily Alexenia Wilson) er)	56.0	0	0
880	881	1	2	Shelley, Mrs. William (Imanita Parrish Hall) er)	25.0	0	0
885	886	0	3	Rice, Mrs. William (Margaret Norton) er)	39.0	0	0

```
# get the observations that contains the word 'Mrs'
titanic.Name.str.count('Mrs.').sum()
```

129

```
# split the name column into two columns
titanic.Name.str.split(',', expand=True)
```

	0	1
0	Braund	Mr. Owen Harris
1	Cumings	Mrs. John Bradley (Florence Briggs Thayer)
2	Heikkinen	Miss. Laina
3	Futrelle	Mrs. Jacques Heath (Lily May Peel)

	0	1
4	Allen	Mr. William Henry
...	...	...
886	Montvila	Rev. Juozas
887	Graham	Miss. Margaret Edith
888	Johnston	Miss. Catherine Helen "Carrie"
889	Behr	Mr. Karl Howell
890	Dooley	Mr. Patrick

```
# get the last part of the split
titanic.Name.str.split(',', expand=True).get(1)
```

```
0                  Mr. Owen Harris
1      Mrs. John Bradley (Florence Briggs Thayer)
2                           Miss. Laina
3      Mrs. Jacques Heath (Lily May Peel)
4                  Mr. William Henry
...
886                  Rev. Juozas
887               Miss. Margaret Edith
888      Miss. Catherine Helen "Carrie"
889                  Mr. Karl Howell
890                  Mr. Patrick
Name: 1, Length: 891, dtype: object
```

Create a new column `Title` that contains the title of the passengers

```
# from the name, extract the title
titanic['Title'] = titanic.Name.apply(lambda x: x.split(',')[-1].split('.')[0].strip())
titanic['Title']
```

```
0      Mr
1      Mrs
2     Miss
3     Mrs
4      Mr
...
886    Rev
887   Miss
888   Miss
```

```
889      Mr  
890      Mr  
Name: Title, Length: 891, dtype: object
```

```
# get the unique titles  
titanic.Title.unique()
```

```
array(['Mr', 'Mrs', 'Miss', 'Master', 'Don', 'Rev', 'Dr', 'Mme', 'Ms',  
       'Major', 'Lady', 'Sir', 'Mlle', 'Col', 'Capt', 'the Countess',  
       'Jonkheer'], dtype=object)
```

Let's create a mapping dictionary next

```
title_sex_mapping = {  
    'Mr': 'Male',  
    'Mrs': 'Female',  
    'Miss': 'Female',  
    'Master': 'Male',  
    'Don': 'Male',  
    'Rev': 'Male',  
    'Dr': 'Male', # Assumed to be Male unless you have additional context  
    'Mme': 'Female',  
    'Ms': 'Female',  
    'Major': 'Male',  
    'Lady': 'Female',  
    'Sir': 'Male',  
    'Mlle': 'Female',  
    'Col': 'Male',  
    'Capt': 'Male',  
    'the Countess': 'Female',  
    'Jonkheer': 'Male'  
}
```

```
titanic['Sex'] = titanic['Title'].map(title_sex_mapping)  
titanic.head()
```

	PassengerId	Survived	Pclass	Name	Age	SibSp	Par
0	1	0	3	Braund, Mr. Owen Harris	22.0	1	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... 38.0	1	0	0
2	3	1	3	Heikkinen, Miss. Laina	26.0	0	0

	PassengerId	Survived	Pclass	Name	Age	SibSp	Par
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	35.0	1	0
4	5	0	3	Allen, Mr. William Henry	35.0	0	0

```
titanic.Sex.value_counts()
```

```
Sex
Male      578
Female    313
Name: count, dtype: int64
```

```
# cross tabulation
pd.crosstab(titanic.Survived, titanic.Sex)
```

Sex	Female	Male
Survived		
0	81	468
1	232	110

In the Titanic disaster, more women survived than men due to the social norms and evacuation protocols followed during the sinking. The principle of “women and children first” was enforced when lifeboats were being loaded. Since there were not enough lifeboats for everyone on board, priority was given to women and children, which contributed to the higher survival rate among females compared to males.

### 9.1.5.2 the `re` module in Python is used for regular expression

The `re` module in Python is used for **regular expressions**, which are powerful tools for text analysis and manipulation. Regular expressions allow you to search, match, and manipulate strings based on specific patterns.

Common Use Cases of `re` in String Text Analysis

- Finding patterns in text: `re.search(r'\d{4}-\d{2}-\d{2}', text)` # searches for a date in the format YYYY-MM-DD
- Extracting Specific parts of a string: `re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)` #extracts email addresses from the text.
- Replacing Parts of a String: `re.sub(r'[$]\d+', '[price]', text)` #replacing any price formatted as \$ with [price].

### 9.1.5.2.1 Commonly Used Patterns in re

- \d: Matches any digit (0-9).
- \w: Matches any alphanumeric character (letters and numbers).
- \s: Matches any whitespace character (spaces, tabs, newlines).
- [a-z]: Matches any lowercase letter from a to z.
- [A-Z]: Matches any uppercase letter from A to Z.
- \*: Matches 0 or more occurrences of the preceding character.
- +: Matches 1 or more occurrences of the preceding character.
- ?: Matches 0 or 1 occurrence of the preceding character.
- ^: Matches the beginning of a string.
- \$: Matches the end of a string.
- |: Acts as an OR operator.

```
data = pd.read_html('https://en.wikipedia.org/wiki/List_of_Chicago_Bulls_seasons')
ChicagoBulls = data[2]
ChicagoBulls.head()
```

	Season	Team	Conference	Finish	Division	Finish.1	Wins	Losses	Win%	GB	Playoffs
0	1966–67	1966–67	—	—	Western	4th	33	48	0.407	11	Lost Div.
1	1967–68	1967–68	—	—	Western	4th	29	53	0.354	27	Lost Div.
2	1968–69	1968–69	—	—	Western	5th	33	49	0.402	22	NaN
3	1969–70	1969–70	—	—	Western	3rd[c]	39	43	0.476	9	Lost Div.
4	1970–71	1970–71	Western	3rd	Midwest[d]	2nd	51	31	0.622	2	Lost conf.

```
# remove all characters between box brackets including the brackets themselves in the columns
import re
def remove_brackets(x):
    return re.sub(r'\[.*?\]', '', x)

# Apply the function to each column separately using map
ChicagoBulls['Division'] = ChicagoBulls['Division'].map(remove_brackets)
ChicagoBulls['Finish'] = ChicagoBulls['Finish'].map(remove_brackets)
ChicagoBulls['Finish.1'] = ChicagoBulls['Finish.1'].map(remove_brackets)

ChicagoBulls.head()
```

	Season	Team	Conference	Finish	Division	Finish.1	Wins	Losses	Win%	GB	Playoffs
0	1966–67	1966–67	—	—	Western	4th	33	48	0.407	11	Lost Division

	Season	Team	Conference	Finish	Division	Finish.1	Wins	Losses	Win%	GB	Playoffs
1	1967–68	1967–68	—	—	Western	4th	29	53	0.354	27	Lost Division
2	1968–69	1968–69	—	—	Western	5th	33	49	0.402	22	NaN
3	1969–70	1969–70	—	—	Western	3rd	39	43	0.476	9	Lost Division
4	1970–71	1970–71	Western	3rd	Midwest	2nd	51	31	0.622	2	Lost conference

Another example

```
gdp_data = pd.read_html("https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_capita_2022")
gdp_data.head()
```

	Country/Territory	IMF[4][5]	World Bank[6]	United Nations[7]			
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
0	Monaco	—	240862	2022	240535	2022	
1	Liechtenstein	—	187267	2022	197268	2022	
2	Luxembourg	135321	2024	128259	2023	125897	2022
3	Bermuda	—	123091	2022	117568	2022	
4	Switzerland	106098	2024	99995	2023	93636	2022

```
# drop the year column
gdp_data.drop(columns=["Year"], level=1, inplace=True, axis=1)

# drop the level 1 column
gdp_data = gdp_data.droplevel(1, axis=1)
```

```
column_name_cleaner = lambda x:re.split(r'\[', x)[0]

gdp_data.columns = gdp_data.columns.map(column_name_cleaner)

gdp_data.head()
```

	Country/Territory	IMF	World Bank	United Nations
0	Monaco	—	240862	240535
1	Liechtenstein	—	187267	197268
2	Luxembourg	135321	128259	125897

	Country/Territory	IMF	World Bank	United Nations
3	Bermuda	—	123091	117568
4	Switzerland	106098	99995	93636

### 9.1.5.3 the NLTK Library for NLP (skipped)

NLTK (Natural Language Toolkit) is a popular Python library for natural language processing (NLP) and text analysis. It provides a wide range of tools and resources for processing and analyzing human language data. NLTK is widely used in research, education, and industry for various text analysis tasks.

#### 9.1.5.3.1 Key Features and Capabilities of NLTK

- **Tokenization:** Splits text into individual words (word tokenization) or sentences (sentence tokenization).
- **Stop Word Removal:** Provides lists of common words (like “and”, “the”, “is”) in various languages that can be removed from text to reduce noise.
- **Stemming:** Reduces words to their root form (e.g., “running” to “run”).
- **Lemmatization:** Similar to stemming, but more sophisticated. It reduces words to their dictionary form using vocabulary and morphological analysis (e.g., “better” to “good”).

## 9.1.6 Working with numerical Data

### 9.1.6.1 Summary statistics across rows/columns in Pandas

The Pandas DataFrame class has functions such as `sum()` and `mean()` to compute sum over rows or columns of a DataFrame.

By default, functions like `mean()` and `sum()` compute the statistics for each column (i.e., all rows are aggregated) in the DataFrame. Let us compute the mean of all the numeric columns of the data:

```
movie_ratings.describe()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes
count	2.228000e+03	2.228000e+03	2.228000e+03	2228.000000	2228.000000
mean	5.076370e+07	1.019370e+08	3.816055e+07	6.239004	33585.154847
std	6.643081e+07	1.648589e+08	3.782604e+07	1.243285	47325.651561
min	0.000000e+00	8.840000e+02	2.180000e+02	1.400000	18.000000

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes
25%	9.646188e+06	1.320737e+07	1.200000e+07	5.500000	6659.250000
50%	2.838649e+07	4.266892e+07	2.600000e+07	6.400000	18169.000000
75%	6.453140e+07	1.200000e+08	5.300000e+07	7.100000	40092.750000
max	7.601676e+08	2.767891e+09	3.000000e+08	9.200000	519541.000000

```
# select the numeric columns
movie_ratings.mean(numeric_only=True)
```

```
US Gross           5.076370e+07
Worldwide Gross   1.019370e+08
Production Budget 3.816055e+07
IMDB Rating        6.239004e+00
IMDB Votes         3.358515e+04
Release Year       2.002005e+03
ratio_wgross_by_budget 1.259483e+01
dtype: float64
```

### Using the axis parameter:

The `axis` parameter controls whether to compute the statistic across rows or columns:  
\* The argument `axis=0`(default) denotes that the mean is taken over all the rows of the DataFrame.  
\* For computing a statistic across column the argument `axis=1` will be used.

If mean over a subset of columns is desired, then those column names can be subset from the data.

For example, let us compute the mean IMDB rating, and mean IMDB votes of all the movies:

```
movie_ratings[['IMDB Rating', 'IMDB Votes']].mean(axis = 0)
```

```
IMDB Rating      6.239004
IMDB Votes       33585.154847
dtype: float64
```

### Pandas sum function

```
data = [[10, 18, 11], [13, 15, 8], [9, 20, 3]]
df = pd.DataFrame(data )
df
```

	0	1	2
0	10	18	11
1	13	15	8
2	9	20	3

```
# By default, the sum method adds values accross rows and returns the sum for each column
df.sum()
```

```
0    32
1    53
2    22
dtype: int64
```

```
# By specifying the column axis (axis='columns'), the sum() method add values accross columns
df.sum(axis = 'columns')
```

```
0    39
1    36
2    32
dtype: int64
```

```
# in python, axis=1 stands for column, while axis=0 stands for rows
df.sum(axis = 1)
```

```
0    39
1    36
2    32
dtype: int64
```

# 10 Introduction to Data Visualization

```
<IPython.core.display.Image object>
```

*“One picture is worth a thousand words”* - Fred R. Barnard

Visual perception offers the highest bandwidth channel, as we acquire much more information through visual perception than with all of the other channels combined, as billions of our neurons are dedicated to this task. Moreover, the processing of visual information is, at its first stages, a highly parallel process. Thus, it is generally easier for humans to comprehend information with plots, diagrams and pictures, rather than with text and numbers. This makes data visualizations a vital part of data science. Some of the key purposes of data visualization are:

1. Data visualization is the first step towards exploratory data analysis (EDA), which reveals trends, patterns, insights, or even irregularities in data.
2. Data visualization can help explain the workings of complex mathematical models.
3. Data visualization are an elegant way to summarise the findings of a data analysis project.
4. Data visualizations (especially interactive ones such as those on Tableau) may be the end-product of data analytics project, where the stakeholders make decisions based on the visualizations.

## 10.1 The Art of Visualization: Choosing the Right Plot Type

There are various types of plots available, and selecting the appropriate one is crucial for successful data visualization. The choice primarily depends on two factors: \* The type of data you are working with, and \* The role of visualization in your data analysis

### 10.1.1 Data Classification for Visualization

Data visualization is commonly used to plot data in a pandas DataFrame. The data can be classified into two categories:

- Numeric Data: This type of data represents quantities and can take any value within a range. Common examples include age, height, temperature, etc.

- Categorical Data: This type of data represents distinct categories or groups. It can be nominal (no inherent order, like colors or names) or ordinal (with a defined order, like ratings).

### **10.1.2 The Role of Visualization in Data Analysis**

Data visualization is essential for effectively communicating insights derived from data analysis. By using various visualization techniques, we can **uncover patterns, and understand relationships**. Below, we discuss different types of data exploration and the relevant visualizations used for each.

#### **10.1.2.1 Univariate Exploration**

**Purpose:** Univariate exploration analyzes a single variable to understand its distribution, central tendency, and spread.

##### **10.1.2.1.1 Common Visualizations:**

- **Histograms:** Display the frequency distribution of a numeric variable, helping to identify the shape of the data (e.g., normal, skewed).
- **Box Plots:** Summarize key statistics of a variable, including median, quartiles, and potential outliers.
- **Bar Plots:** Show the count or proportion of categorical variables, revealing the frequency of each category.
- **Line Plots:** Used to display trends in numeric data over time, helping to visualize changes in a variable.

##### **10.1.2.1.2 Insights Gained:**

- Identify outliers and anomalies.
- Understand the range and distribution of values.
- Determine central tendency (mean, median, mode).

### **10.1.2.2 Bivariate Analysis**

**Purpose:** Bivariate analysis examines the relationship between two variables, helping to understand how changes in one variable might affect another.

#### **10.1.2.2.1 Common Visualizations:**

- **Scatter Plots:** Illustrate the relationship between two numeric variables, highlighting trends and correlations.
- **Grouped Bar Plots:** Compare categorical variables against a numeric variable, revealing trends across categories.
- **Heatmaps:** Represent correlation coefficients between pairs of variables, allowing easy identification of strong correlations.

#### **10.1.2.2.2 Insights Gained:**

- Assess the strength and direction of relationships (positive, negative, or no correlation).
- Identify potential predictive relationships for further analysis.
- Discover patterns that may indicate causal relationships.

#### **10.1.2.3 Multivariate Analysis**

**Purpose:** Multivariate analysis investigates more than two variables simultaneously, providing a comprehensive view of complex relationships and interactions.

#### **10.1.2.3.1 Common Visualizations:**

- **Pair Plots:** Show pairwise relationships in a dataset, facilitating quick insights into correlations among multiple variables.
- **3D Scatter Plots:** Visualize the interaction between three numeric variables in a three-dimensional space.
- **Facet Grids:** Display multiple plots for different subsets of data, enabling comparisons across categories.

#### **10.1.2.3.2 Insights Gained:**

- Understand interactions and dependencies among multiple variables.
- Identify clusters or groups within the data.
- Enhance predictive modeling by considering multiple influences.

### 10.1.3 Summary

Choosing the appropriate plot depends on the data type and the specific analysis purpose. Numeric data typically requires plots that can handle continuous data (like line plots or histograms), while categorical data often benefits from comparisons (like bar plots or pie charts). Always consider what story you want to tell with your data and select your visualization method accordingly.

## 10.2 Visualization Tools

We'll use three libraries for making data visualizations - pandas, [matplotlib](#), and [seaborn](#).

To get started, let's import these libraries.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# let's import numpy as well
import numpy as np
```

### 10.2.1 Basic Plotting with Pandas

In previous chapters, we focused on using the pandas library for data reading and analysis. In addition to its powerful data manipulation capabilities, pandas also provides tools for creating basic plots, making it especially valuable for exploratory data analysis.

In this section, we will use the COVID dataset to demonstrate basic plotting techniques with pandas.

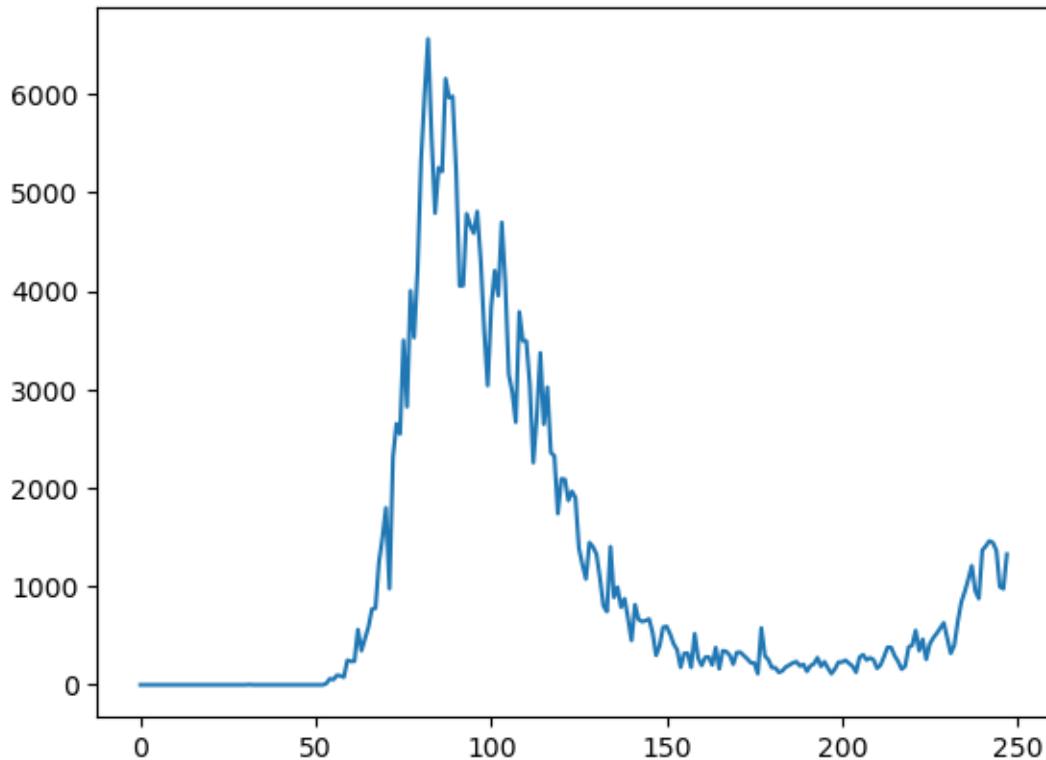
```
covid_df = pd.read_csv('../Datasets/covid.csv')
covid_df.head(5)
```

	date	new_cases	total_cases	new_deaths	total_deaths	new_tests	total_tests	cases_per
0	2019-12-31	0.0	0.0	0.0	0.0	NaN	NaN	0.0
1	2020-01-01	0.0	0.0	0.0	0.0	NaN	NaN	0.0
2	2020-01-02	0.0	0.0	0.0	0.0	NaN	NaN	0.0
3	2020-01-03	0.0	0.0	0.0	0.0	NaN	NaN	0.0
4	2020-01-04	0.0	0.0	0.0	0.0	NaN	NaN	0.0

Let's begin by visualizing the trend of new COVID cases over time using a line plot.

A line plot is ideal for showing changes over continuous data, such as the progression of new cases over a series of dates.

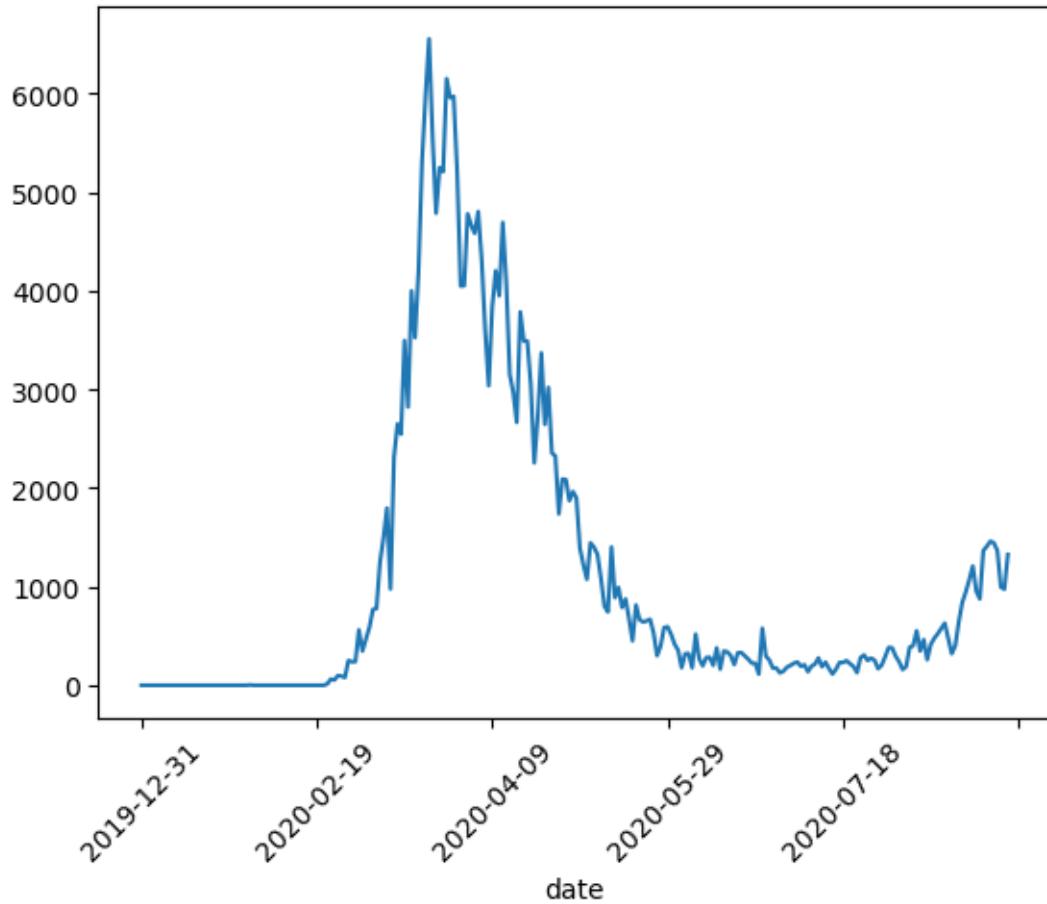
```
covid_df.new_cases.plot()
```



While this plot shows the overall trend, it's hard to tell where the peak occurred, as there are no dates on the X-axis. We can use the date column as the index for the data frame to address this issue since it is a time series dataset

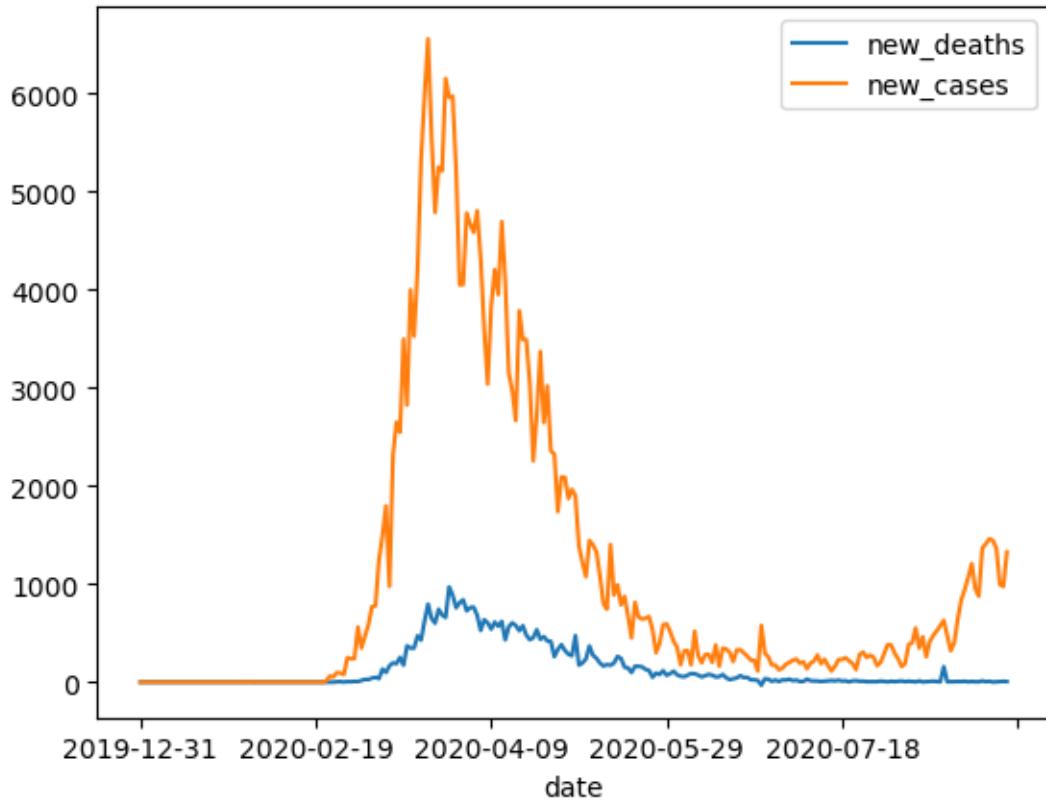
```
covid_df.set_index('date', inplace=True)
```

```
covid_df.new_cases.plot(rot=45)
```



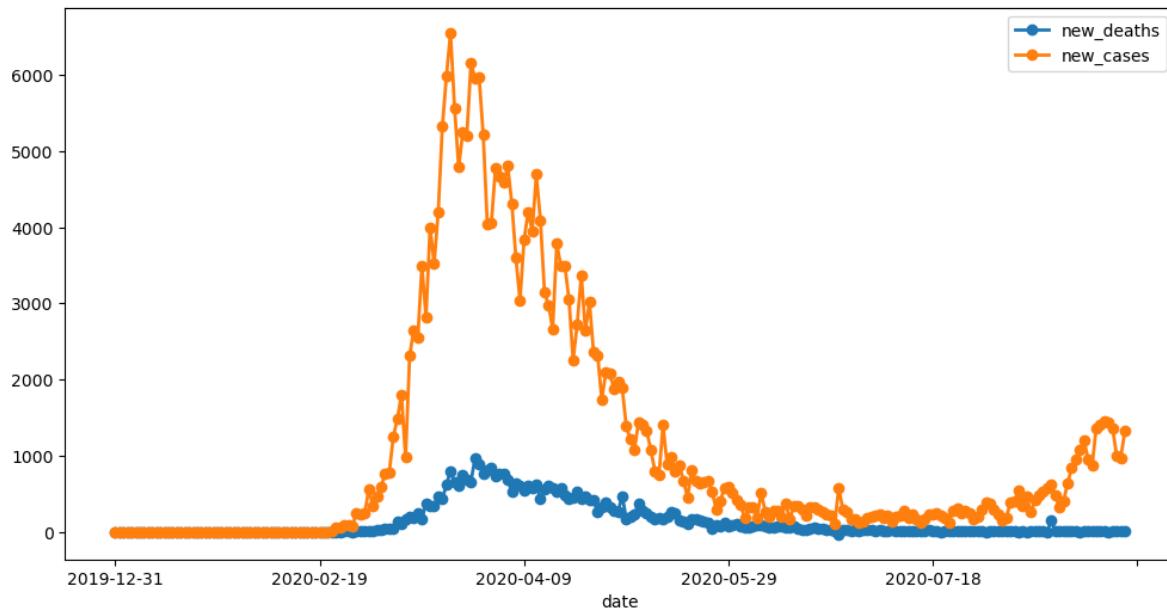
With the date set as the index, we can observe that the peak occurred around March 2020. Next, let's plot both new cases and new deaths together to compare their trends over the same time period.

```
covid_df[['new_deaths', 'new_cases']].plot()
```



By default, pandas generates line plots when using the `.plot` method. However, there are several parameters you can adjust to enhance the appearance of the line plot.

```
covid_df[['new_deaths', 'new_cases']].plot(figsize=(12, 6), linewidth=2, marker='o')
```

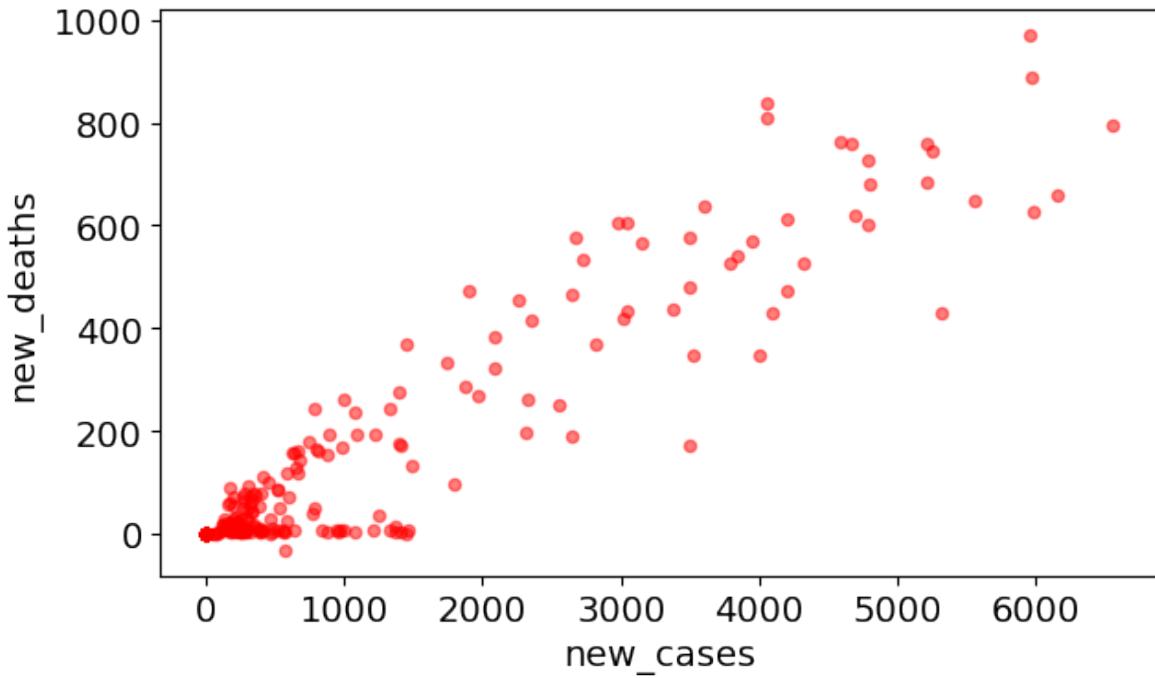


You can create other types of visualizations by setting the kind parameter in the plot function. The kind parameter accepts eleven different string values, which specify the type of plot:

- “area” is for area plots.
- “bar” is for vertical bar charts.
- “barh” is for horizontal bar charts.
- “box” is for box plots.
- “hexbin” is for hexbin plots.
- “hist” is for histograms.
- “kde” is for kernel density estimate charts.
- “density” is an alias for “kde”.
- “line” is for line graphs.
- “pie” is for pie charts.
- “scatter” is for scatter plots.

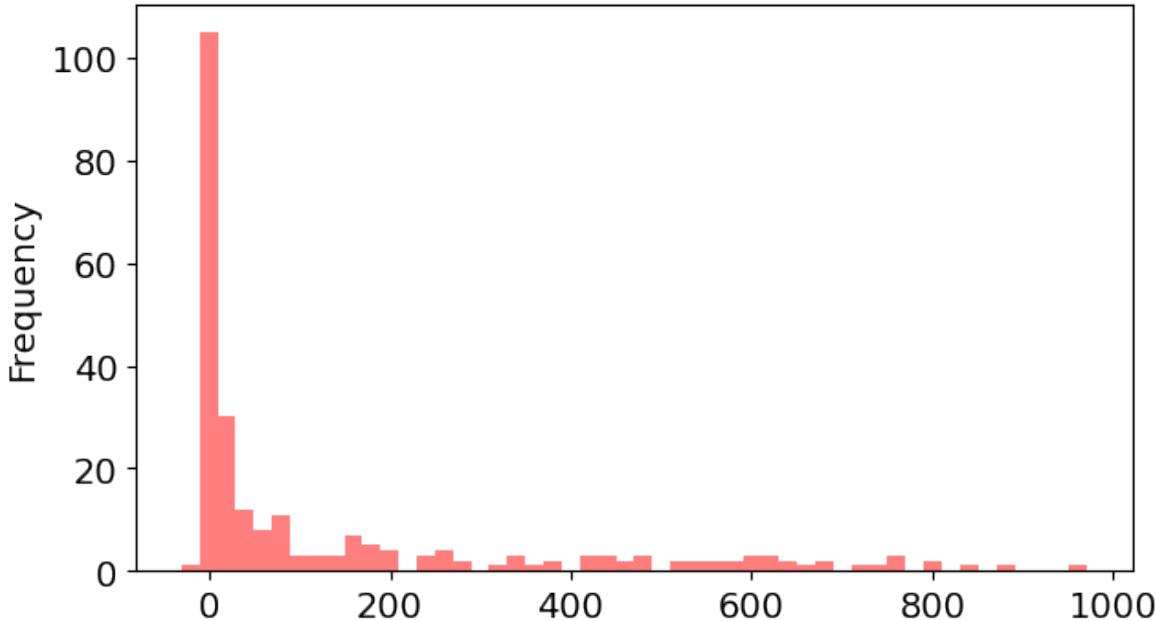
Let’s next create a scatter plot to visualize the relationship between new cases and new deaths, and explore whether there’s a correlation between them.

```
covid_df.plot(kind='scatter', x='new_cases', y='new_deaths', color='r', alpha=0.5);
```

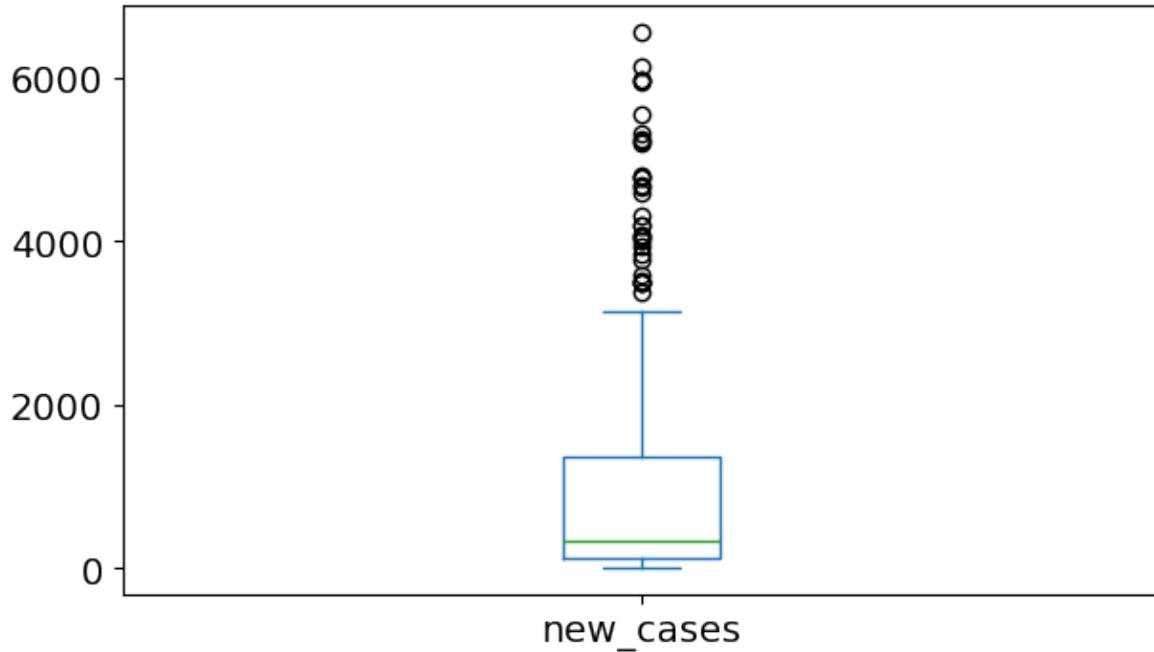


Next, let's examine the distribution of new deaths using a histogram.

```
covid_df.new_deaths.plot(kind='hist', color='r', alpha=0.5, bins=50);
```



```
covid_df.new_cases.plot(kind='box');
```



For more plot types and detailed information, refer to the official pandas documentation:

- [Series.plot](#)
- [DataFrame.plot](#)

#### 10.2.1.1 Limitation of using pandas for plotting

While pandas provides a straightforward way to create plots, there are some limitations to be aware of:

- Customization: Pandas offers basic customization options, but it may not provide the level of detail or flexibility that you can achieve with matplotlib directly. For complex visualizations, you might need to switch to matplotlib for more control.
- Plot Types: The range of plot types available in pandas is limited compared to what you can create with matplotlib or seaborn. For instance, advanced plots like violin plots or 3D plots require switching to other libraries.
- Aesthetic Choices: The default aesthetics in pandas may not be as visually appealing as those created using seaborn or other specialized visualization libraries. For polished presentations, additional customization might be necessary.

## 10.2.2 Data Plotting with Matplotlib Pyplot Interface

Pandas data visualization is built on top of matplotlib. When you use the `.plot()` method in pandas, it internally calls matplotlib functions to create the plots.

Matplotlib is:

- a low-level graph plotting library in python that strives to emulate MATLAB,
- can be used in Python scripts, Python and IPython shells, Jupyter notebooks and web application servers.
- is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

### 10.2.2.1 Matplotlib pyplot

- Matplotlib is the whole package; pyplot is a module in matplotlib
- Most of the Matplotlib utilities lies under the pyplot module, and are usually imported under the plt alias:

```
import matplotlib.pyplot as plt
```

### 10.2.2.2 Data Source

- Python lists, NumPy arrays as well as a pandas series
- However, all the sequences are internally converted to numpy arrays.

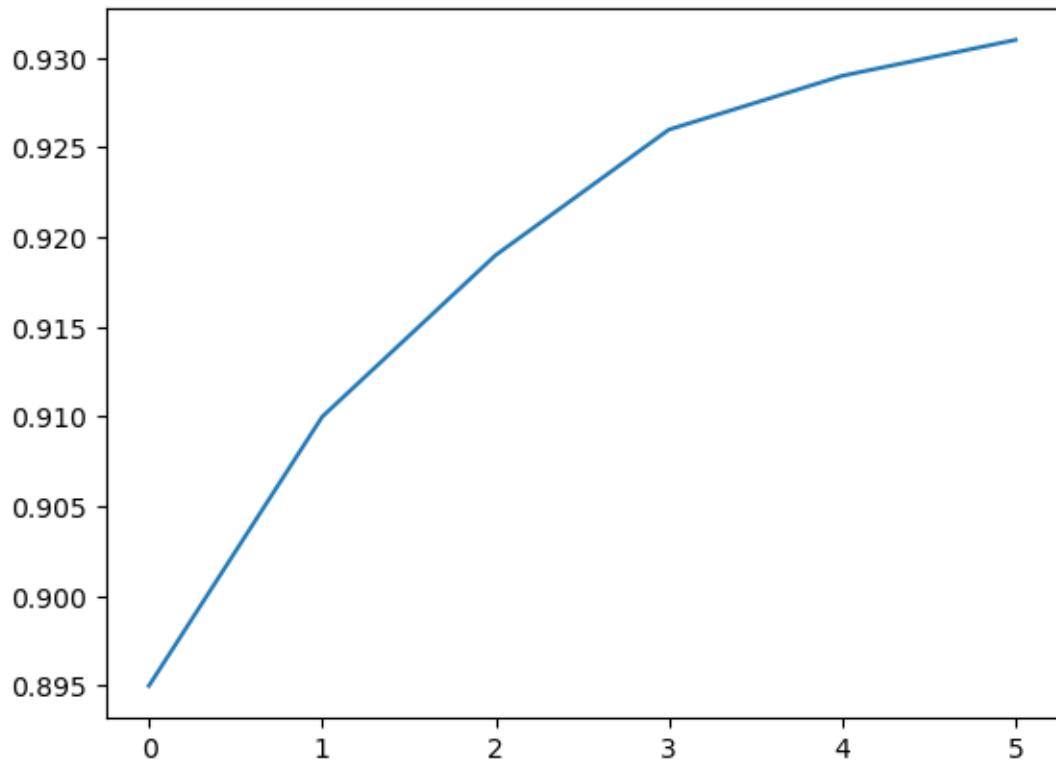
Let's create a python list to illustrate basic plotting with Matplotlib pyplot

```
yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
```

### 10.2.2.3 Basic Plotting

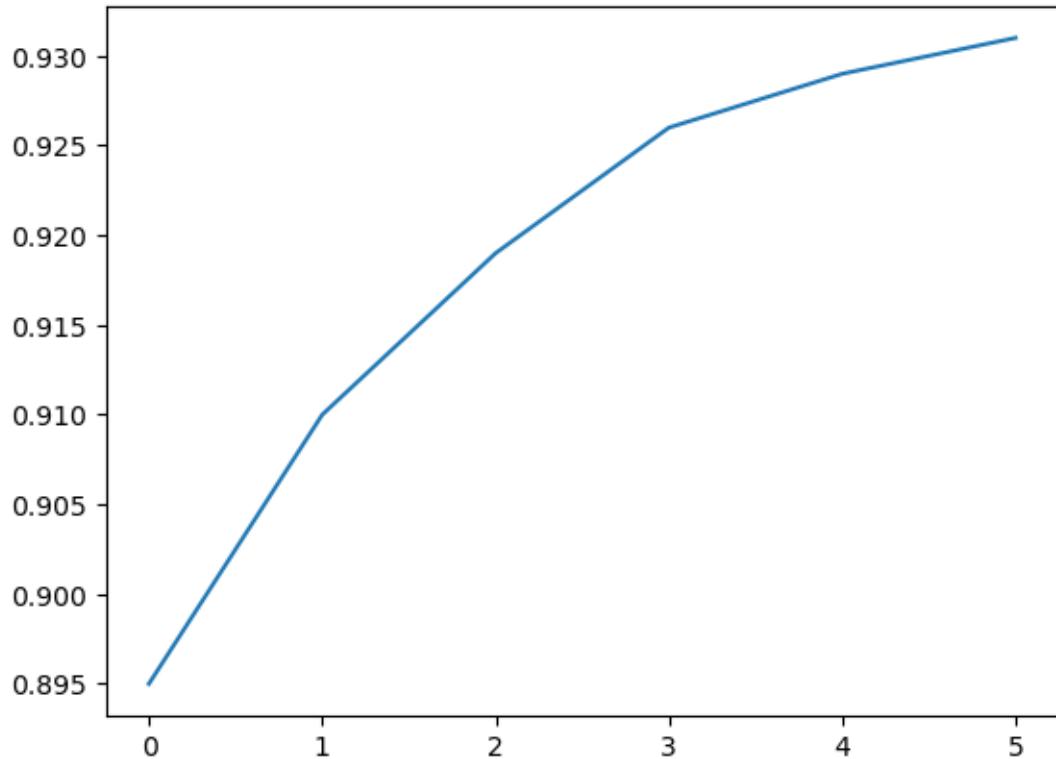
#### 10.2.2.3.1 Plotting the overall trend

```
plt.plot(yield_apples)
```



Calling the `plt.plot` function draws the line chart as expected. It also returns a list of plots drawn [`<matplotlib.lines.Line2D at 0x2194b571df0>`], shown within the output. We can include a semicolon (;) at the end of the last statement in the cell to avoiding showing the output and display just the graph.

```
plt.plot(yield_apples);
```

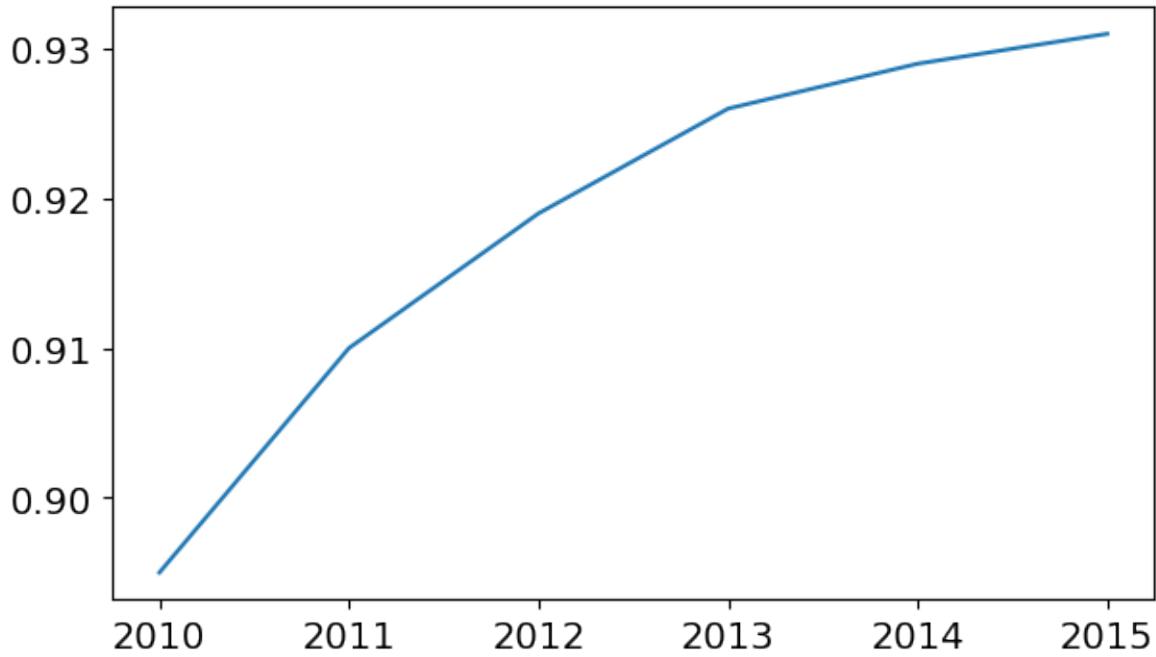


#### 10.2.2.3.2 Customizing the X-axis

The X-axis of the plot currently shows list element indexes 0 to 5. The plot would be more informative if we could display the year for which we're plotting the data. We can do this by two arguments `plt.plot`.

```
years = [2010, 2011, 2012, 2013, 2014, 2015]
yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
```

```
plt.plot(years, yield_apples);
```

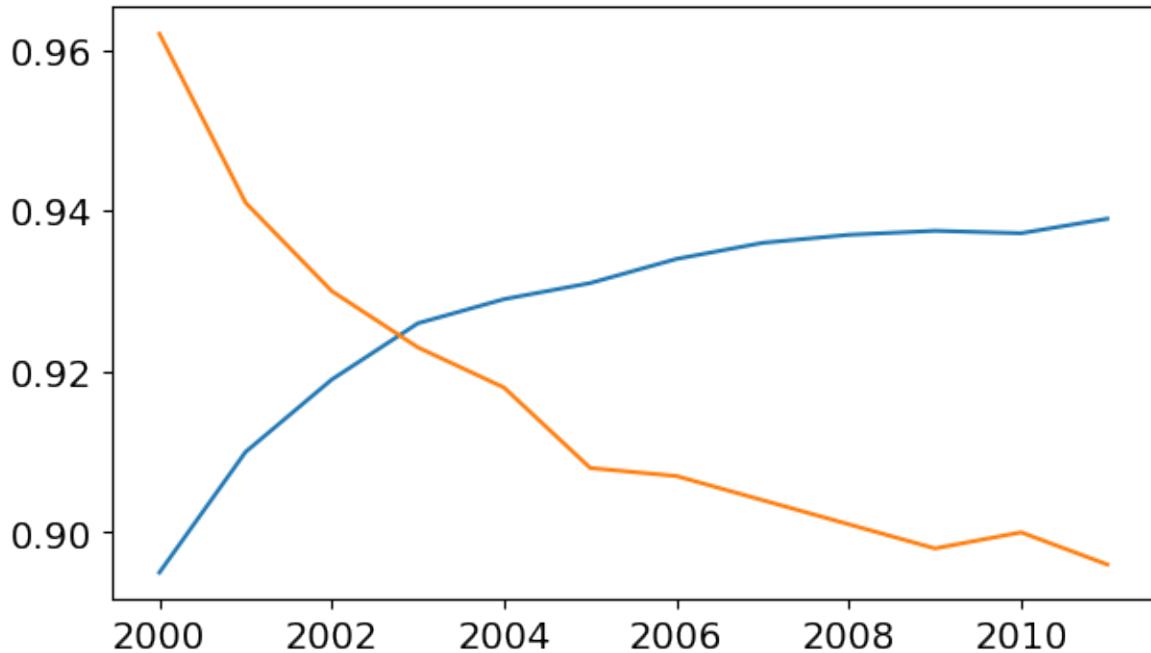


#### 10.2.2.3.3 Plotting multiple lines

You can invoke the `plt.plot` function once for each line to plot multiple lines in the same graph. Let's compare the yields of apples vs. oranges.

```
years = range(2000, 2012)
apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931, 0.934, 0.936, 0.937, 0.9375, 0.9372, 0.937,
oranges = [0.962, 0.941, 0.930, 0.923, 0.918, 0.908, 0.907, 0.904, 0.901, 0.898, 0.9, 0.896,
```

```
plt.plot(years, apples)
plt.plot(years, oranges);
```



When plt.plot command is called without any formatting parameters, pyplot uses the following defaults:

- Figure size: 6.4 X 4.8 inches
- Plot style: solid line
- Linewidth: 1.5
- Color: Blue (code ‘b’, hex code: ‘#1f77b4’)

You can also edit default styles directly by modifying the `matplotlib.rcParams` dictionary. Learn more: <https://matplotlib.org/3.2.1/tutorials/introductory/customizing.html#matplotlib-rcparams>.

You can customize default plot styles by directly modifying the `matplotlib.rcParams` dictionary. For more details, visit the official Matplotlib guide on [customizing with rcParams](#).

```
import matplotlib
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (7, 4)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

**Conceptual model:** Plotting in Matplotlib involves multiple levels of control, from setting the figure size to customizing individual text elements. To offer complete control over the plotting process, Matplotlib provides an object-oriented interface in a hierarchical structure. This approach allows users to create and manage `Figure` and `Axes` objects, which serve as

the foundation for all plotting actions. In the next chapter, you will explore how to use this object-oriented interface to gain more precise control over your plots.

#### 10.2.2.4 Enhancing the plot

Matplotlib provides a wide range of customizable components within a figure, allowing for fine-tuned control over every aspect of the plot. These components include elements like axes, labels, ticks, legends, and the overall layout. Each can be tailored to enhance the clarity, aesthetics, and effectiveness of the visual representation, making the plot more engaging and easier to interpret.

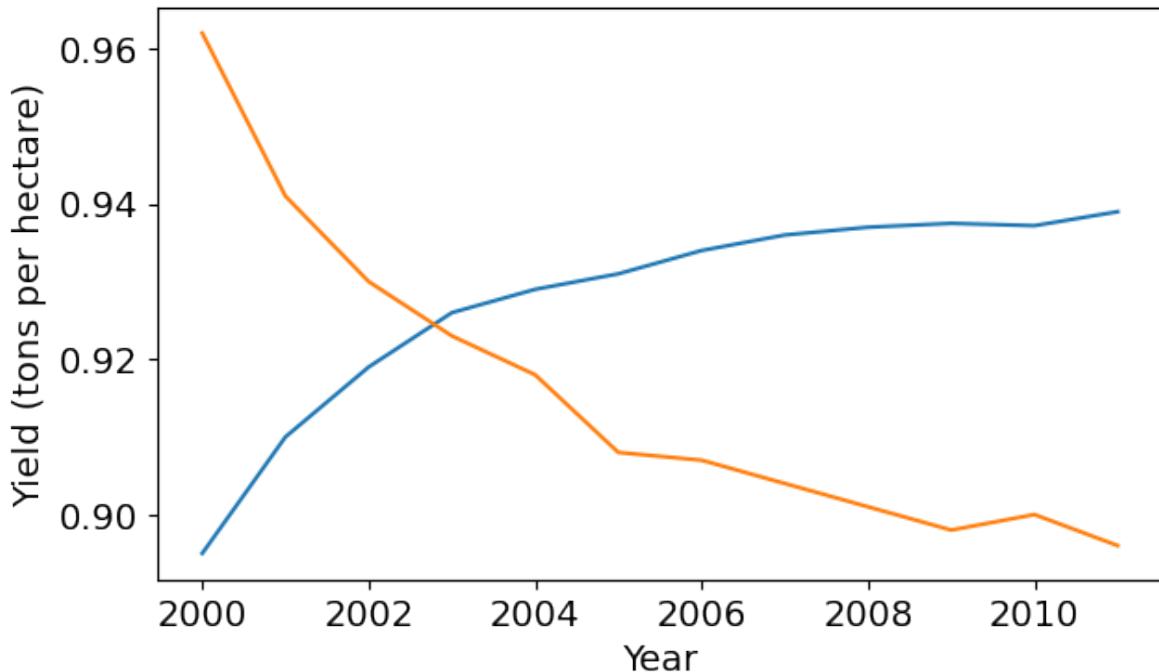
```
<IPython.core.display.Image object>
```

Figure 10.1: Matplotlib anatomy of a figure

##### 10.2.2.4.1 Adding Axis Labels

We can add labels to the axes to show what each axis represents using the `plt.xlabel` and `plt.ylabel` methods.

```
plt.plot(years, apples)
plt.plot(years, oranges)
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)');
```



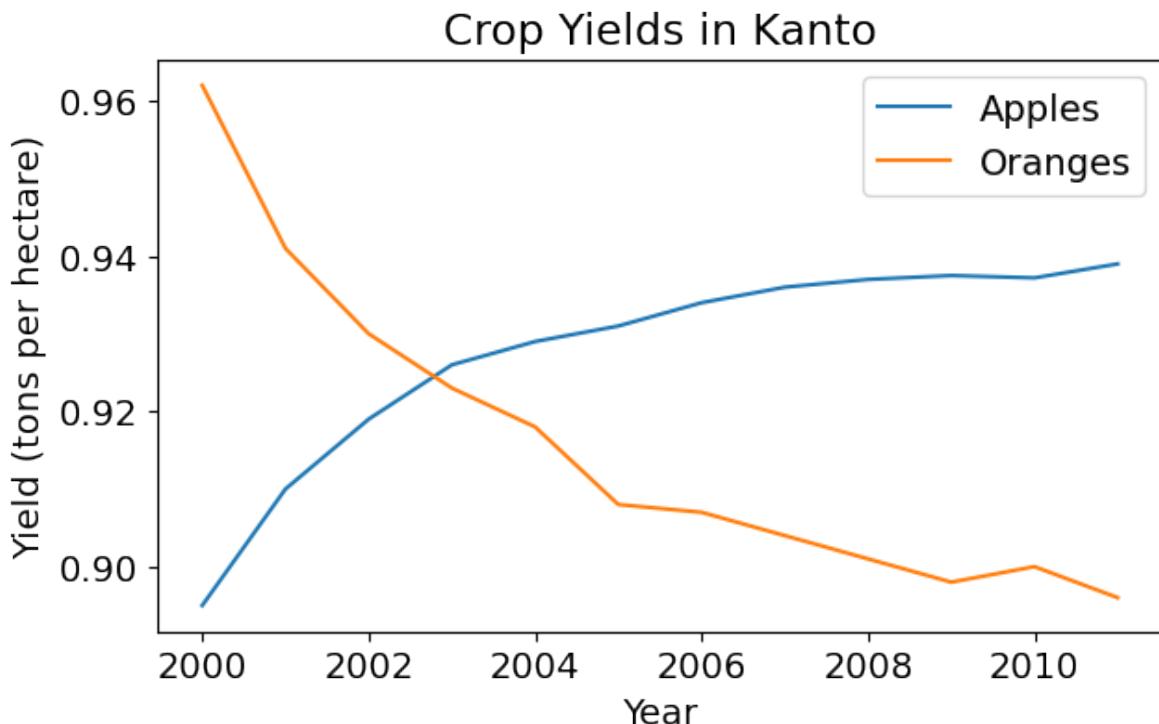
#### 10.2.2.4.2 Adding Chart Title and Legend

To differentiate between multiple lines, we can include a legend within the graph using the `plt.legend` function. We can also set a title for the chart using the `plt.title` function.

```
plt.plot(years, apples)
plt.plot(years, oranges)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



#### 10.2.2.4.3 Adding Line Markers

We can also show markers for the data points on each line using the `marker` argument of `plt.plot`. Matplotlib provides many different markers, like a circle, cross, square, diamond, etc. You can find the full list of marker types [here](#).

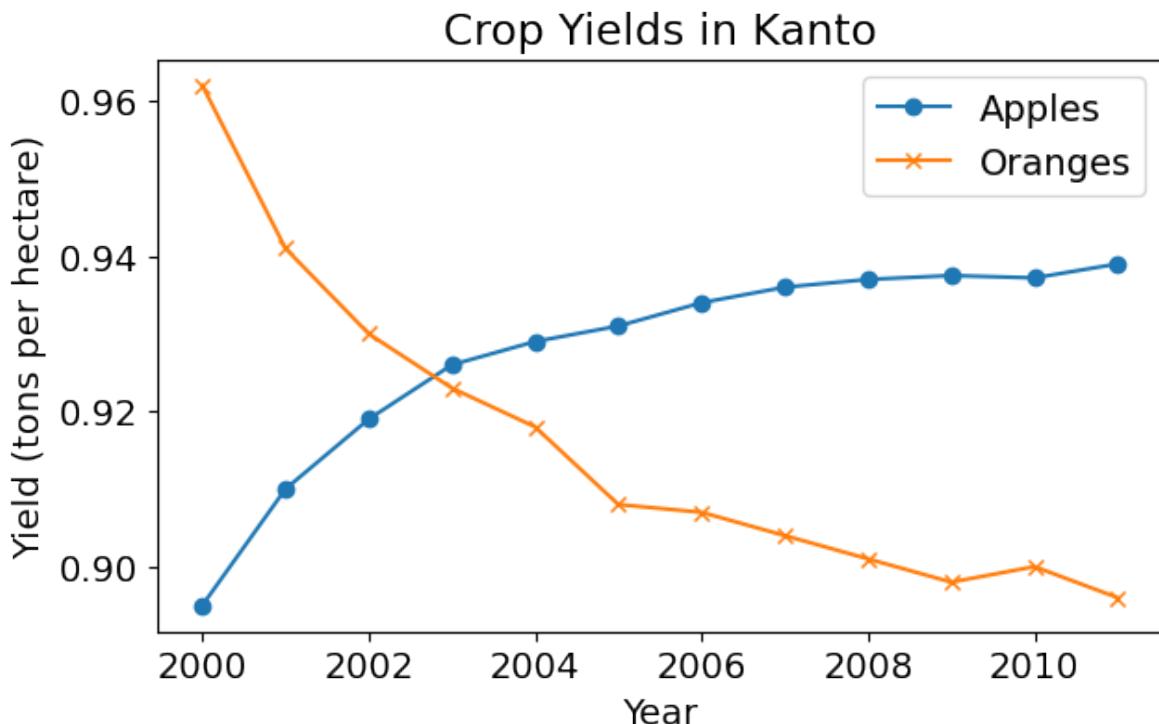
```

plt.plot(years, apples, marker='o')
plt.plot(years, oranges, marker='x')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);

```



#### 10.2.2.4.4 Styling Lines and Markers

The `plt.plot` function supports many arguments for styling lines and markers:

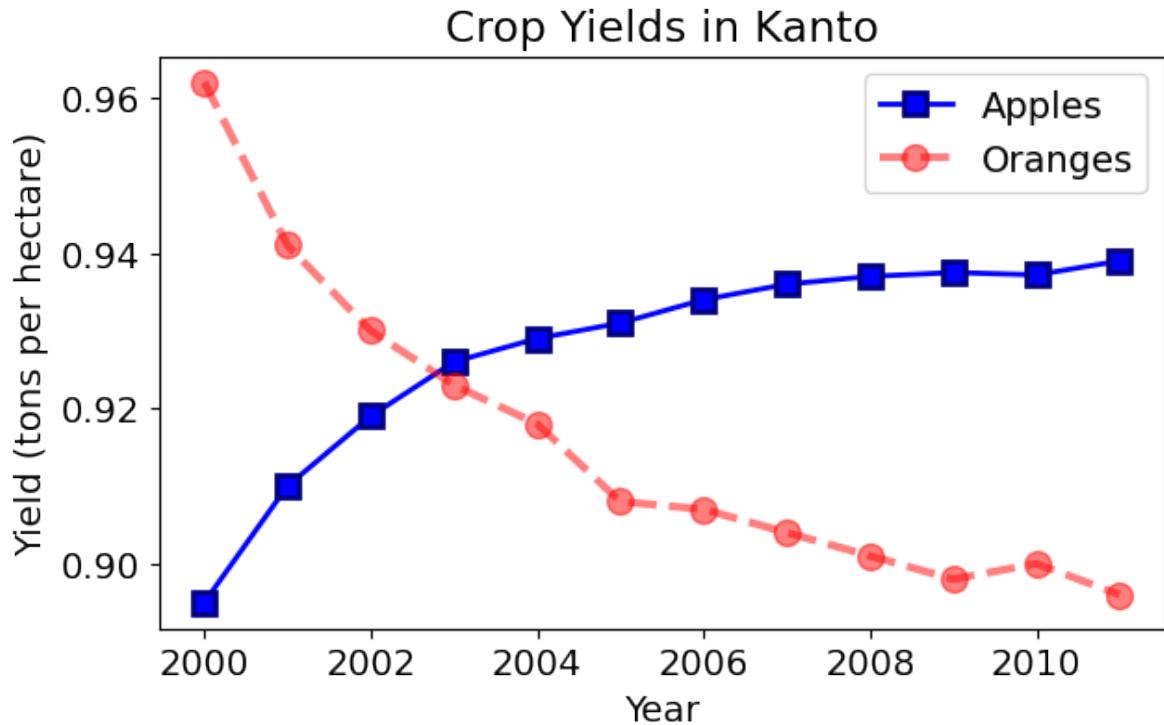
- `color` or `c`: Set the color of the line ([supported colors](#))
- `linestyle` or `ls`: Choose between a solid or dashed line
- `linewidth` or `lw`: Set the width of a line
- `markersize` or `ms`: Set the size of markers
- `markeredgecolor` or `mec`: Set the edge color for markers
- `markeredgewidth` or `mew`: Set the edge width for markers
- `markerfacecolor` or `mfc`: Set the fill color for markers
- `alpha`: Opacity of the plot

Check out the documentation for `plt.plot` to learn more: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot) .

```
plt.plot(years, apples, marker='s', c='b', ls='-', lw=2, ms=8, mew=2, mec='navy')
plt.plot(years, oranges, marker='o', c='r', ls='--', lw=3, ms=10, alpha=.5)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')
```

```
plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



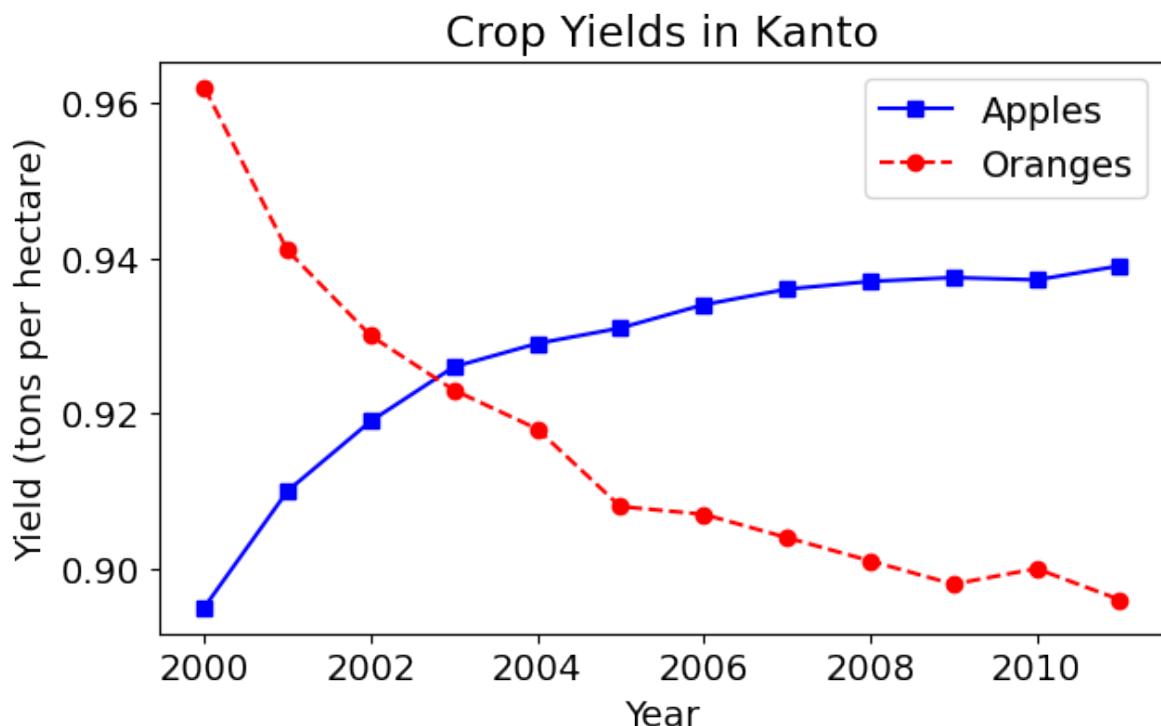
The `fmt` argument provides a shorthand for specifying the marker shape, line style, and line color. It can be provided as the third argument to `plt.plot`.

```
fmt = '[marker][line][color]'

plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

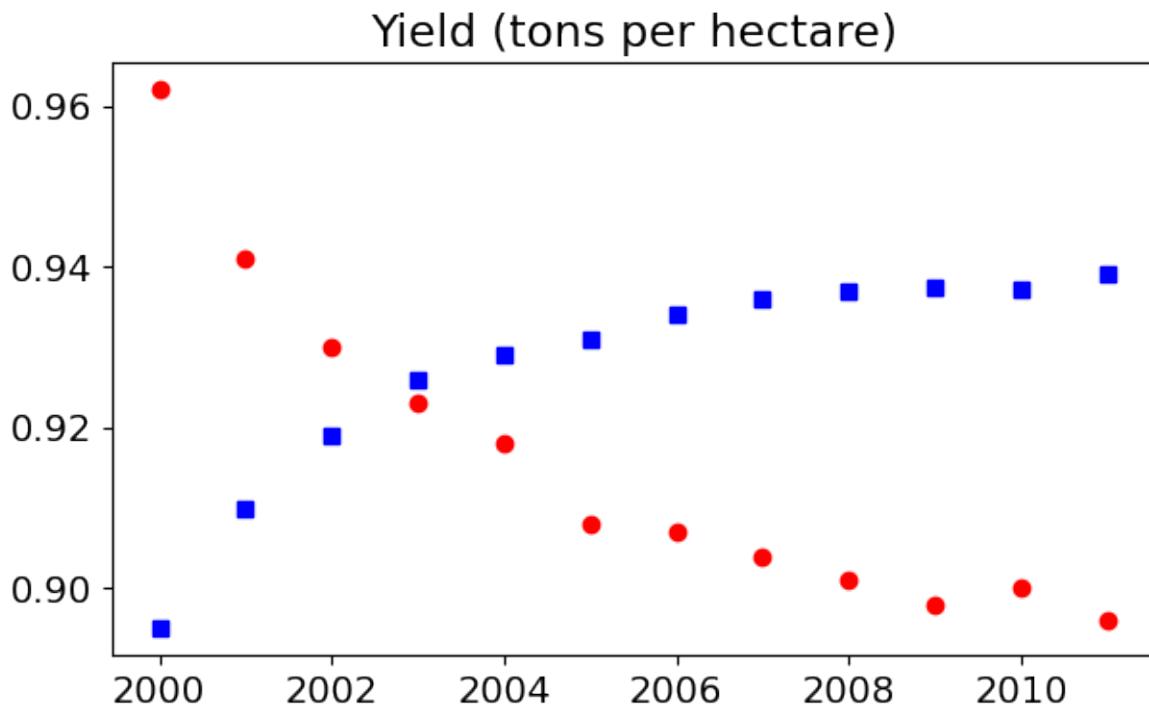
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



If you don't specify a line style in `fmt`, only markers are drawn.

```
plt.plot(years, apples, 'sb')
plt.plot(years, oranges, 'or')
plt.title("Yield (tons per hectare)");
```

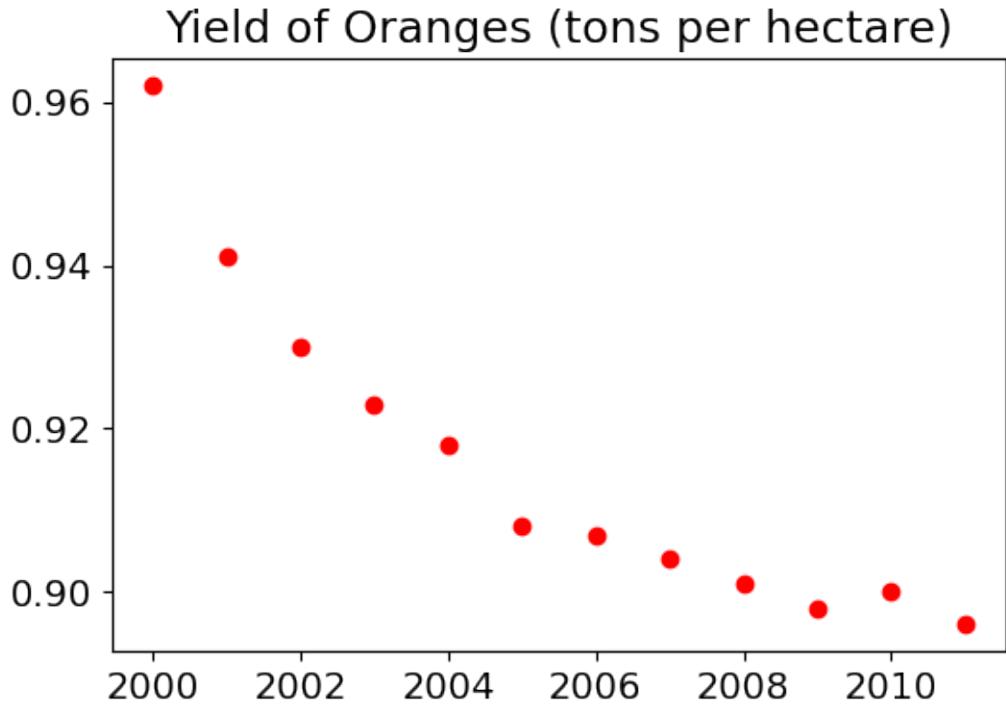


#### 10.2.2.4.5 Changing the Figure Size

You can use the `plt.figure` function to change the size of the figure.

```
plt.figure(figsize=(6, 4))

plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```



#### 10.2.2.5 Plotting other types of plots with matplotlib pyplot

Let's read the *fifa\_data.csv* as our data source

```
fifa = pd.read_csv('./Datasets/fifa_data.csv')
fifa.head(5)
```

	Unnamed: 0	ID	Name	Age	Photo	Nation
0	0	158023	L. Messi	31	https://cdn.sofifa.org/players/4/19/158023.png	Argen...
1	1	20801	Cristiano Ronaldo	33	https://cdn.sofifa.org/players/4/19/20801.png	Portug...
2	2	190871	Neymar Jr	26	https://cdn.sofifa.org/players/4/19/190871.png	Brazil...
3	3	193080	De Gea	27	https://cdn.sofifa.org/players/4/19/193080.png	Spain...
4	4	192985	K. De Bruyne	27	https://cdn.sofifa.org/players/4/19/192985.png	Belgiu...

##### 10.2.2.5.1 Histogram

```

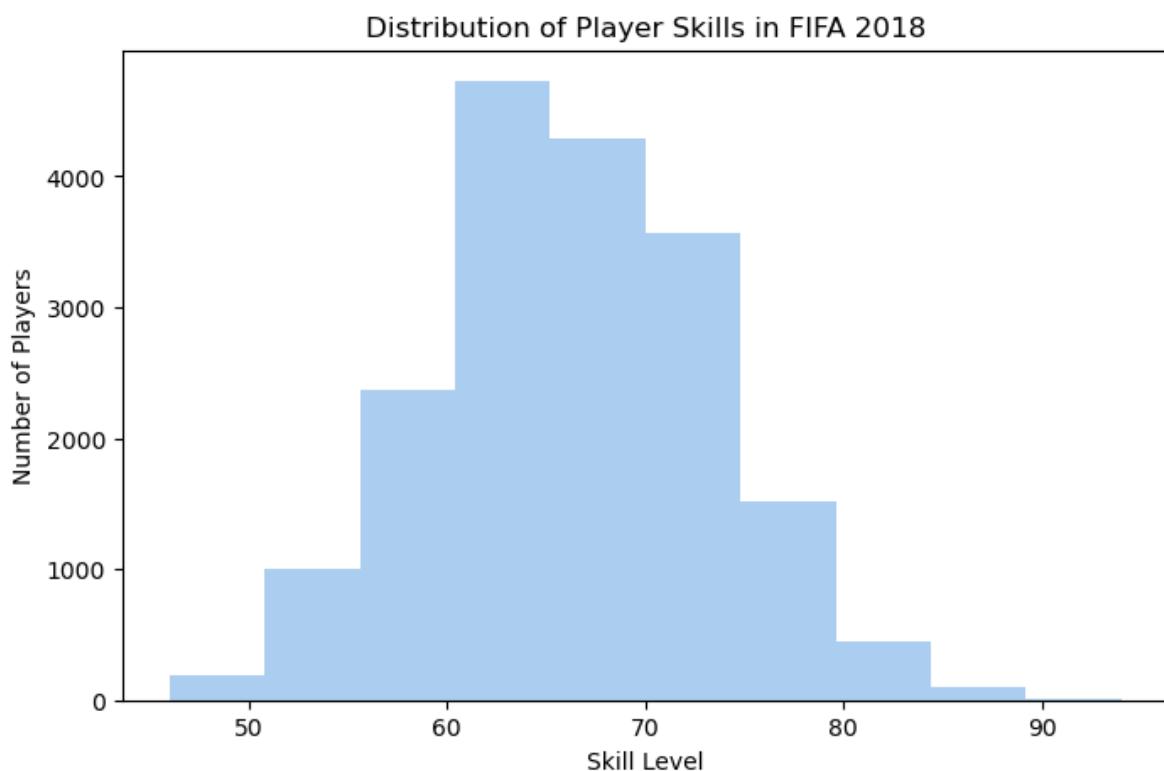
plt.figure(figsize=(8,5))

plt.hist(fifa.Overall, color='#abcdef')

plt.ylabel('Number of Players')
plt.xlabel('Skill Level')
plt.title('Distribution of Player Skills in FIFA 2018')

```

Text(0.5, 1.0, 'Distribution of Player Skills in FIFA 2018')



#### 10.2.2.5.2 Bar chart

```

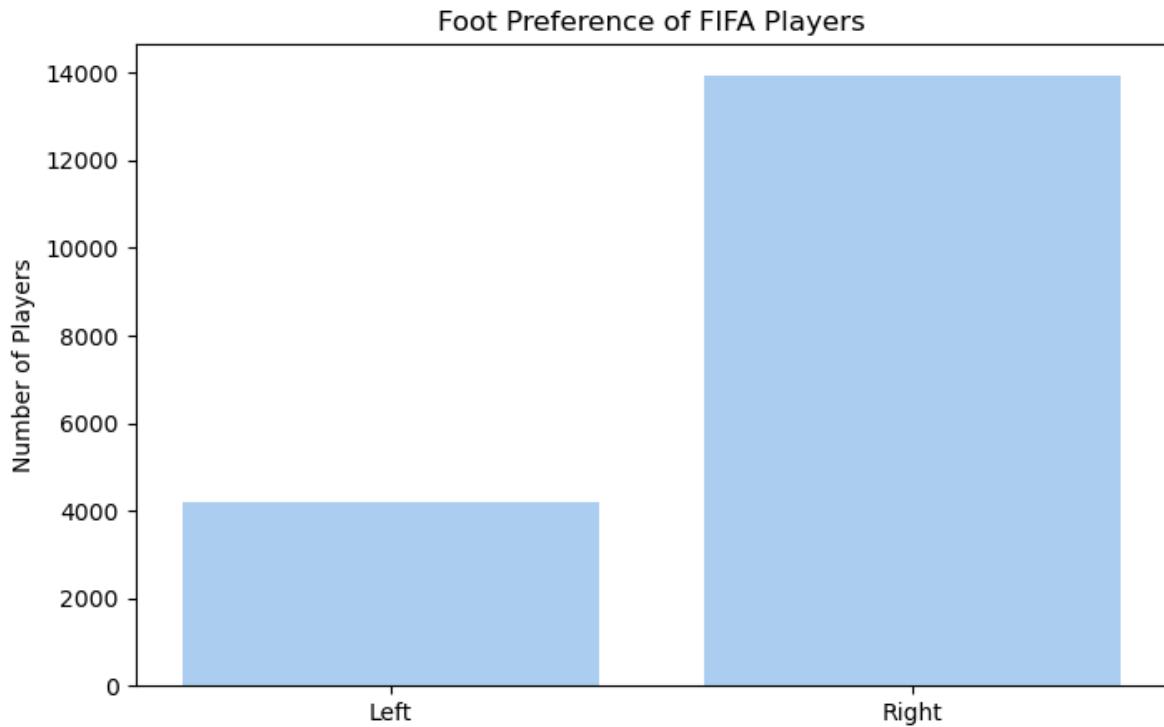
# plotting bar chart for the best players
plt.figure(figsize=(8,5))

foot_preference = fifa['Preferred Foot'].value_counts()

```

```
plt.bar(['Left', 'Right'], [foot_preference.iloc[1], foot_preference.iloc[0]], color='#abcdef')

plt.ylabel('Number of Players')
plt.title('Foot Preference of FIFA Players');
```



#### 10.2.2.5.3 Pie chart

```
left = fifa.loc[fifa['Preferred Foot'] == 'Left'].count().iloc[0]
right = fifa.loc[fifa['Preferred Foot'] == 'Right'].count().iloc[0]

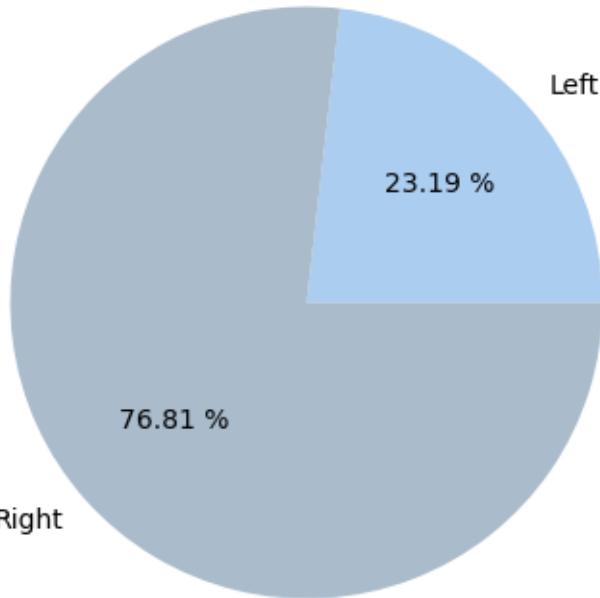
plt.figure(figsize=(8,5))

labels = ['Left', 'Right']
colors = ['#abcdef', '#aabbcc']

plt.pie([left, right], labels = labels, colors=colors, autopct='%.2f %%')

plt.title('Foot Preference of FIFA Players');
```

Foot Preference of FIFA Players



Another Pie Chart on wight of players

```
plt.figure(figsize=(8,5), dpi=100)

plt.style.use('ggplot')

fifa.Weight = [int(x.strip('lbs')) if type(x)==str else x for x in fifa.Weight]

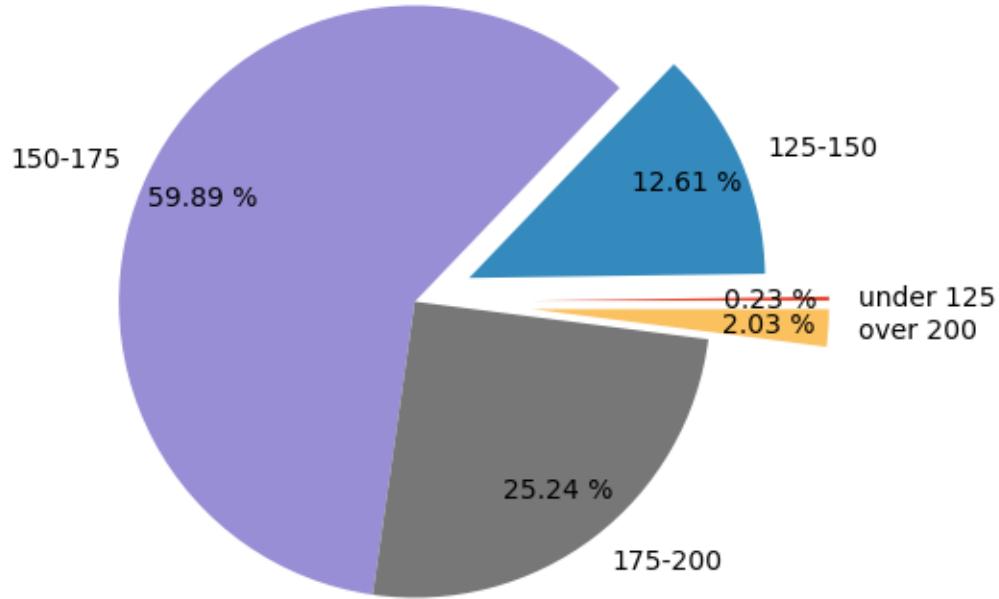
light = fifa.loc[fifa.Weight < 125].count().iloc[0]
light_medium = fifa[(fifa.Weight >= 125) & (fifa.Weight < 150)].count().iloc[0]
medium = fifa[(fifa.Weight >= 150) & (fifa.Weight < 175)].count().iloc[0]
medium_heavy = fifa[(fifa.Weight >= 175) & (fifa.Weight < 200)].count().iloc[0]
heavy = fifa[fifa.Weight >= 200].count().iloc[0]

weights = [light,light_medium, medium, medium_heavy, heavy]
label = ['under 125', '125-150', '150-175', '175-200', 'over 200']
explode = (.4,.2,0,0,.4)

plt.title('Weight of Professional Soccer Players (lbs)')
```

```
plt.pie(weights, labels=label, explode=explode, pctdistance=0.8, autopct='%.2f %%');
```

Weight of Professional Soccer Players (lbs)



#### 10.2.2.5.4 Box and Whiskers Chart

```
plt.figure(figsize=(5,8), dpi=100)

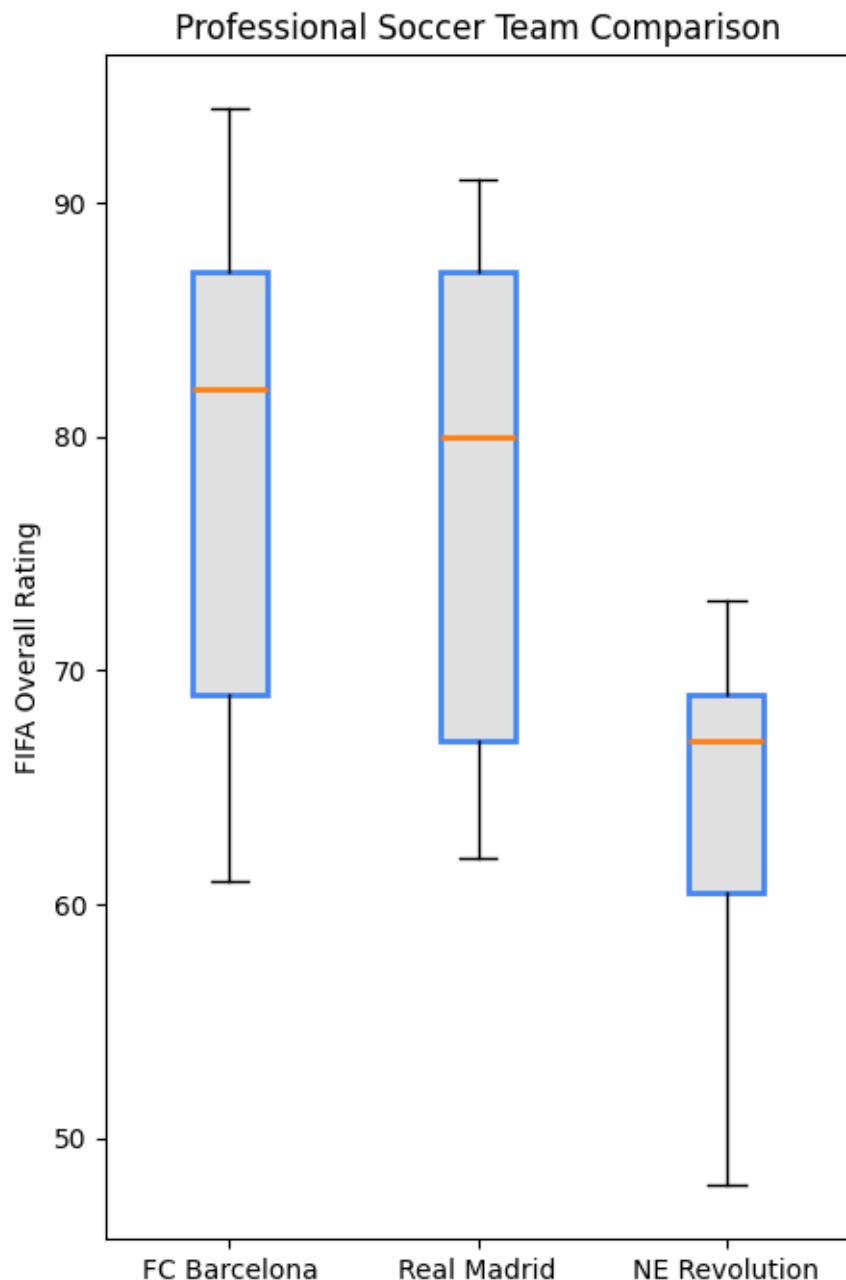
plt.style.use('default')

barcelona = fifa.loc[fifa.Club == "FC Barcelona"]['Overall']
madrid = fifa.loc[fifa.Club == "Real Madrid"]['Overall']
revs = fifa.loc[fifa.Club == "New England Revolution"]['Overall']

#bp = plt.boxplot([barcelona, madrid, revs], labels=['a','b','c'], boxprops=dict(facecolor='red'))
bp = plt.boxplot([barcelona, madrid, revs], tick_labels=['FC Barcelona', 'Real Madrid', 'NE Rev'])

plt.title('Professional Soccer Team Comparison')
plt.ylabel('FIFA Overall Rating')
```

```
for box in bp['boxes']:
    # change outline color
    box.set(color='#4286f4', linewidth=2)
    # change fill color
    box.set(facecolor = '#e0e0e0' )
    # change hatch
    #box.set(hatch = '/')
```



You've already learned how to create plots using Matplotlib's Pyplot. Next, you'll explore how to simplify and enhance your visualizations by using its powerful wrappers: Pandas and Seaborn.

#### 10.2.2.6 Limitations of Using Matplotlib for Plotting

Matplotlib is a popular visualization library, but it has flaws.

- Defaults are not ideal (gridlines, background, etc need to be configured.)
- Library is low-level (doing anything complicated takes quite a bit of codes)
- Lack of integration with pandas data structures

To address these challenges, Seaborn act as higher-level interfaces to Matplotlib, offering better defaults, simpler syntax, and seamless integration with DataFrames.

#### 10.2.3 Plotting with Seaborn

Seaborn is a powerful data visualization library built on top of Matplotlib, designed to make statistical plots easier and more attractive. It integrates seamlessly with Pandas, making it an excellent tool for plotting data from DataFrames. Seaborn comes with better default aesthetics and provides more specialized plots that are easy to implement.

Some of the key advantages of using Seaborn include:

- **Simpler syntax:** Seaborn simplifies the process of creating complex plots with just a few lines of code.
- **Beautiful default styles:** Seaborn's default plots are more polished and aesthetically pleasing compared to Matplotlib's defaults.
- **Seamless Pandas integration:** You can directly pass Pandas DataFrames to Seaborn, and it understands column names for axis labels and plot elements.

`seaborn` comes with 17 built-in datasets. That means you don't have to spend a whole lot of your time finding the right dataset and cleaning it up to make Seaborn-ready; rather you will focus on the core features of Seaborn visualization techniques to solve problems.

```
import seaborn as sns
# get names of the builtin dataset
sns.get_dataset_names()
```

```
['anagrams',
 'anscombe',
 'attention',
 'brain_networks',
 'car_crashes',
 'diamonds',
 'dots',
 'dowjones',
```

```
'exercise',
'flights',
'fmri',
'geyser',
'glue',
'healthexp',
'iris',
'mpg',
'penguins',
'planets',
'seaice',
'taxis',
'tips',
'titanic']
```

### 10.2.3.1 Customizing Plot Aesthetics

Seaborn provides a convenient function called `sns.set_style()` that allows users to customize the visual appearance of their plots. This function is particularly useful for enhancing the aesthetics of visualizations, making them more appealing and easier to interpret.

#### 10.2.3.1.1 Purpose of `sns.set_style()`

The primary purpose of `sns.set_style()` is to set the visual context and style for all plots created after the call. This allows for a consistent and visually pleasing representation of data across multiple visualizations.

#### 10.2.3.1.2 Available Style Options

Seaborn offers several built-in styles that can be set using `sns.set_style()`. The options include:

1. `darkgrid`:

- A dark background with a grid overlay.
- Ideal for visualizing data with many points or intricate details.

2. `whitegrid`:

- A white background with a grid overlay.
- Provides a clean and professional look, suitable for most types of visualizations.

3. `dark`:

- A dark background without gridlines.
- Useful for emphasizing data points without distractions from the grid.

4. **white:**

- A simple white background without gridlines.
- Offers a minimalist aesthetic, focusing solely on the data.

5. **ticks:**

- A white background with ticks on the axes.
- Combines the clarity of a white background with a bit of added detail for reference.

#### 10.2.3.1.3 How to Use `sns.set_style()`

To apply a specific style, simply call `sns.set_style()` with the desired style name before creating your plots. Here's an example:

```
sns.set_style("whitegrid")
```

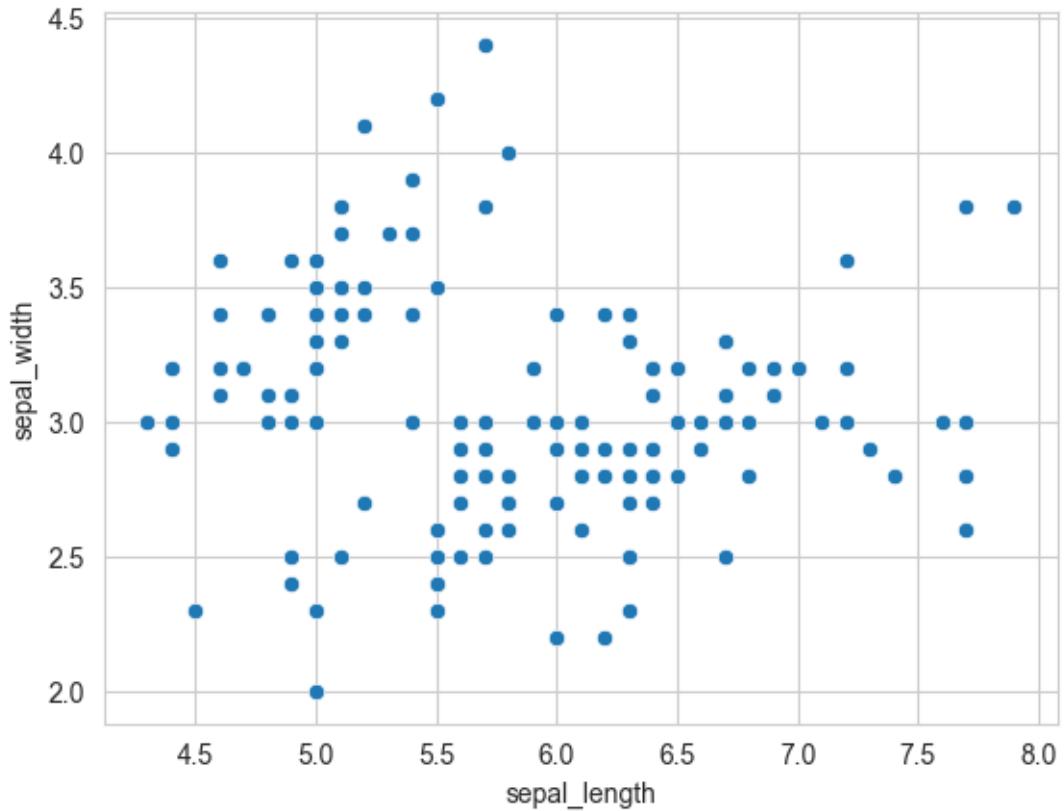
Let's do a few common plots with Seaborn tips dataset

```
# Load data into a Pandas dataframe
flowers_df = sns.load_dataset("iris")
flowers_df.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

#### 10.2.3.2 Scatterplot

```
sns.scatterplot(x=flowers_df.sepal_length, y=flowers_df.sepal_width);
```



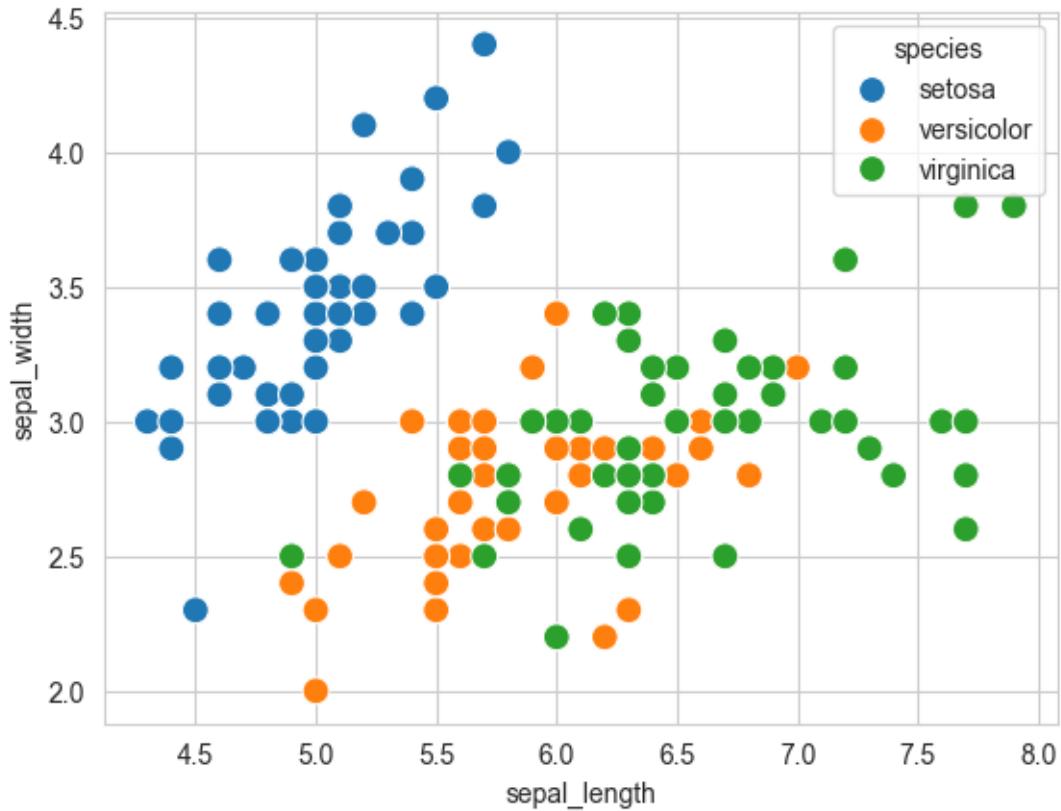
#### 10.2.3.2.1 Adding Hues to the scatterplot

Notice how the points in the above plot seem to form distinct clusters with some outliers. We can color the dots using the flower species as a hue. We can also make the points larger using the `s` argument.

```
flowers_df.species.unique()
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

```
sns.scatterplot(x=flowers_df.sepal_length, y=flowers_df.sepal_width, hue=flowers_df.species,
```



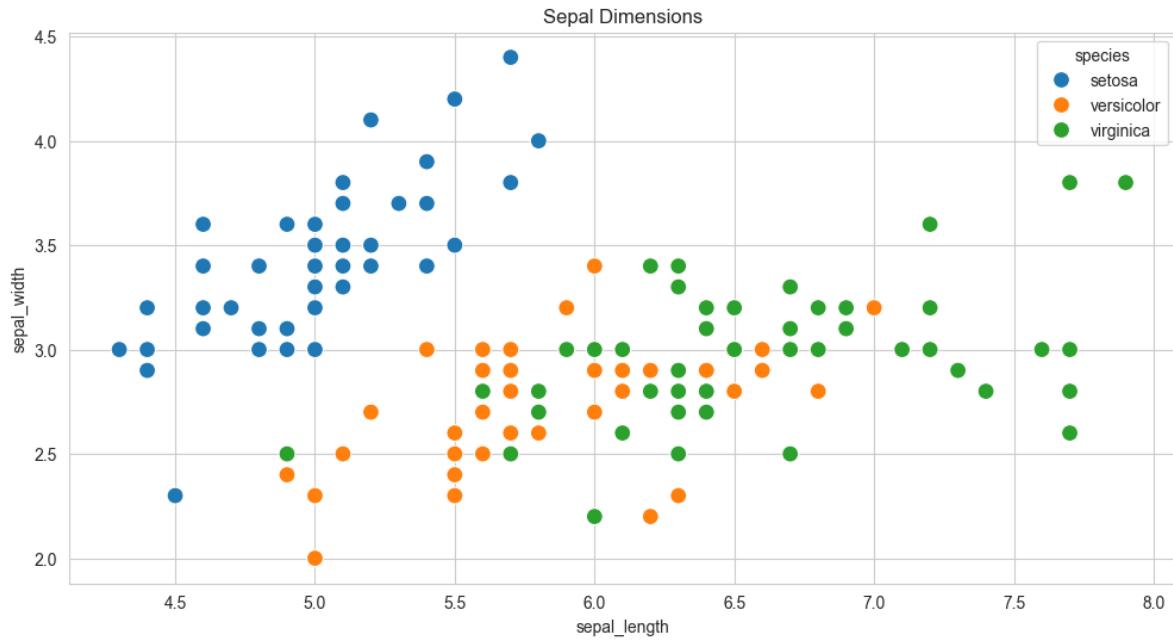
Adding hues makes the plot more informative. We can immediately tell that Setosa flowers have a smaller sepal length but higher sepal widths. In contrast, the opposite is true for Virginica flowers.

#### 10.2.3.2.2 Customizing Seaborn Figures

Since Seaborn uses Matplotlib's plotting functions internally, we can use functions like `plt.figure` and `plt.title` to modify the figure.

```
plt.figure(figsize=(12, 6))
plt.title('Sepal Dimensions')

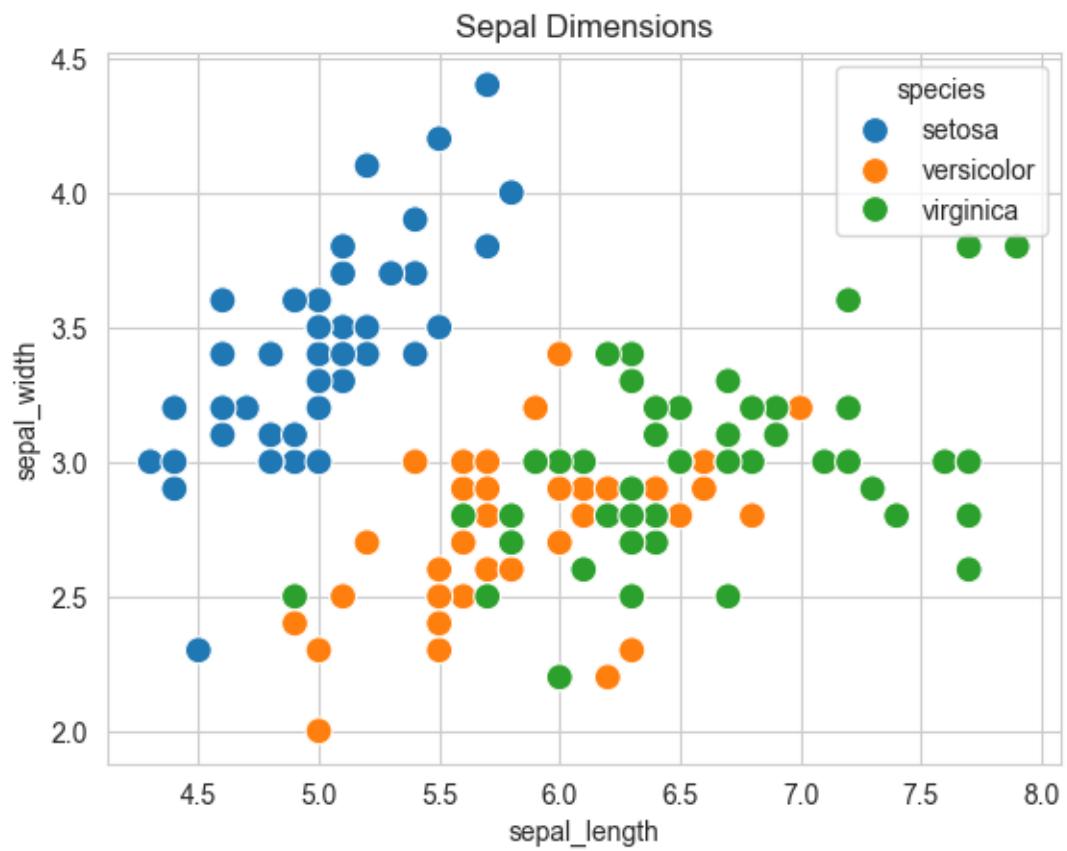
sns.scatterplot(x=flowers_df.sepal_length,
                 y=flowers_df.sepal_width,
                 hue=flowers_df.species,
                 s=100);
```



#### 10.2.3.2.3 Integration with Pandas Data Frames

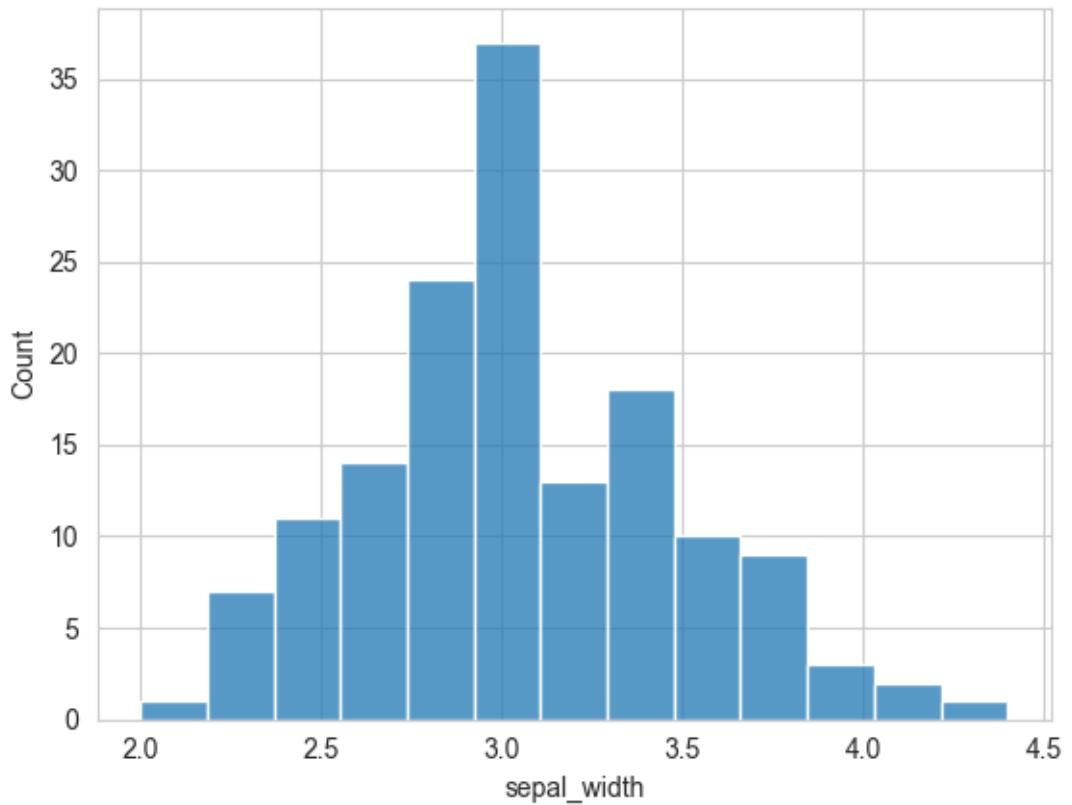
Seaborn has in-built support for Pandas data frames. Instead of passing each column as a series, you can provide column names and use the `data` argument to specify a data frame.

```
plt.title('Sepal Dimensions')
sns.scatterplot(x='sepal_length',
                 y='sepal_width',
                 hue='species',
                 s=100,
                 data=flowers_df);
```

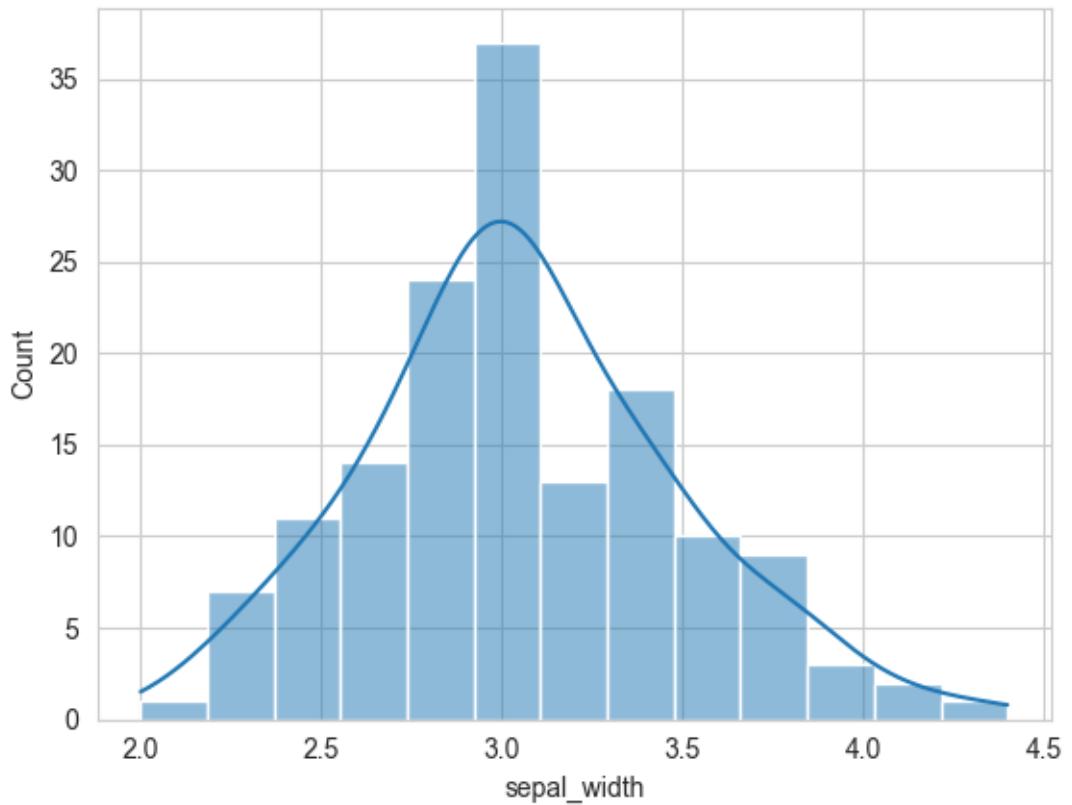


#### 10.2.3.3 Histogram

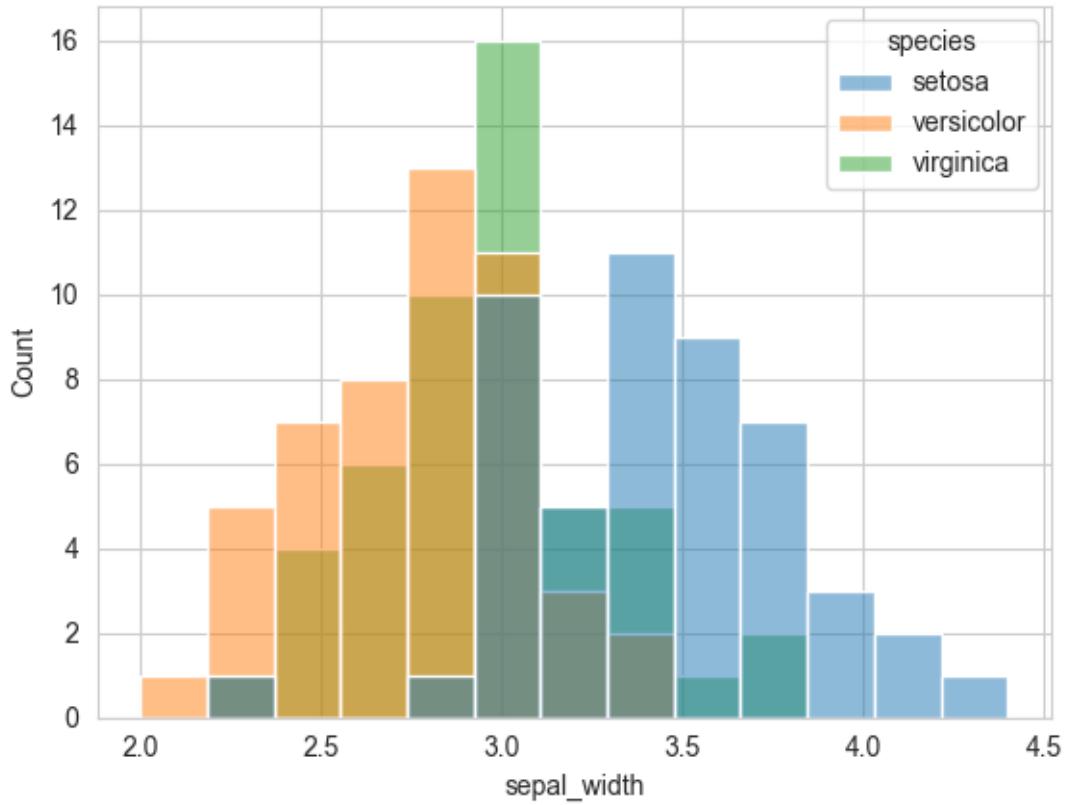
```
sns.histplot(data=flowers_df, x='sepal_width');
```



```
# show kde(kernel density estimate)
sns.histplot(data=flowers_df, x='sepal_width', kde=True);
```

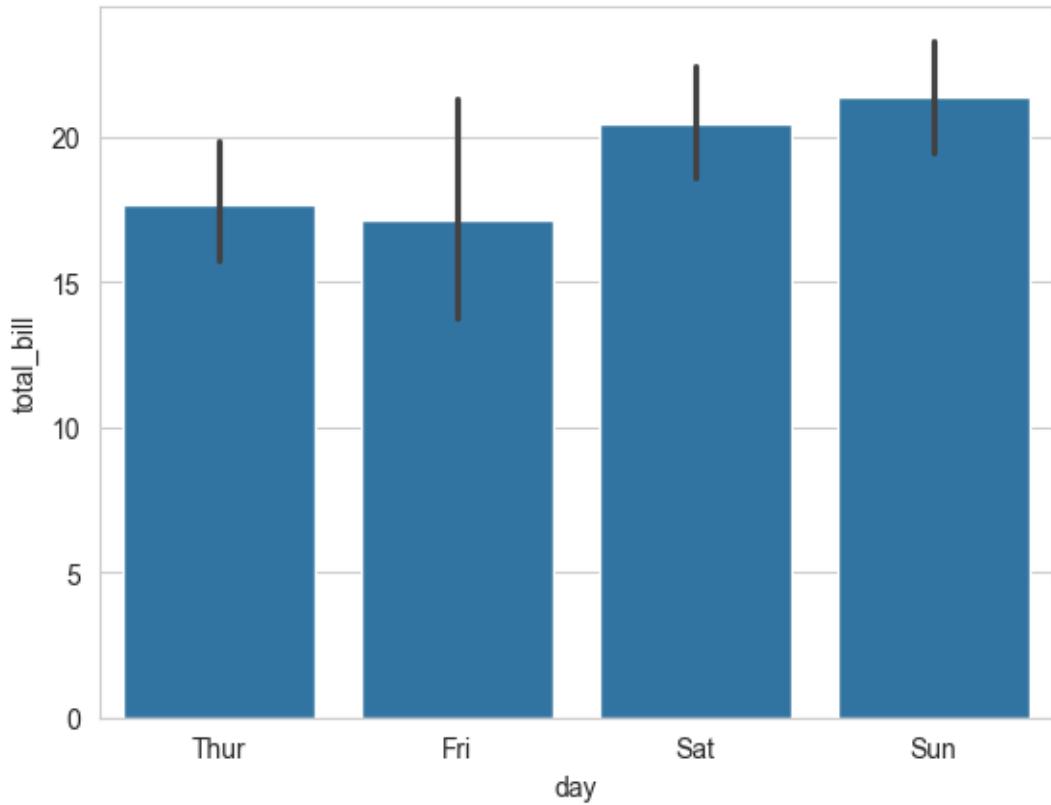


```
# adding hue
sns.histplot(data=flowers_df, x="sepal_width", hue="species");
```

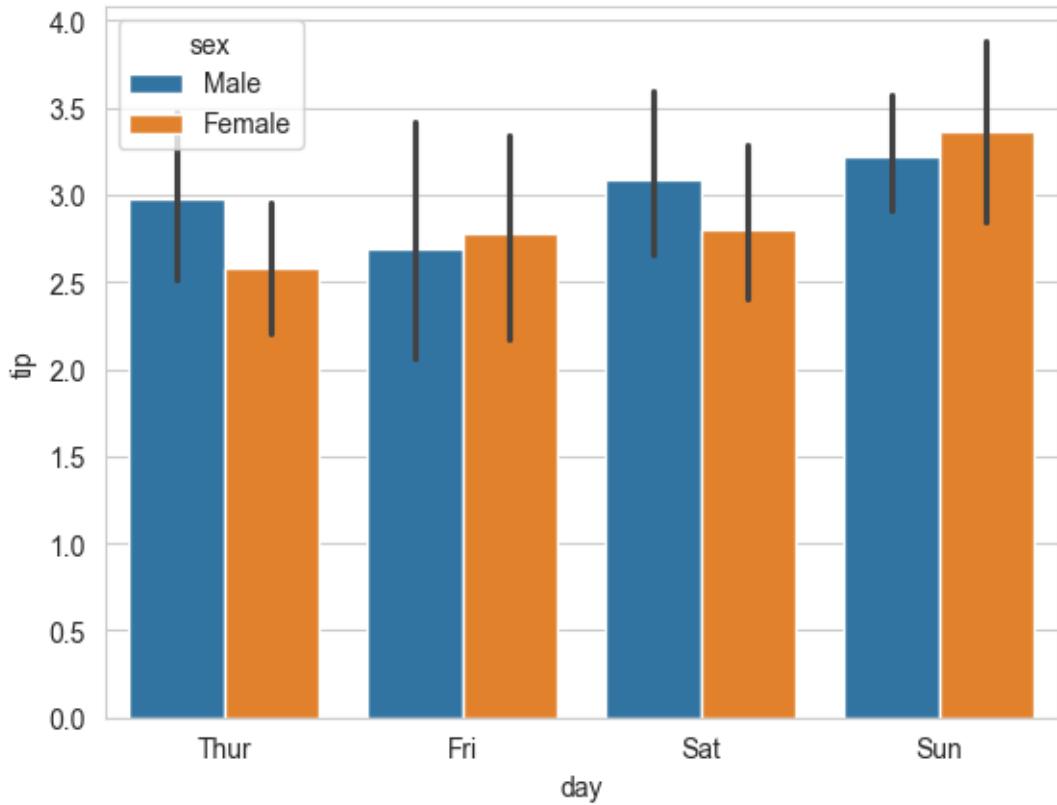


#### 10.2.3.4 Barplot

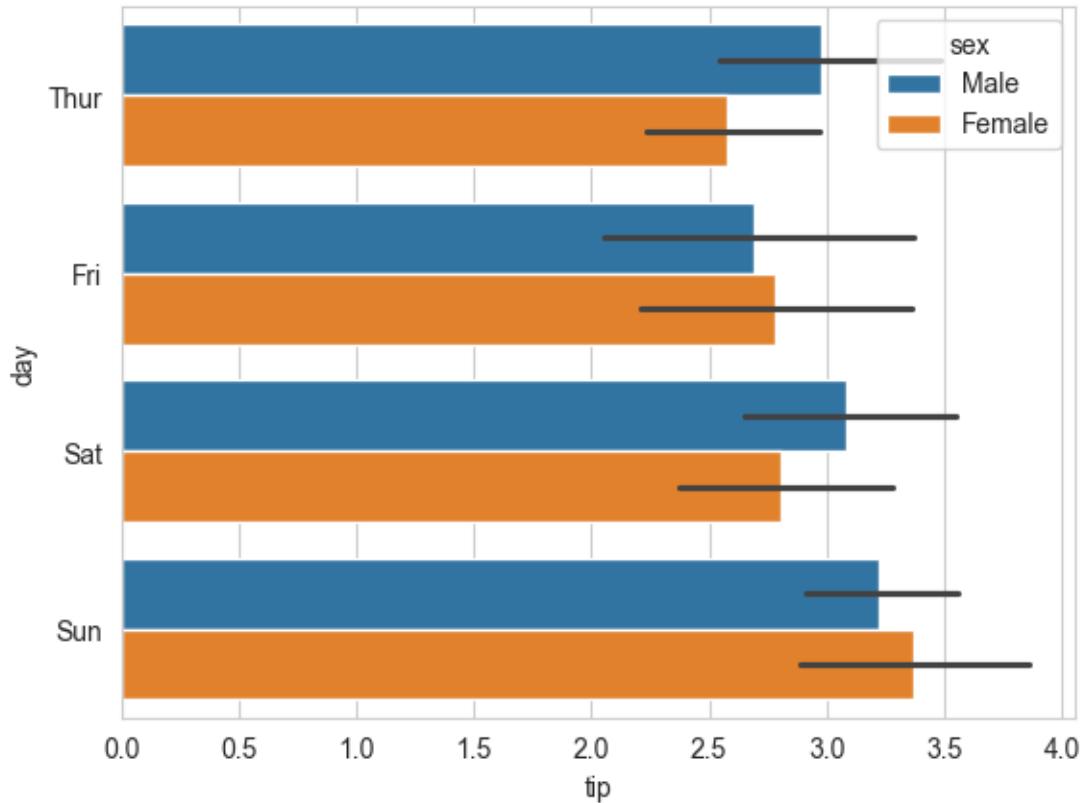
```
tips_df = sns.load_dataset("tips")
sns.barplot(x='day', y='total_bill', data=tips_df);
```



```
sns.barplot(x='day', y='tip', hue='sex', data=tips_df);
```



```
# make the bars horizontal simply by switching the axes
sns.barplot(x='tip', y='day', hue='sex', data=tips_df);
```



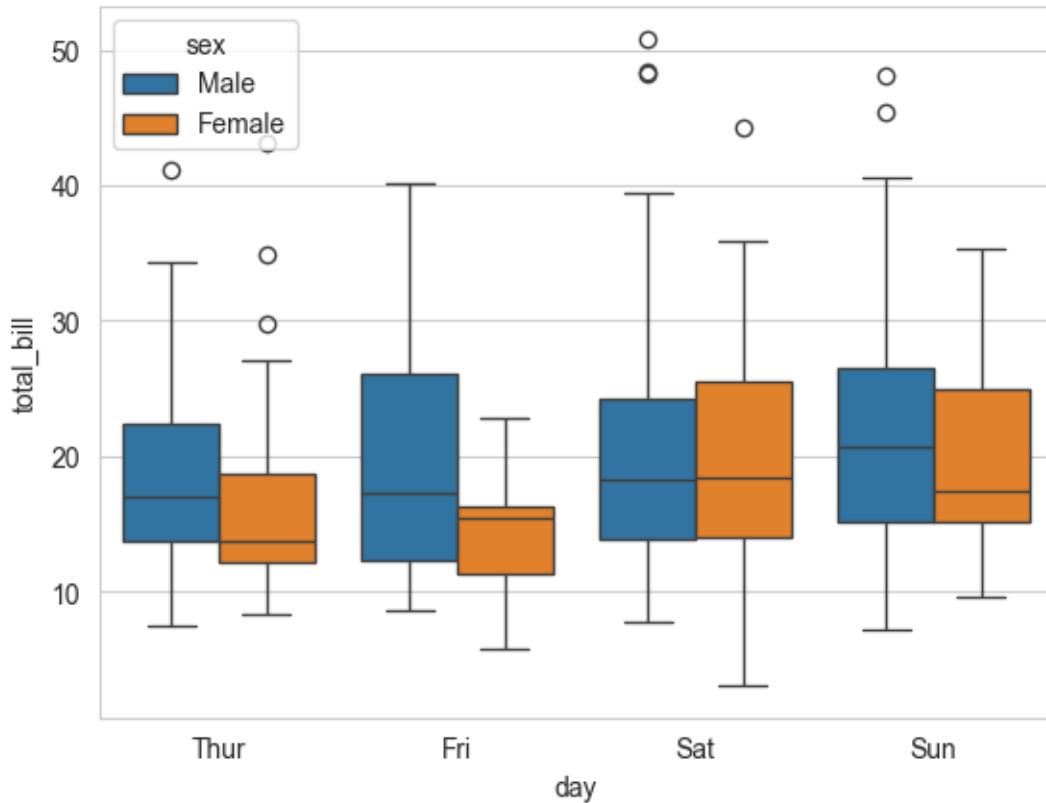
#### 10.2.3.5 Boxplot

**Purpose:** Boxplots is a standardized way of visualizing the distribution of a continuous variable. They show five key metrics that describe the data distribution - median, 25th percentile value, 75th percentile value, minimum and maximum, as shown in the figure below. Note that the minimum and maximum exclude the outliers.

<IPython.core.display.Image object>

**Example:** Create a box plot to compare the distributions of total tips based on the day of the week, differentiating between male and female patrons.

```
sns.boxplot(data=tips_df, y='total_bill', x='day', hue='sex');
```



From the above plot, what you can observe?

#### 10.2.3.6 Heatmap

Represent 2-dimensional data like a matrix or table using colors

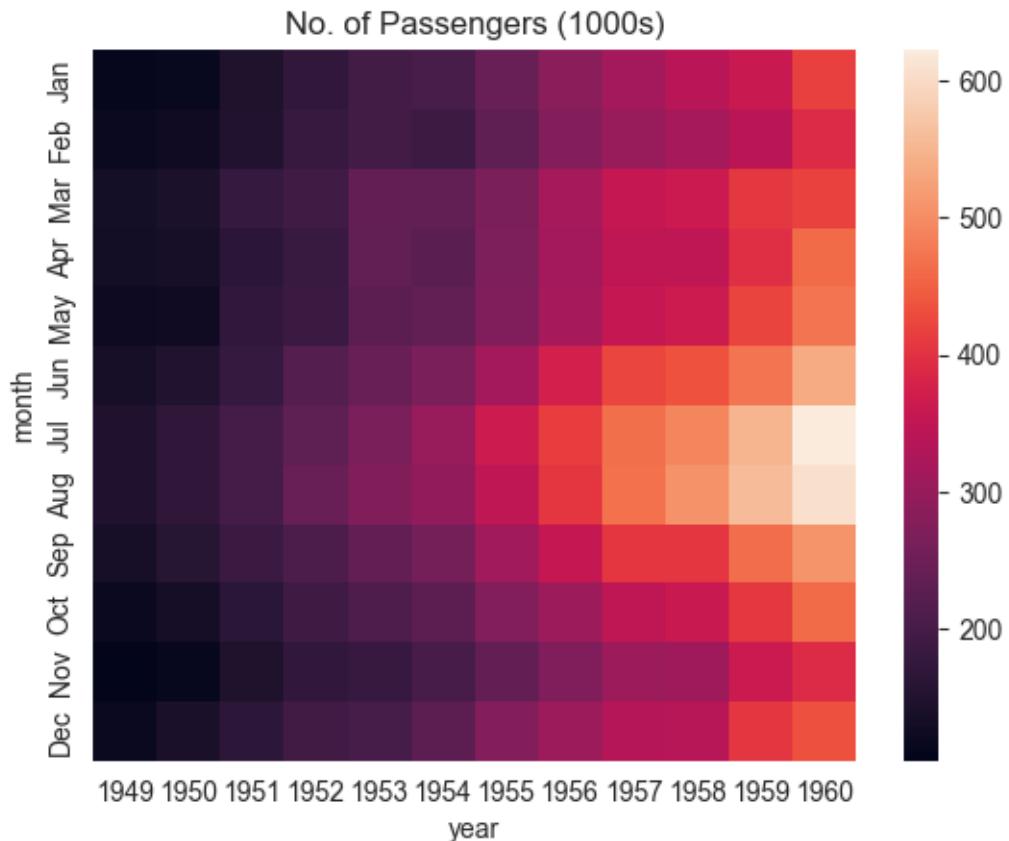
```
flights_df = sns.load_dataset("flights").pivot(index="month", columns="year", values="passenger")
flights_df
# you will learn pivot in the later chapters
```

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
Jan	112	115	145	171	196	204	242	284	315	340	360	417
Feb	118	126	150	180	196	188	233	277	301	318	342	391
Mar	132	141	178	193	236	235	267	317	356	362	406	419
Apr	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
Jun	135	149	178	218	243	264	315	374	422	435	472	535
Jul	148	170	199	230	264	302	364	413	465	491	548	622
Aug	148	170	199	242	272	293	347	405	467	505	559	606
Sep	136	158	184	209	237	259	312	355	404	404	463	508
Oct	119	133	162	191	211	229	274	306	347	359	407	461
Nov	104	114	146	172	180	203	237	271	305	310	362	390
Dec	118	140	166	194	201	229	278	306	336	337	405	432

`flights_df` is a matrix with one row for each month and one column for each year. The values show the number of passengers (in thousands) that visited the airport in a specific month of a year. We can use the `sns.heatmap` function to visualize the footfall at the airport.

```
plt.title("No. of Passengers (1000s)")
sns.heatmap(flights_df);
```

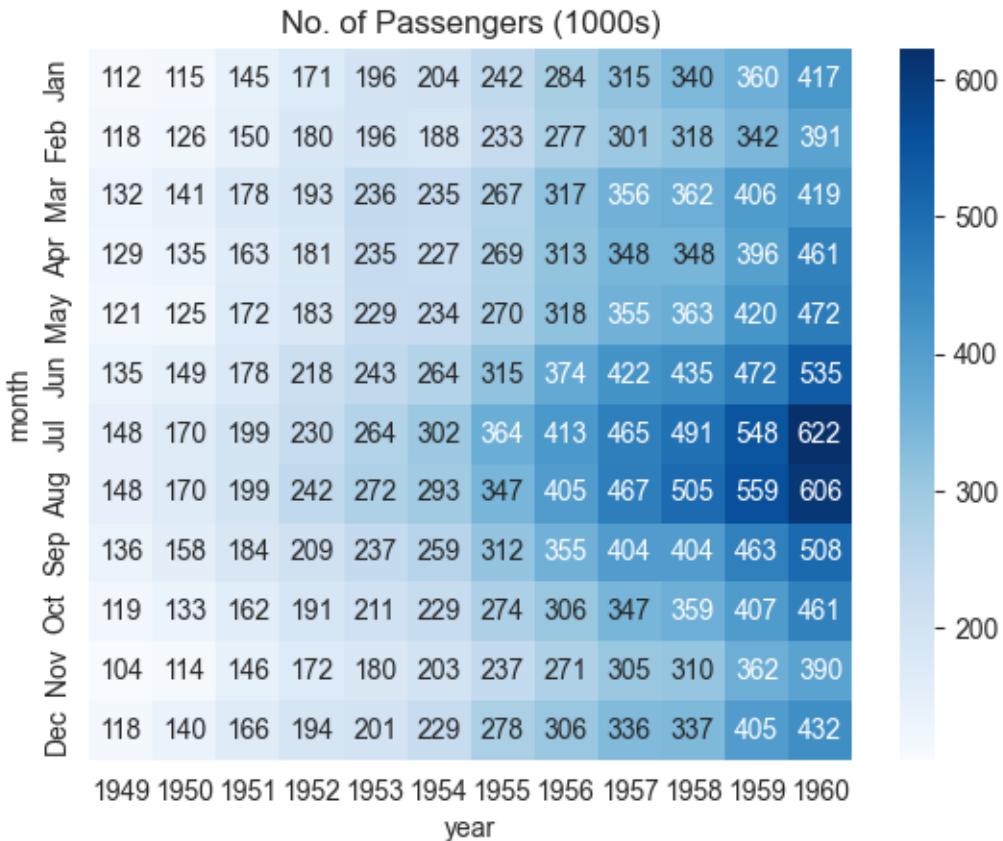


The brighter colors indicate a higher footfall at the airport. By looking at the graph, we can infer two things:

- The footfall at the airport in any given year tends to be the highest around July & August.
- The footfall at the airport in any given month tends to grow year by year.

We can also display the actual values in each block by specifying `annot=True` and using the `cmap` argument to change the color palette.

```
# fmt = "d" decimal integer. output are the number in base 10
plt.title("No. of Passengers (1000s)")
sns.heatmap(flights_df, fmt="d", annot=True, cmap='Blues');
```



#### 10.2.3.7 Correlation Map

A correlation map is a specific type of heatmap where the values represent the correlation coefficients between variables (ranging from -1 to 1). It visually shows the strength and

direction of relationships between numerical variables.

Correlation may refer to any kind of association between two random variables. However, in this book, we will always consider correlation as the linear association between two random variables, or the Pearson's correlation coefficient. Note that correlation does not imply causality and vice-versa.

The Pandas function `corr()` provides the pairwise correlation between all columns of a DataFrame, or between two Series. The function `corrwith()` provides the pairwise correlation of a DataFrame with another DataFrame or Series.

```
#Pairwise correlation amongst all columns
survey_data = pd.read_csv('./Datasets/survey_data_clean.csv')

survey_data.head()
```

	Timestamp	fav_alcohol	parties_per_month	smoke	weed
0	2022/09/13 1:43:34 pm GMT-5	I don't drink	1.0	No	Occasionally
1	2022/09/13 5:28:17 pm GMT-5	Hard liquor/Mixed drink	3.0	No	Occasionally
2	2022/09/13 7:56:38 pm GMT-5	Hard liquor/Mixed drink	3.0	No	Yes
3	2022/09/13 10:34:37 pm GMT-5	Hard liquor/Mixed drink	12.0	No	No
4	2022/09/14 4:46:19 pm GMT-5	I don't drink	1.0	No	No

```
#Pairwise correlation amongst all columns
survey_data.select_dtypes(include='number').corr()
```

	parties_per_month	love_first_sight	num_insta_followers	expected_n
parties_per_month	1.000000	0.096129	0.239705	-0.064079
love_first_sight	0.096129	1.000000	-0.024010	-0.084406
num_insta_followers	0.239705	-0.024010	1.000000	-0.130157
expected_marriage_age	-0.064079	-0.084406	-0.130157	1.000000
expected_starting_salary	0.114881	0.080138	0.127226	-0.014881
minutes_ex_per_week	0.195561	0.099244	0.099341	-0.088073
sleep_hours_per_day	-0.052542	-0.025378	-0.042421	0.182009
farthest_distance_travelled	-0.017081	-0.075539	0.011308	-0.024038
fav_number	-0.050139	0.105095	-0.124763	-0.008924
internet_hours_per_day	0.087390	-0.007652	-0.028427	-0.029772
only_child	-0.142519	0.124345	-0.152184	-0.043141
num_majors_minors	-0.073127	0.108730	0.050431	-0.055280
high_school_GPA	0.295646	0.069288	0.147402	0.017052

	parties_per_month	love_first_sight	num_insta_followers	expected_n
NU_GPA	-0.080548	-0.114041	0.004702	0.011925
age	-0.032771	0.142384	-0.230698	0.060416
height	-0.005405	0.216072	0.009318	0.044577
height_father	0.126741	0.029419	0.179684	0.026949
height_mother	0.079121	0.082684	0.129716	0.075316
procrastinator	-0.056871	0.033951	-0.089871	-0.020012
num_clubs	-0.010514	0.083342	0.265958	-0.137069
student_athlete	0.290830	0.014595	0.044807	-0.036122
AP_stats	-0.013222	-0.062992	0.005947	0.010447
used_python_before	-0.040033	-0.034692	-0.016201	0.052727
childhood_in_US	0.081905	-0.118260	0.072622	0.053759
cant_change_math_ability	-0.052912	0.005254	-0.150658	-0.072163
can_change_math_ability	0.055575	0.020758	0.130774	0.087633
math_is_genetic	-0.013374	-0.003710	-0.018411	-0.086898
much_effort_is_lack_of_talent	-0.029838	0.013376	-0.165899	0.052813

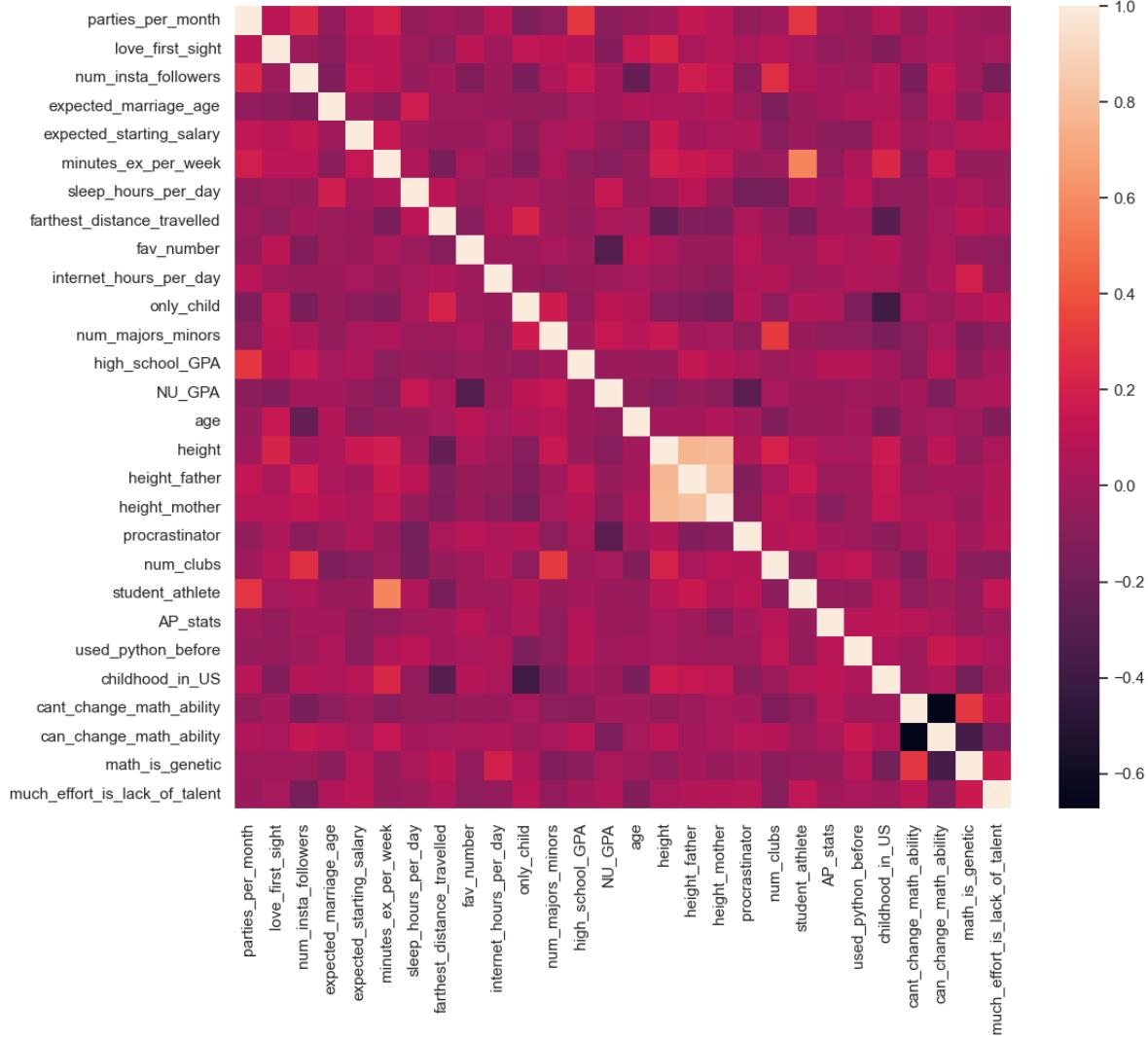
**Q:** Which feature is the most correlated with *NU\_GPA*?

```
survey_data.select_dtypes(include='number').corrwith(survey_data.NU_GPA).sort_values(ascending=False)
```

NU_GPA	1.000000
sleep_hours_per_day	0.143997
num_majors_minors	0.141988
only_child	0.106440
much_effort_is_lack_of_talent	0.047840
farthest_distance_travelled	0.038238
math_is_genetic	0.036731
num_clubs	0.016724
expected_marriage_age	0.011925
num_insta_followers	0.004702
cant_change_math_ability	0.002094
used_python_before	-0.008536
internet_hours_per_day	-0.014531
AP_stats	-0.026544
student_athlete	-0.027378
childhood_in_US	-0.028968
high_school_GPA	-0.030883
height_father	-0.040120
expected_starting_salary	-0.048069
age	-0.052039

```
height_mother           -0.079276
parties_per_month       -0.080548
height                  -0.099082
minutes_ex_per_week     -0.108177
love_first_sight        -0.114041
can_change_math_ability -0.137330
procrastinator          -0.269552
fav_number              -0.307656
dtype: float64
```

```
sns.set(rc={'figure.figsize':(12,10)})
sns.heatmap(survey_data.select_dtypes(include='number').corr());
```



From the above map, we can see that:

- `student_athlete` is strongly positively correlated with `minutes_ex_per_week`
- `procrastinator` is strongly negatively correlated with `NU_GPA`

## 10.3 Independent Study

### 10.3.1 Practice exercise 1

Read the `gas_price.csv` file and plot the trend for each country over the time using matplotlib `pyplot`

## 10.3.2 Practice exercise 2

### 10.3.2.1

Is NU\_GPA associated with parties\_per\_month? Analyze the association separately for *Sophomores, Juniors, and Seniors* (categories of the variable school\_year).

Make scatterplots of NU\_GPA vs parties\_per\_month in a 1 x 3 grid, where each grid is for a distinct school\_year. Plot the trendline as well for each scatterplot. Use the file *survey\_data\_clean.csv*.

### 10.3.2.2

Capping the the values of parties\_per\_month to 30, and make the visualizations again.

## 10.3.3 Practice exercise 3

How does the expected marriage age of the people of STAT303-1 depend on their characteristics? We'll use visualizations to answer this question. Use data from the file *survey\_data\_clean.csv*. Proceed as follows:

### 10.3.3.1

Make a visualization that compares the mean expected\_marriage\_age of introverts and extroverts (*use the variable introvert\_extrovert*). What insights do you obtain?

### 10.3.3.2

Does the mean expected\_marriage\_age of introverts and extroverts depend on whether they believe in love in first sight (*variable name: love\_first\_sight*)? Update the previous visualization to answer the question.

### 10.3.3.3

In addition to *love\_first\_sight*, does the mean expected\_marriage\_age of introverts and extroverts depend on whether they are a procrastinator (*variable name: procrastinator*)? Update the previous visualization to answer the question.

#### **10.3.3.4**

Is there any critical information missing in the above visualizations that, if revealed, may cast doubts on the patterns observed in them?

#### **10.3.4 Practice exercise 4**

Read *Australia\_weather.csv*,

##### **10.3.4.1**

Create a histogram showing the distributions of maximum temperature in Sydney, Canberra and Melbourne.

##### **10.3.4.2**

Make a density plot showing the distributions of maximum temperature in Sydney, Canberra and Melbourne.

##### **10.3.4.3**

Show the distributions of the maximum and minimum temperatures in a single plot.

##### **10.3.4.4**

Create a scatter plot with a trendline for `MinTemp` and `MaxTemp`, including a confidence interval.

**Hint:** Using Seaborn, the `regplot()` function enables us to overlay a trendline on the scatter plot, complete with a 95% confidence interval for the trendline

# 11 Advanced Data Visualization

In the previous chapter, we explored basic plotting techniques using Pandas, Seaborn, and Matplotlib to create visualizations. Now, we'll elevate our skills by diving into more advanced topics, such as crafting complex subplots and visualizing geospatial data, enabling us to build richer and more insightful plots.

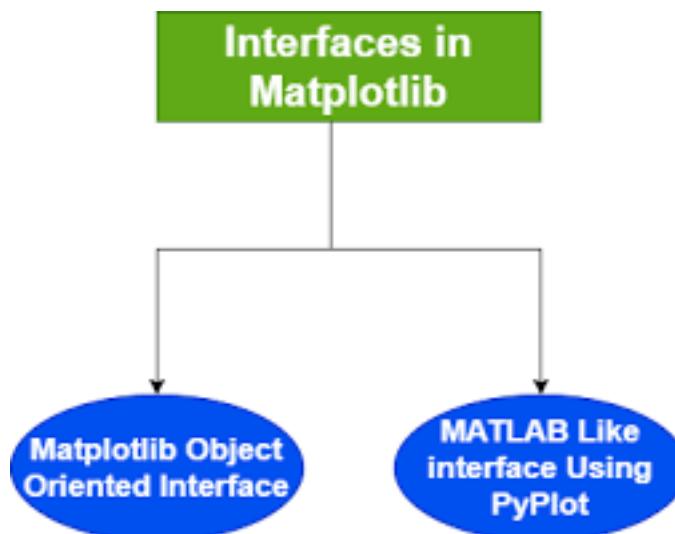
To get started, let's import necessary libraries.

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
%matplotlib inline
```

## 11.1 Matplotlib Plotting Interfaces

### 11.1.1 Pyplot interface and OOP Interface

There are two types of interfaces in Matplotlib for visualization and these are given below:

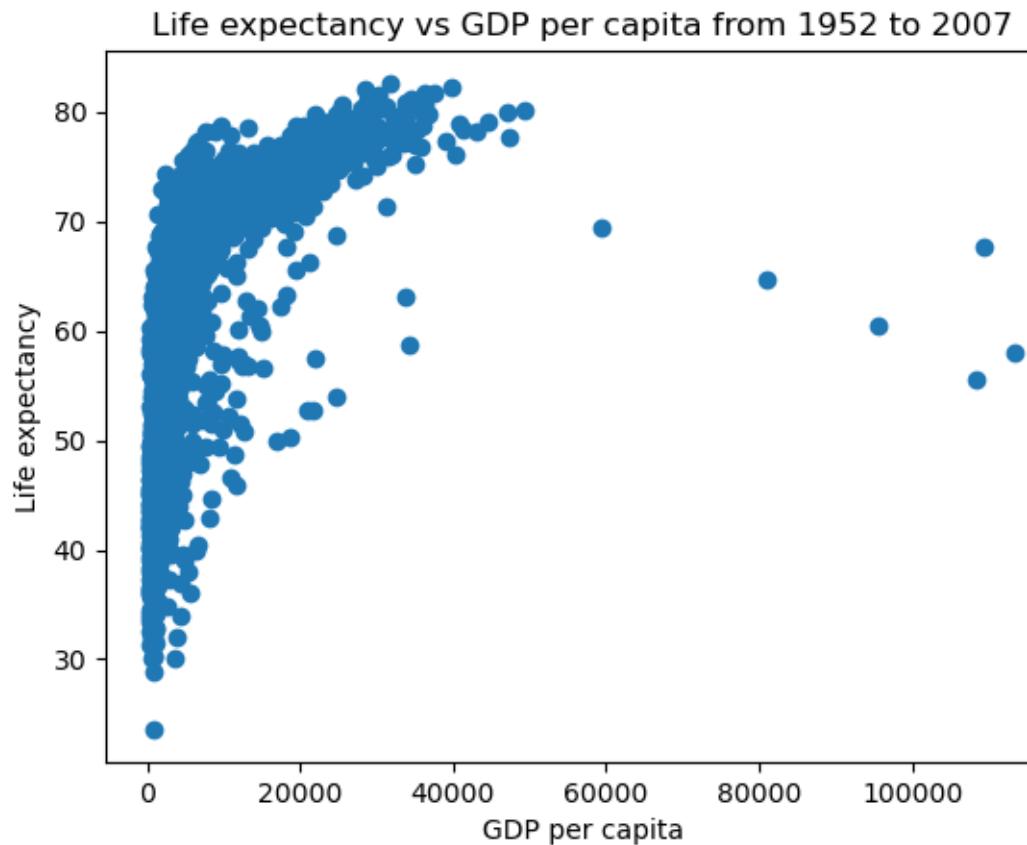


### 11.1.2 Plot a simple figure using two interfaces

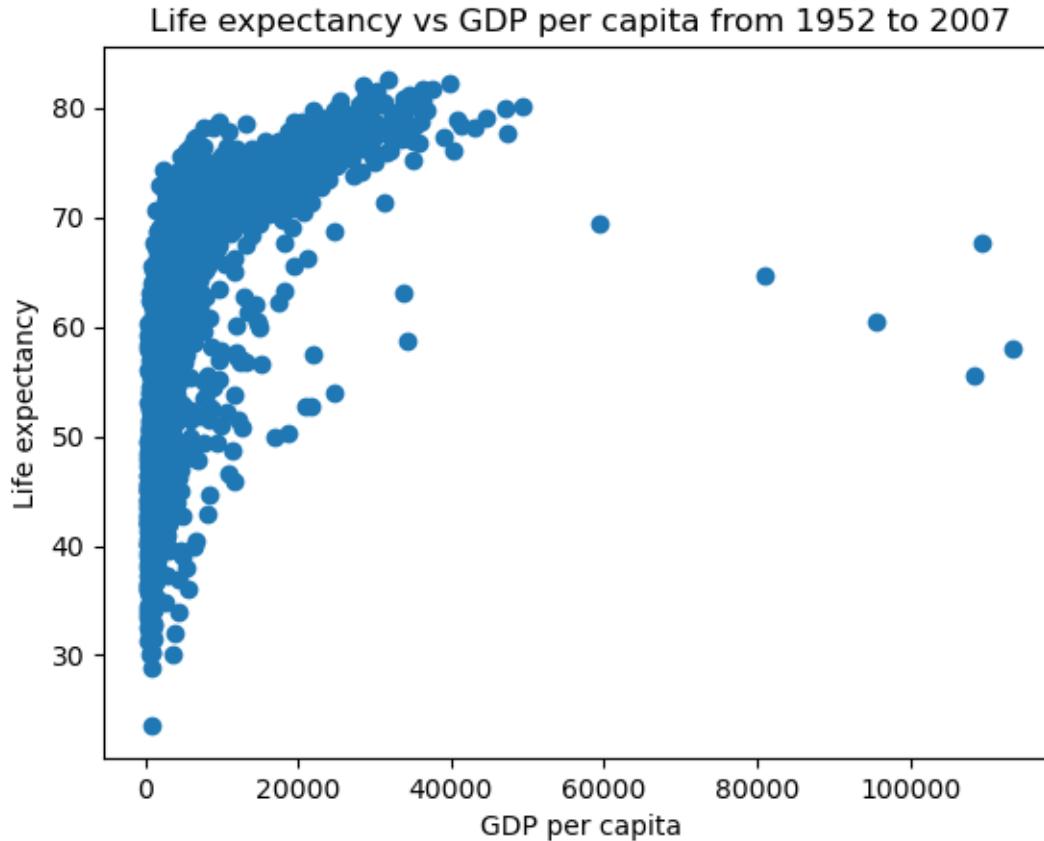
```
gdp_data = pd.read_csv('datasets/gdp_lifeExpectancy.csv')
gdp_data.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

```
#Object-oriented interface
fig, ax = plt.subplots() #Create a figure and an axes
x = gdp_data.gdpPercap
y = gdp_data.lifeExp
ax.plot(x,y,'o')    #Plot data on the axes
ax.set_xlabel('GDP per capita')    #Add an x-label to the axes
ax.set_ylabel('Life expectancy')    #Add a y-label to the axes
ax.set_title('Life expectancy vs GDP per capita from 1952 to 2007');
```



```
#pyplot interface
x = gdp_data.gdpPercap
y = gdp_data.lifeExp
plt.plot(x,y,'o') #By default, the plot() function makes a lineplot. The 'o' argument specifies
plt.xlabel('GDP per capita') #Labelling the horizontal X-axis
plt.ylabel('Life expectancy') #Labelling the vertical Y-axis
plt.title('Life expectancy vs GDP per capita from 1952 to 2007');
```



### 11.1.3 Pyplot Interface

In the previous chapter, our plotting is completely based on pyplot interface of Matplotlib, which is just for basic plotting. You can easily generate plots using pyplot module in the matplotlib library just by importing `matplotlib.pyplot` module.

- pyplot interface is a **state-based interface**. It implicitly tracks the plot that it wants to reference
- Simple functions are used to **add plot elements and/or modify the plot** as we need, whenever we need it.
- The Pyplot interface **shares a lot of similarities in syntax and methodology with MATLAB**.

However, The `pyplot` interface has limited functionality in these two cases:

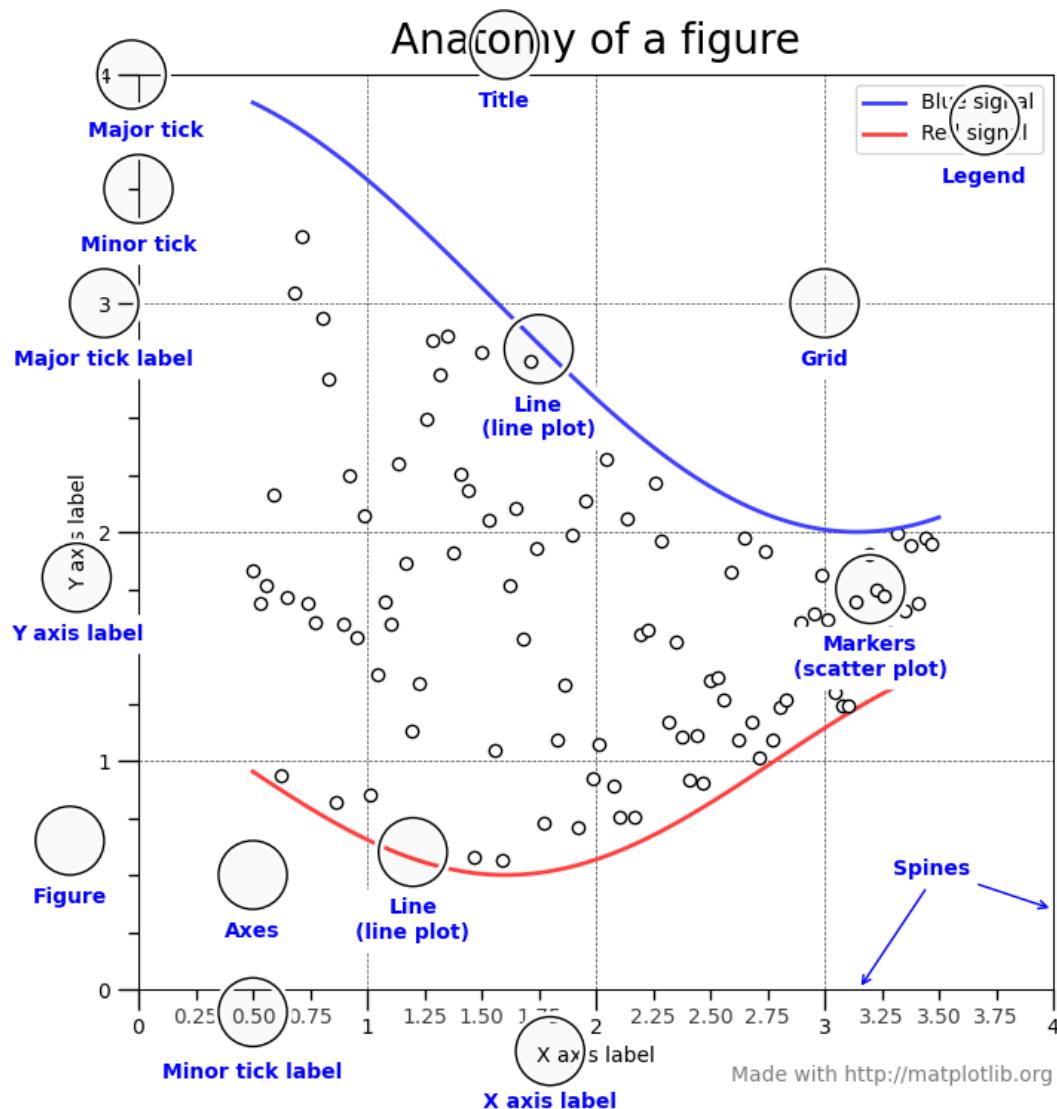
- when there is a need to make multiple plots
- when we have to make plots that require a lot of customization.

For more advanced plotting with Matplotlib, you have to learn Object-Oriented Interface.

## 11.2 Plotting with Object-Oriented Interface of Matplotlib

### 11.2.1 Matplotlib Object Anatomy

To use Matplotlib's object-oriented interface effectively, it's important to understand the anatomy of a plot—how different visual elements are structured within a figure:



## 11.2.2 Matplotlib Object Hierarchy

Matplotlib plots follow a **hierarchical structure**, with two core components:

- Figure: The overall container for one or more plots.
- Axes: The individual plot area where data is visualized.

Each Axes contains further elements such as:

- Axis (x-axis and y-axis)
- Title, Labels, Ticks
- Drawable objects like Lines, Text, and Patches (collectively called **Artists**)

Here's a simplified view of this hierarchy:

### Figure

```
Axes (1 or more)
  Axis (XAxis, YAxis)
  Title
  Label
  Tick
  Artist objects (Lines, Text, Patches, etc.)
```

Understanding this structure is key to using Matplotlib's object-oriented interface, as it allows you to **access and customize each component directly**.

### 11.2.2.1 Figure

The outermost object is the figure which is an instance of `figure.Figure`. It is the **top level container** for all the plot elements. The Figure is the final image that may contain one or more Axes and it keeps track of all the Axes. Figure is only a container and you can not plot data on figure.

```
# To begin with, we create a figure instance which provides an empty canvas
fig = plt.figure()
```

```
<Figure size 640x480 with 0 Axes>
```

Figure is just a container, note that creating figure using `plt.figure()` does not automatically create an axes and hence all you can see is the figure object.

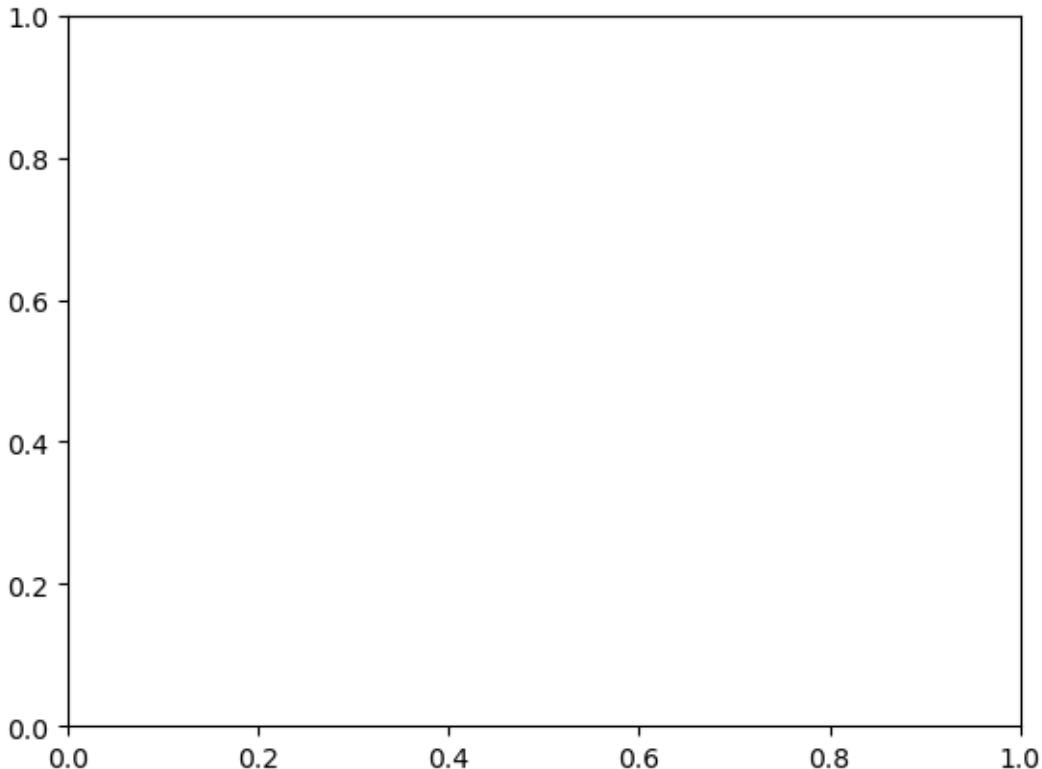
### 11.2.2.2 Axes

Axes is the instance of `matplotlib.axes.Axes`. Axes is the area on which data are plotted. A given figure can contain many Axes, but a given Axes object can only be in one Figure.

#### 11.2.2.2.1 Creating an Axes Object Explicitly

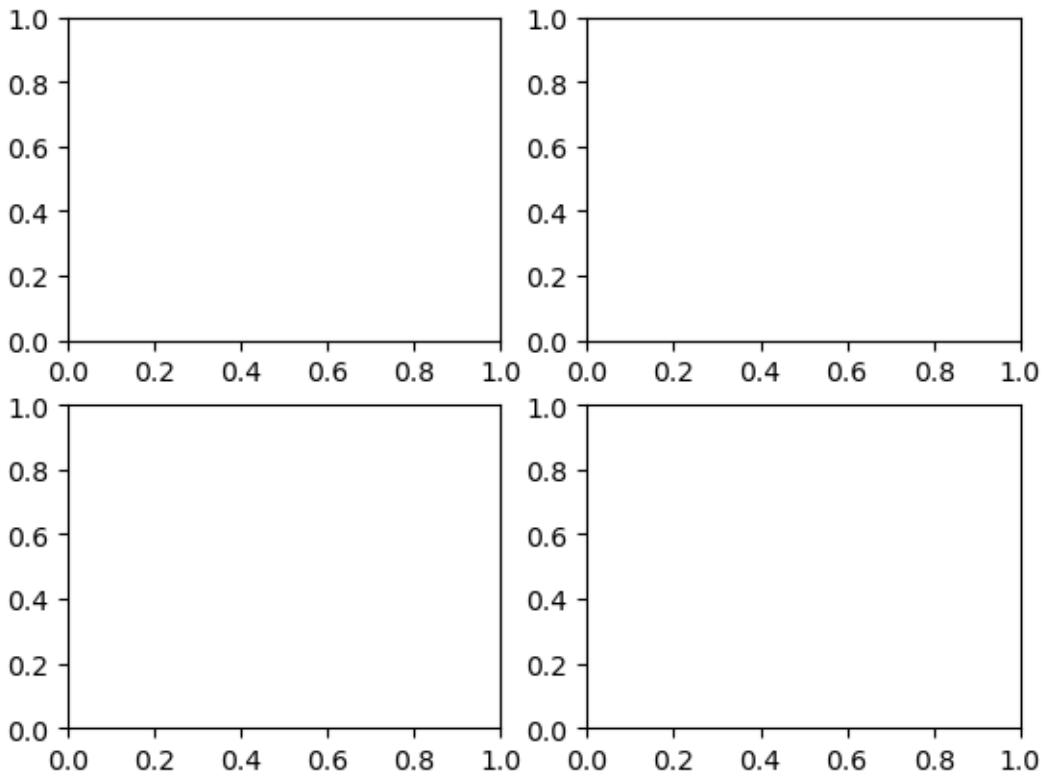
In the OOP interface, Axes objects are usually created using `plt.subplots()` or `plt.figure().add_subplot()`.

```
# A figure only contains one axes by default
fig = plt.figure()
# add axes to the figure
ax = fig.add_subplot()
```



We create a blank axes (area) for plotting, if we need to add more axes to the figure

```
# create a figure with 4 axes, arranged in 2 rows and 2 columns
fig, axes = plt.subplots(2, 2)
```



#### 11.2.2.2 Components of an Axes Object

The `Axes` object contains several elements that make up the plot

- Data plotting area: Contains the data (lines, bars, points, etc.).
- X-axis and Y-axis: Controls the axis limits, labels, and ticks.
- Title and Labels: The overall title and labels for each axis.
- Gridlines: Optional lines to help align the data visually.
- Spines: The borders around the plot.
- Legend: An optional component to explain the data series.
- Annotations: Text or arrows highlighting points of interest.

This is what makes the `Axes` object central to any plot in the OOP interface of Matplotlib.

```

# create data for plotting
x = np.arange(10)
y = x**2

# create a figure and axes
fig,ax = plt.subplots(1,1)

# plot the data
ax.plot(x,y)

# set the title
ax.set_title("Exponential Plot")

# set the labels of x and y axes
ax.set_xlabel("age")
ax.set_ylabel("Cell growth")

# set the limits of x and y axes
ax.set_xlim([0, 10])
ax.set_ylim([0, 100])

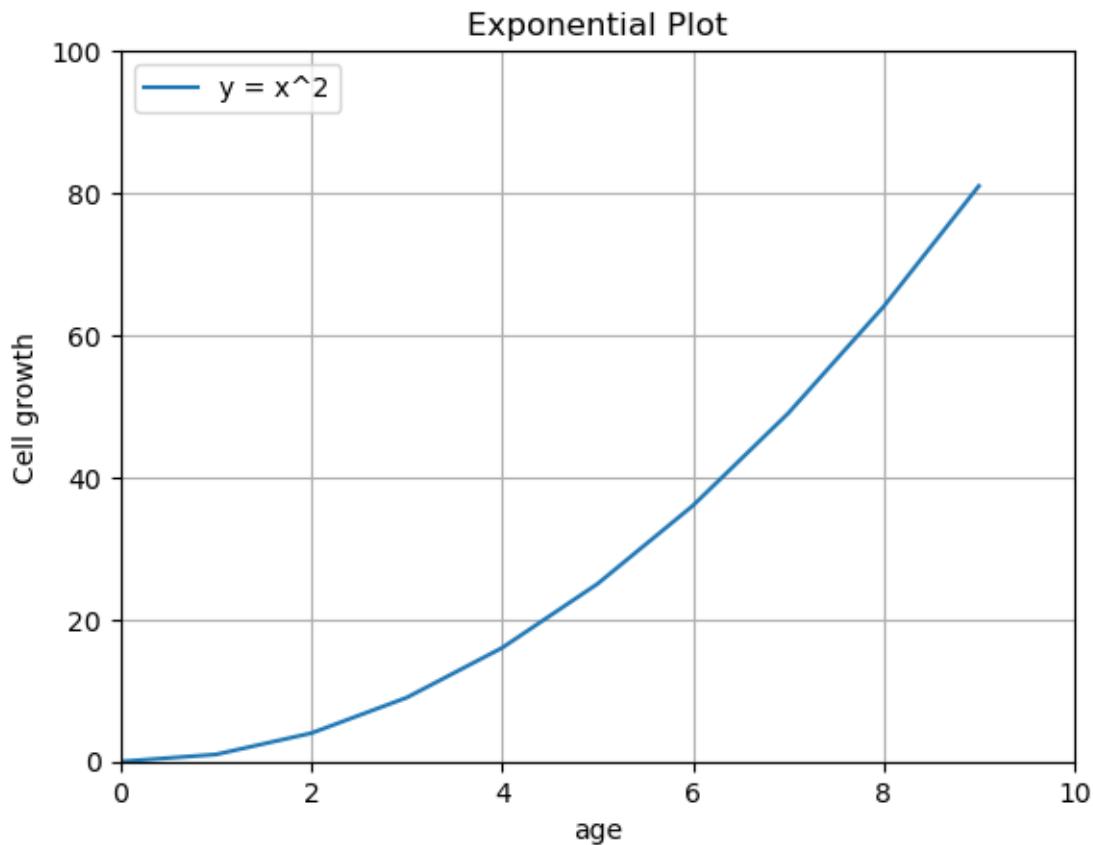
# set the ticks of x and y axes
ax.set_xticks(range(0, 11, 2))
ax.set_yticks(range(0, 101, 20))

# add grid
ax.grid(True)

# add legend
ax.legend(["y = x^2"], loc = "upper left")

# show the plot
plt.show()

```



### 11.2.3 Creating Complex Plots with Multiple Subplots

To help illustrate subplots in matplotlib, we can cover two scenarios:

- subplots that don't overlap and
- subplots inside other subplots

#### 11.2.3.1 Plotting non-overlapped subplots

This is the most common use case, where multiple plots are placed next to each other in a grid, without overlap. The `plt.subplots()` function allows for a clean layout where each plot is contained in its own space. You can specify the number of rows and columns to create a grid of subplots.

```

flowers_df = sns.load_dataset('iris')
tips_df = sns.load_dataset('tips')
flights_df = sns.load_dataset("flights").pivot(index="month", columns="year", values="passenger")

fig, axes = plt.subplots(2, 2, figsize=(10, 6))

# Pass the axes into seaborn
axes[0,0].set_title('Sepal Length vs. Sepal Width')
sns.scatterplot(x=flowers_df.sepal_length,
                 y=flowers_df.sepal_width,
                 hue=flowers_df.species,
                 s=100,
                 ax=axes[0,0])

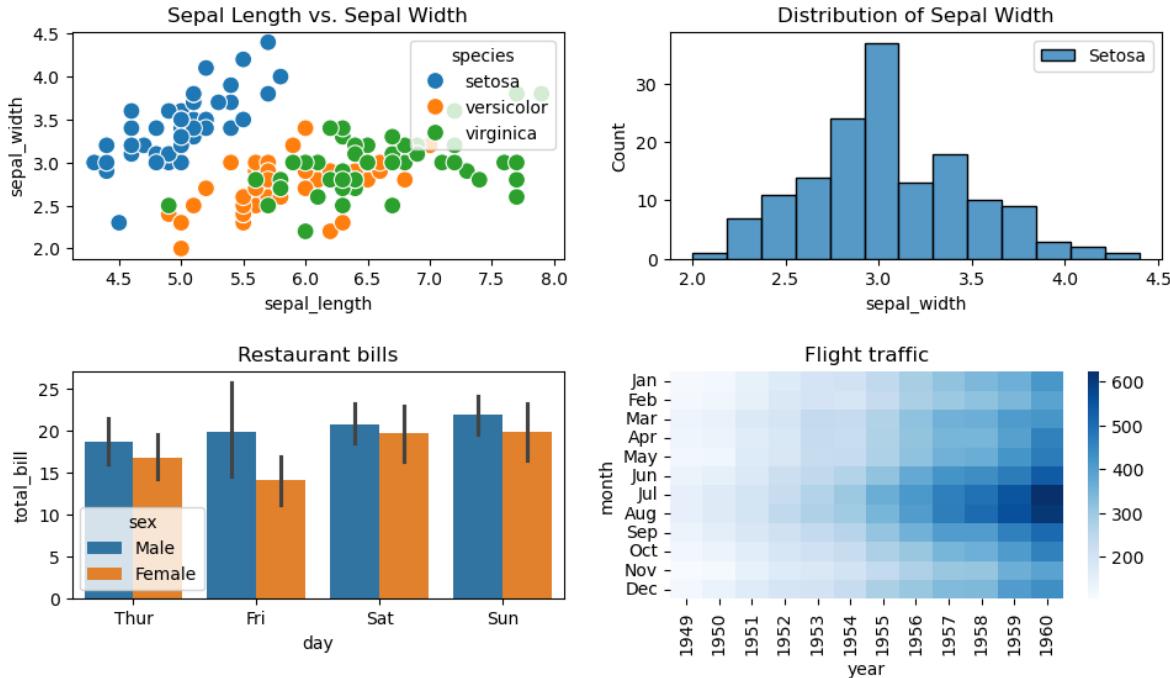
# Use the axes for plotting
axes[0,1].set_title('Distribution of Sepal Width')
sns.histplot(flowers_df.sepal_width, ax=axes[0,1])
axes[0,1].legend(['Setosa', 'Versicolor', 'Virginica'])

# Pass the axes into seaborn
axes[1,0].set_title('Restaurant bills')
sns.barplot(x='day', y='total_bill', hue='sex', data=tips_df, ax=axes[1,0])

# Pass the axes into seaborn
axes[1,1].set_title('Flight traffic')
sns.heatmap(flights_df, cmap='Blues', ax=axes[1,1])

plt.tight_layout(pad=2)

```



- The `plt.tight_layout()` ensures the subplots do not overlap by adjusting the spacing automatically.
- Seaborn and pandas are wrappers of Matplotlib. To create a Seaborn/pandas plot in a specific Matplotlib subplot, you pass the `ax` parameter to its plotting function. This allows you to use their visualization capabilities while fully controlling the layout of the plot using Matplotlib's `plt.subplots()`

### 11.2.3.2 Plotting Nested Subplots (Subplots Inside Other Subplots)

You can create a subplot inside another plot using `add_axes()` or `inset_axes()` from matplotlib's `Axes` object. This is useful for creating insets or focusing on a specific region within a larger plot.

Syntax of `add_axes()`

```
ax = fig.add_axes([left, bottom, width, height])
```

where

- `left`: The x-position (horizontal starting point) of the axes, as a fraction of the figure width (0 to 1).
- `bottom`: The y-position (vertical starting point) of the axes, as a fraction of the figure height (0 to 1).
- `width`: The width of the axes, as a fraction of the figure width (0 to 1).

- height: The height of the axes, as a fraction of the figure height (0 to 1).

Below is an example demonstrating the use of `add_axes()`. You can also explore the `inset_axes()` method for creating inset plots with more flexibility

```
# create inset axes within the main plot axes

np.random.seed(19680801) # Fixing random state for reproducibility.

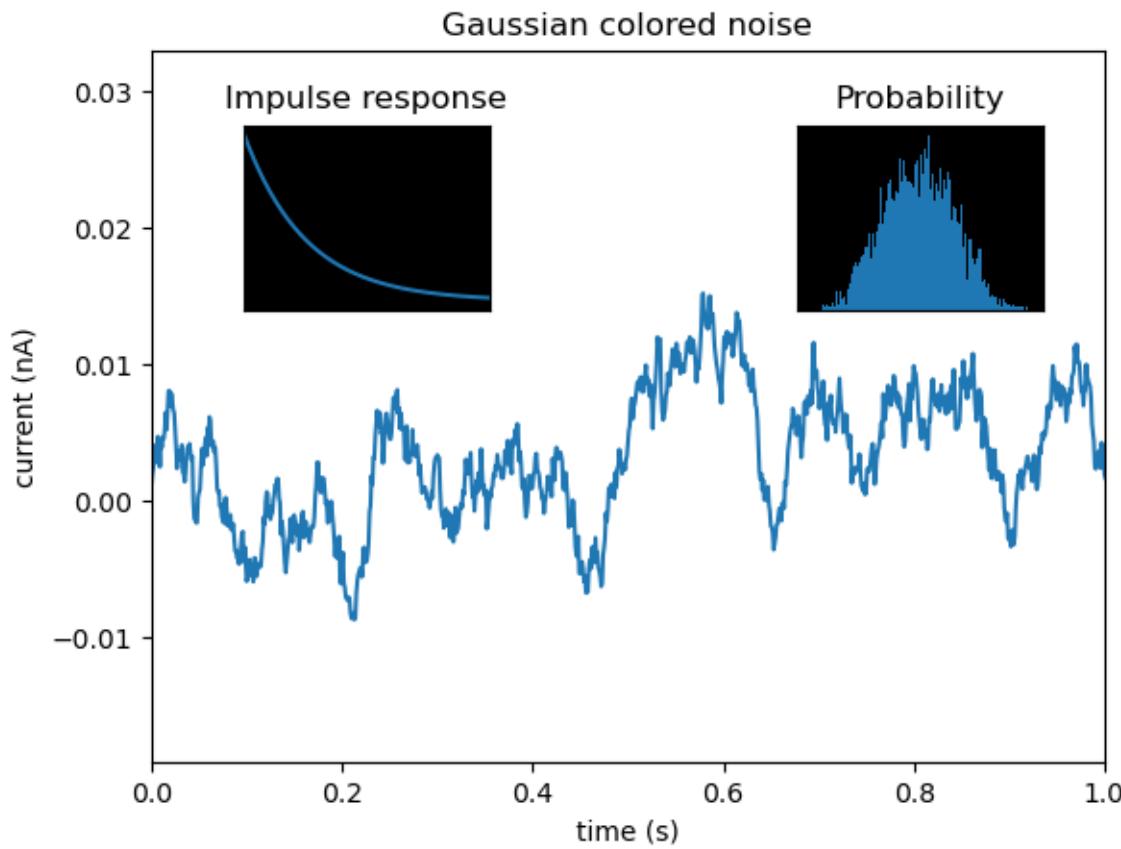
# create some data to use for the plot
dt = 0.001
t = np.arange(0.0, 10.0, dt)
r = np.exp(-t[:1000] / 0.05) # impulse response
x = np.random.randn(len(t))
s = np.convolve(x, r)[:len(x)] * dt # colored noise

fig, main_ax = plt.subplots()
main_ax.plot(t, s)
main_ax.set_xlim(0, 1)
main_ax.set_ylim(1.1 * np.min(s), 2 * np.max(s))
main_ax.set_xlabel('time (s)')
main_ax.set_ylabel('current (nA)')
main_ax.set_title('Gaussian colored noise')

# this is an inset axes over the main axes
right_inset_ax = fig.add_axes([.65, .6, .2, .2], facecolor='k')
right_inset_ax.hist(s, 400, density=True)
right_inset_ax.set(title='Probability', xticks=[], yticks=[])

# this is another inset axes over the main axes
left_inset_ax = fig.add_axes([.2, .6, .2, .2], facecolor='k')
left_inset_ax.plot(t[:len(r)], r)
left_inset_ax.set(title='Impulse response', xlim=(0, .2), xticks=[], yticks=[])

plt.show()
```



#### 11.2.4 Advanced Customization with Matplotlib's Object-Oriented Interface

Below, we are reading the dataset of noise complaints of type Loud music/Party received the police in New York City in 2016.

```
nyc_party_complaints = pd.read_csv('datasets/party_nyc.csv')
nyc_party_complaints.head()
```

	Created Date	Closed Date	Location Type	Incident Zip	City	Borough
0	12/31/2015 0:01	12/31/2015 3:48	Store/Commercial	10034.0	NEW YORK	MANHATTAN
1	12/31/2015 0:02	12/31/2015 4:36	Store/Commercial	10040.0	NEW YORK	MANHATTAN
2	12/31/2015 0:03	12/31/2015 0:40	Residential Building/House	10026.0	NEW YORK	MANHATTAN
3	12/31/2015 0:03	12/31/2015 1:53	Residential Building/House	11231.0	BROOKLYN	BROOKLYN
4	12/31/2015 0:05	12/31/2015 3:49	Residential Building/House	10033.0	NEW YORK	MANHATTAN

Below, we will begin with basic plotting, utilizing Matplotlib's object-oriented interface to handle more complex tasks, such as setting the major axis formatting. When it comes to advanced customization, Matplotlib's object-oriented interface offers greater flexibility and control compared to the pyplot interface.

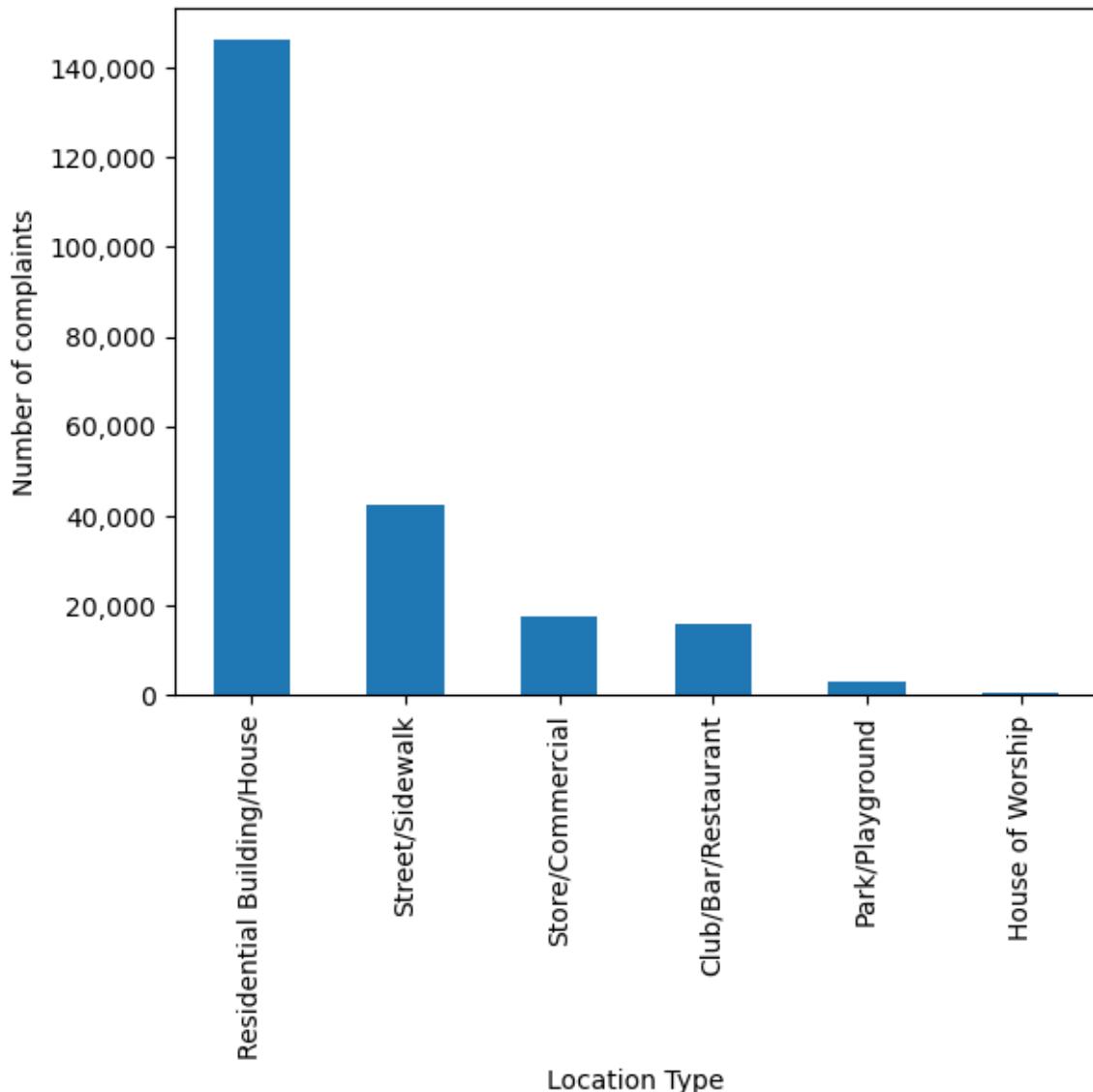
#### 11.2.4.1 Bar plots with Pandas

**Purpose of bar plots:** Barplots are used to visualize any aggregate statistics of a continuous variable with respect to the categories or levels of a categorical variable.

Bar plots can be made using the pandas `bar` function with the DataFrame or Series, just like the line plots and scatterplots.

Let us visualise the locations from where the complaints are coming.

```
ax = nyc_party_complaints['Location Type'].value_counts().plot.bar(ylabel = 'Number of complaints')
ax.yaxis.set_major_formatter('{x:, .0f}')
```



In the above code, we use `ax.yaxis.set_major_formatter` to format the y-axis labels in a currency style. From the above plot, we observe that most of the complaints come from residential buildings and houses, as one may expect.

For categorical variables, we can use the `.value_counts()` method to get the statistical frequency of each unique value.

```
nyc_party_complaints['Location Type'].value_counts()
```

Location Type

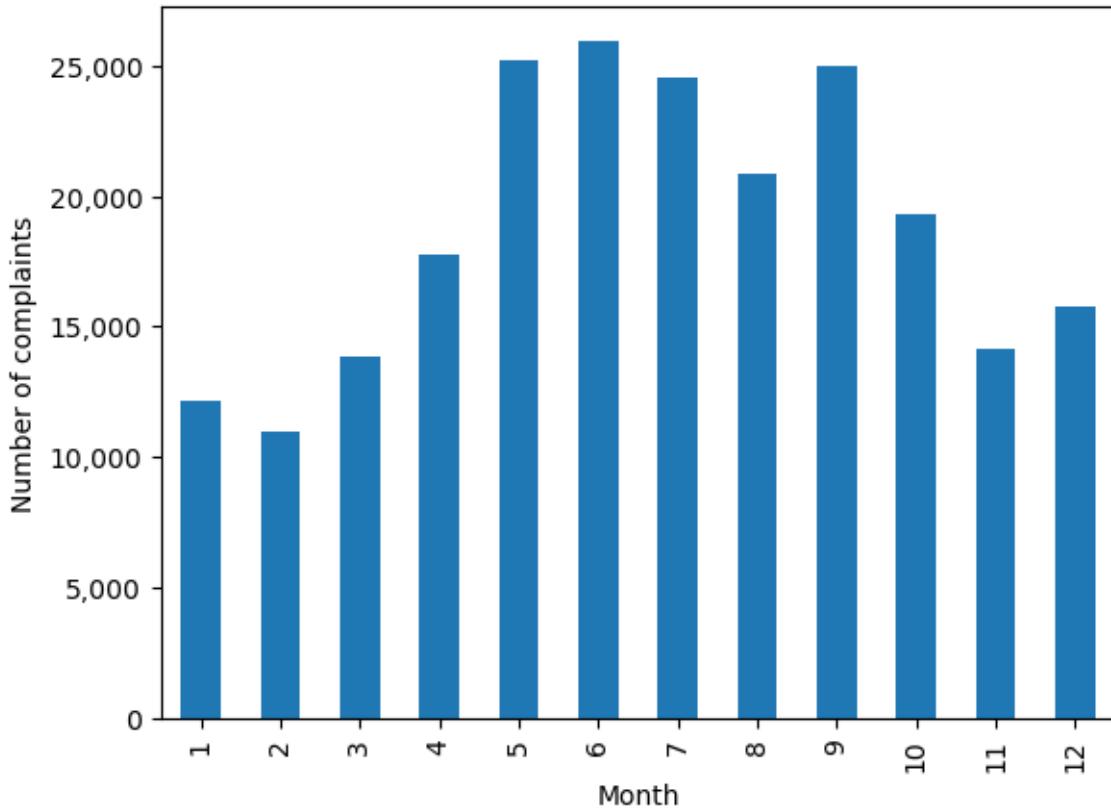
```
Residential Building/House      146040
Street/Sidewalk                  42353
Store/Commercial                 17617
Club/Bar/Restaurant               15766
Park/Playground                  3036
House of Worship                  602
Name: count, dtype: int64
```

Next, Let's visualize the time of the year when most complaints occur.

```
nyc_party_complaints['Month_of_the_year'].value_counts()
```

```
Month_of_the_year
6      25933
5      25192
9      25000
7      24502
8      20833
10     19332
4      17718
12     15730
11     14146
3      13880
1      12171
2      10977
Name: count, dtype: int64
```

```
#Using the pandas function bar() to create bar plot
ax = nyc_party_complaints['Month_of_the_year'].value_counts().sort_index().plot.bar(ylabel =
                                                                           xlabel = "Montl
ax.yaxis.set_major_formatter('{x:,.0f}')
```



Try executing the code without `sort_index()` to figure out the purpose of using the function.

From the above plot, we observe that most of the complaints occur during summer and early Fall.

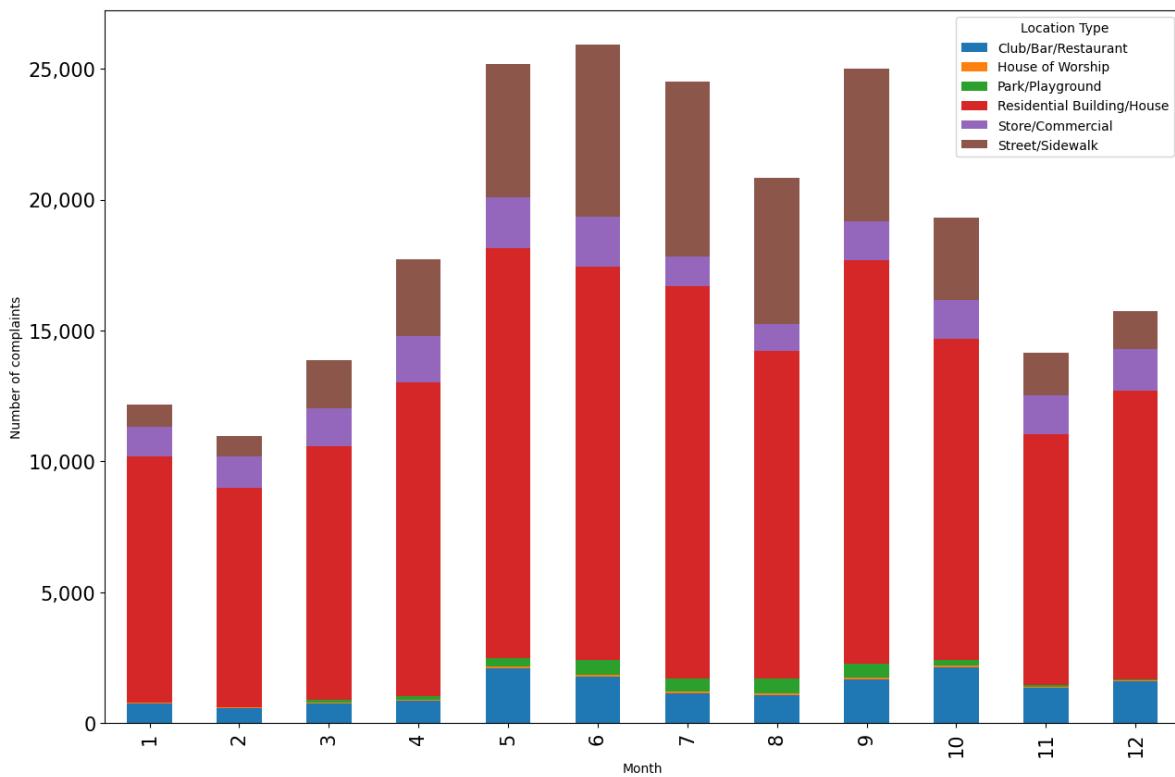
Let us create a stacked bar chart that combines both the above plots into a single plot. You may ignore the code used for re-shaping the data until Chapter 10. The purpose here is to show the utility of the pandas `bar()` function.

```
#Reshaping the data to make it suitable for a stacked barplot - ignore this code until chapter 10
complaints_location=pd.crosstab(nyc_party_complaints.Month_of_the_year, nyc_party_complaints
complaints_location.head()
```

Location Type Month_of_the_year	Club/Bar/Restaurant	House of Worship	Park/Playground	Residential Building/Hotel
1	748	24	17	9393
2	570	29	16	8383
3	747	39	90	9689

Location Type	Club/Bar/Restaurant	House of Worship	Park/Playground	Residential Building/House	Street/Sidewalk
Month_of_the_year					
4	848	53	129	11984	
5	2091	72	322	15676	

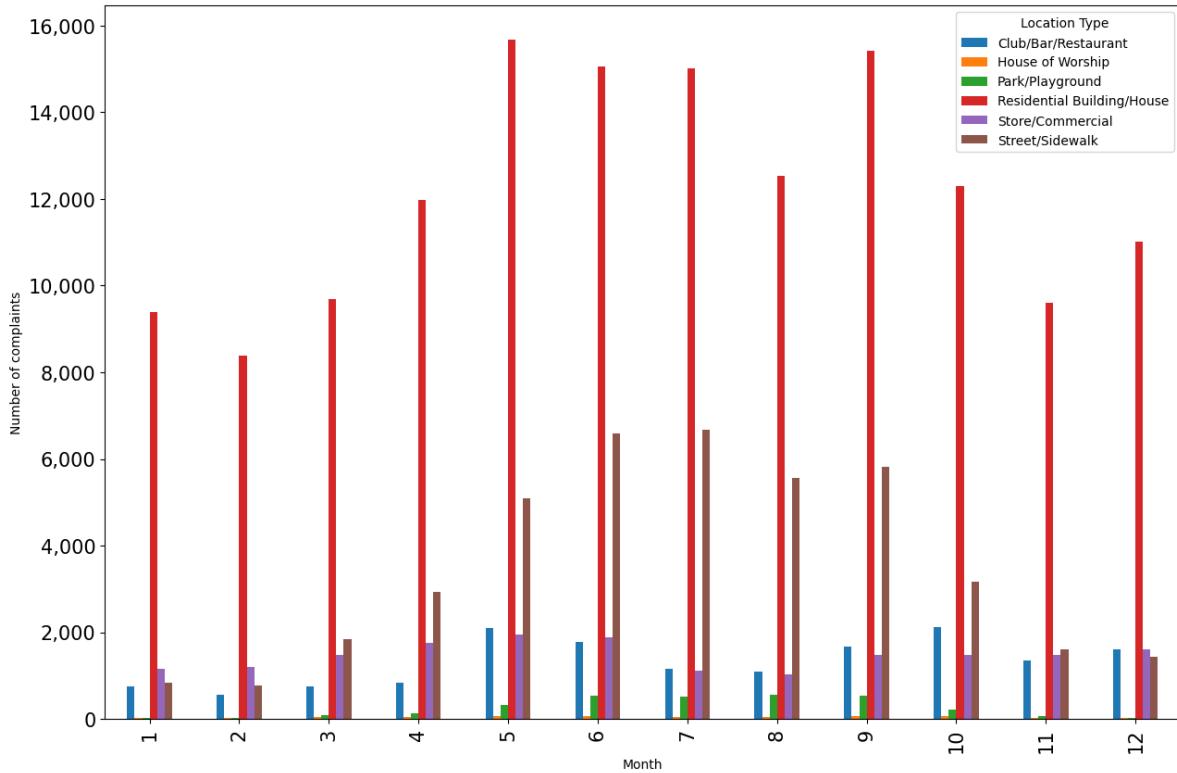
```
#Stacked bar plot showing number of complaints at different months of the year, and from different location types
ax = complaints_location.plot.bar(stacked=True,ylabel = 'Number of complaints',figsize=(15, 10))
ax.tick_params(axis = 'both',labelsize=15)
ax.yaxis.set_major_formatter('{x:,.0f}')
```



The above plots gives the insights about location and day of the year simultaneously that were previously separately obtained by the individual plots.

An alternative to stacked barplots are side-by-side barplots, as shown below.

```
#Side-by-side bar plot showing number of complaints at different months of the year, and from different location types
ax = complaints_location.plot.bar(ylabel = 'Number of complaints',figsize=(15, 10), xlabel = 'Month')
ax.tick_params(axis = 'both',labelsize=15)
ax.yaxis.set_major_formatter('{x:,.0f}')
```



Question: In which scenarios should we use a stacked barplot instead of a side-by-side barplot and vice-versa?

#### 11.2.4.2 Bar plots with confidence intervals with Seaborn

We'll group the data to obtain the total complaints for each *Location Type*, *Borough*, *Month\_of\_the\_year*, and *Hour\_of\_the\_day*. Note that you'll learn grouping data in later chapters, so you may ignore the next code block. The grouping is done to shape the data in a suitable form for visualization.

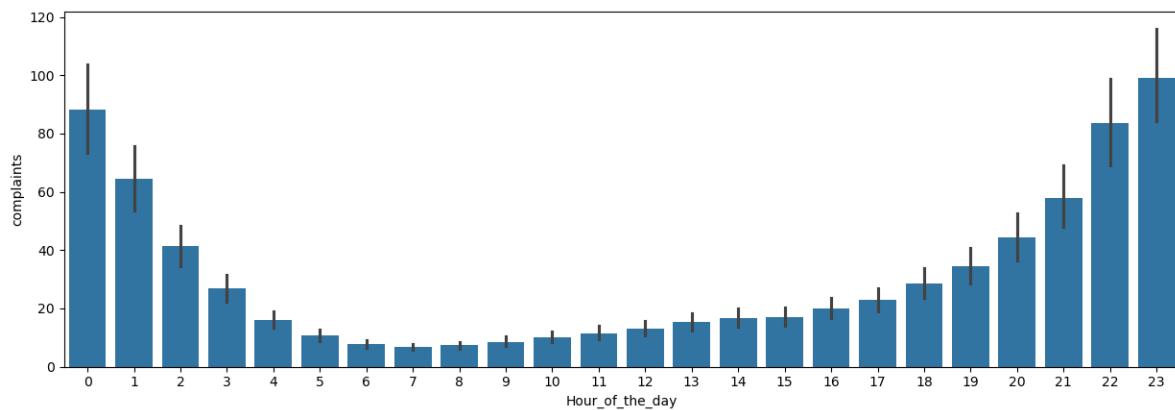
```
#Grouping the data to make it suitable for visualization using Seaborn. Ignore this code block
nyc_complaints_grouped = nyc_party_complaints[['Location Type','Borough','Month_of_the_year']]
nyc_complaints_grouped.head()
```

	Location Type	Borough	Month_of_the_year	Hour_of_the_day	complaints
0	Club/Bar/Restaurant	BRONX	1	0	10
1	Club/Bar/Restaurant	BRONX	1	1	10
2	Club/Bar/Restaurant	BRONX	1	2	6

Location_Type	Borough	Month_of_the_year	Hour_of_the_day	complaints
3 Club/Bar/Restaurant	BRONX	1	3	6
4 Club/Bar/Restaurant	BRONX	1	4	3

Let us create a bar plot visualizing the average number of complaints with the time of the day.

```
ax = sns.barplot(x="Hour_of_the_day", y = 'complaints', data=nyc_complaints_grouped)
ax.figure.set_figwidth(15)
```



From the above plot, we observe that most of the complaints are made around midnight. However, interestingly, there are some complaints at each hour of the day.

Note that the above barplot shows the mean number of complaints in a month at each hour of the day. The black lines are the 95% confidence intervals of the mean number of complaints.

### 11.2.5 pyplot: a convenience wrapper around the object-oriented interface

While the pyplot interface is simpler for quick, basic plots, it ultimately wraps around the object-oriented structure of Matplotlib, meaning that it's built on top of the object-oriented interface.

## 11.3 Creating Subplots with Seaborn

We previously demonstrated how Seaborn integrates seamlessly with Matplotlib's object-oriented interface, allowing you to pass the `ax` argument to any Seaborn function, thereby directing the plot to a specific axis within a subplot grid.

Additionally, Seaborn offers a more convenient and simplified approach to creating subplots, thanks to its high-level functions and built-in integration with Matplotlib. Here's how Seaborn makes working with subplots easier:

### 11.3.1 Using Facetgrid

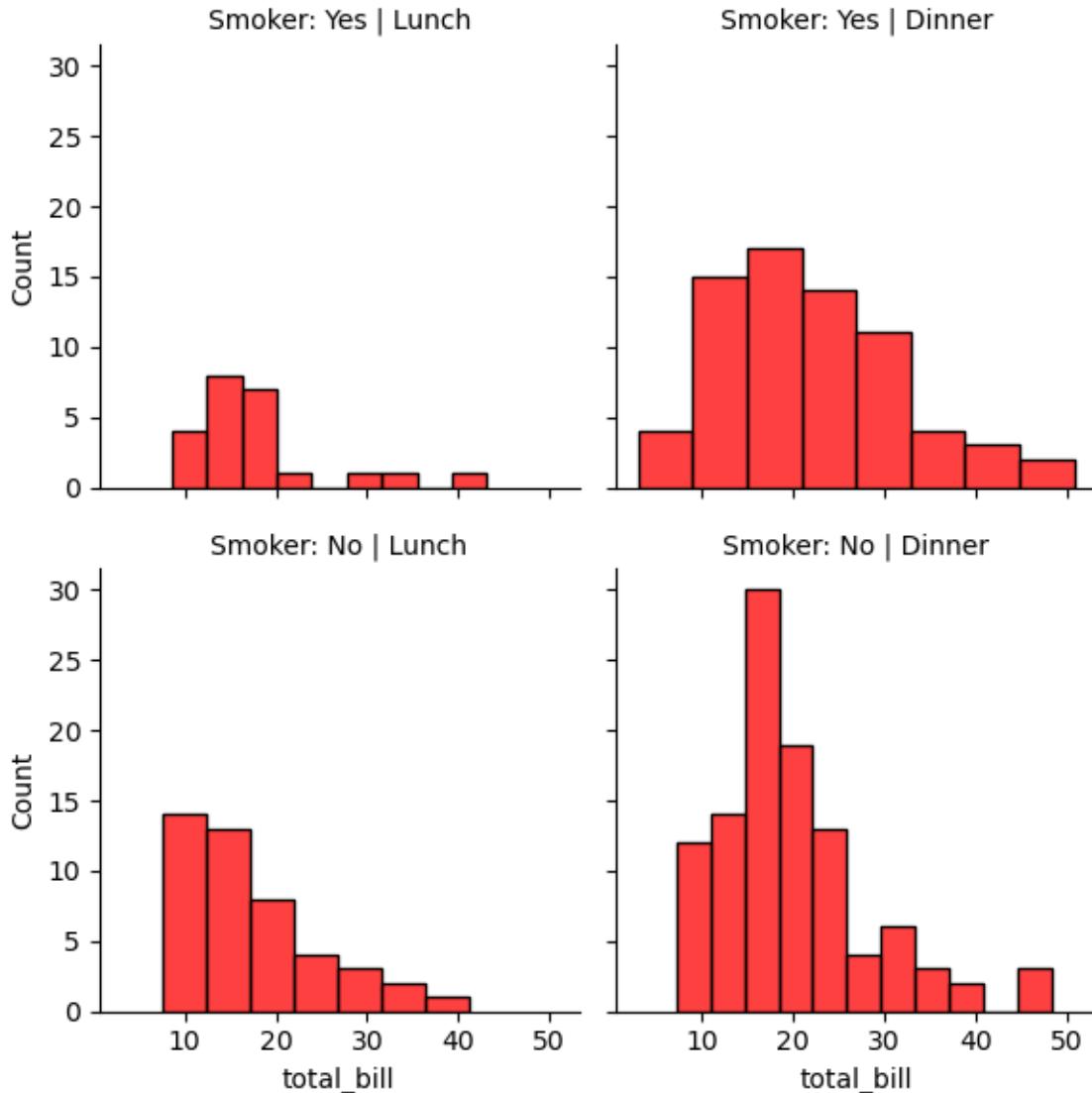
Seaborn's `FacetGrid` function make it very easy to create facet grids or subplots based on data dimensions (such as categories), which would require more manual effort with Matplotlib.

You can use the `row` and `col` parameters to control how the data is split into subplots along these dimensions.

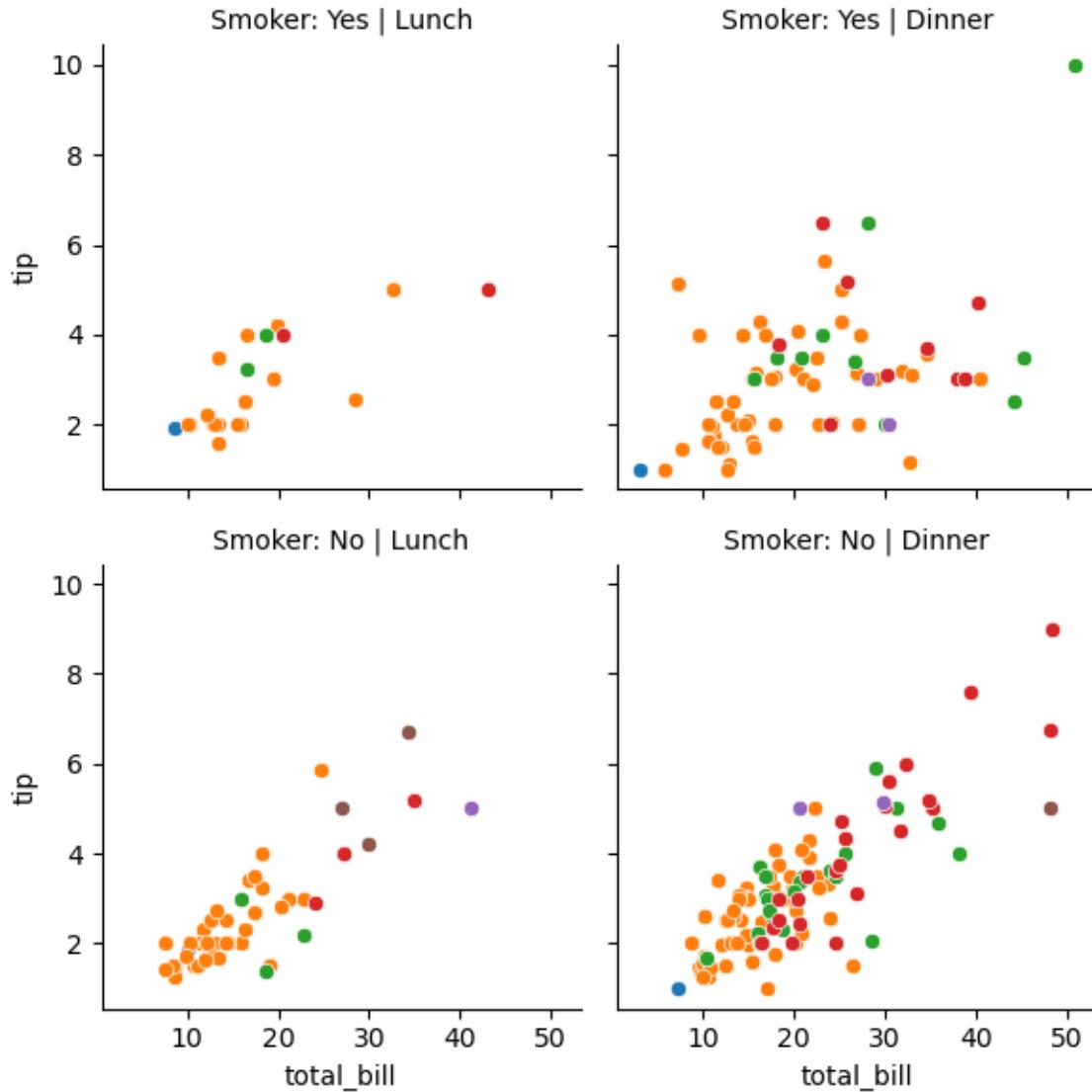
```
# Seaborn Example using FacetGrid:  
tips_df = sns.load_dataset("tips")  
tips_df.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

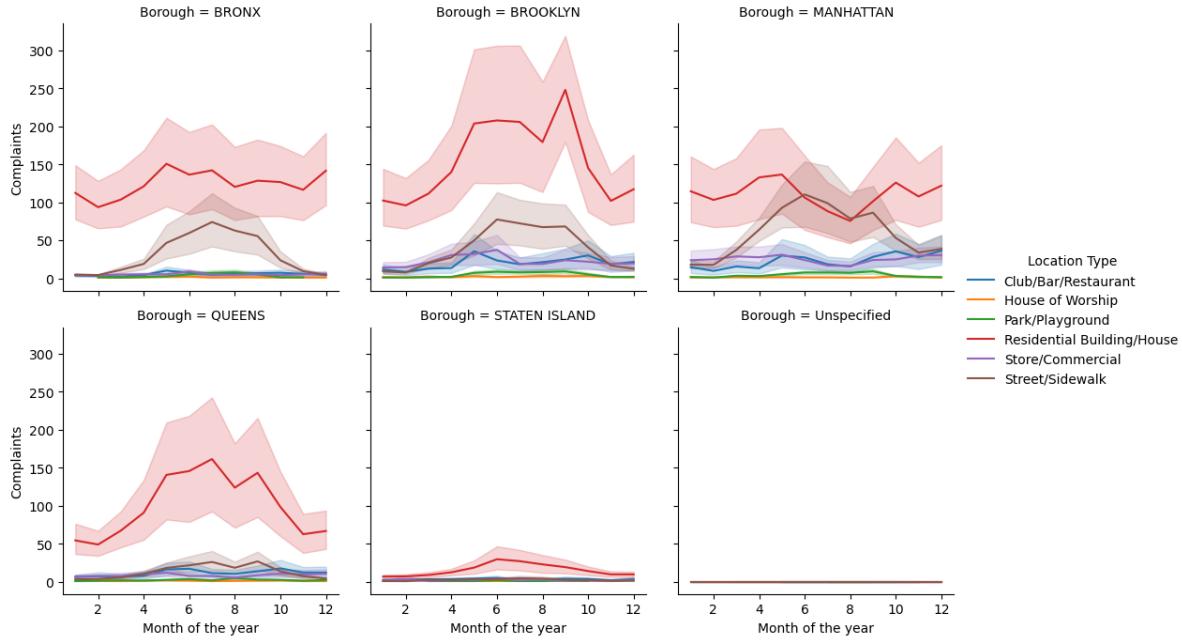
```
g = sns.FacetGrid(tips_df, col='time', row='smoker')  
g.map(sns.histplot, 'total_bill', color='r')  
g.set_titles(col_template="{col_name}", row_template="Smoker: {row_name}");
```



```
# adding hue to the FacetGrid
g = sns.FacetGrid(tips_df, col='time', row='smoker', hue='size')
# Plot a scatterplot of the total bill and tip for each combination of time and smoker
g.map(sns.scatterplot, 'total_bill', 'tip')
g.set_titles(col_template="{col_name}", row_template="Smoker: {row_name}");
```



```
#Visualizing the number of complaints with Month_of_the_year, Location Type, and Borough.
a = sns.FacetGrid(nyc_complaints_grouped, hue = 'Location Type', col = 'Borough', col_wrap=3, )
# Plotting a lineplot to show the number of complaints with Month_of_the_year
a.map(sns.lineplot,'Month_of_the_year','complaints')
a.set_axis_labels("Month of the year", "Complaints")
a.add_legend()
```

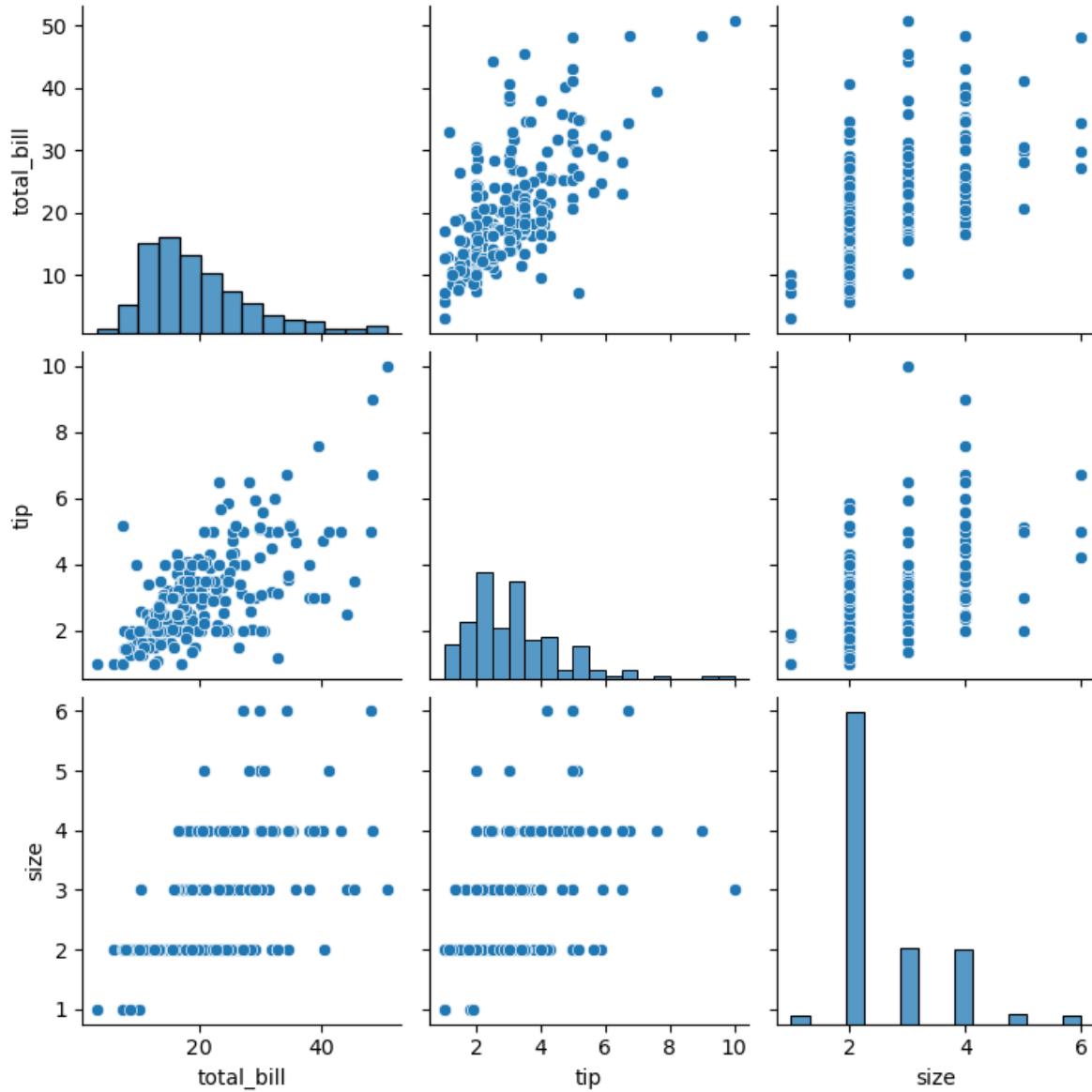


### 11.3.2 Using Pairplot

Pairplots are used to visualize the association between all variable-pairs in the data. In other words, pairplots simultaneously visualize the scatterplots between all variable-pairs.

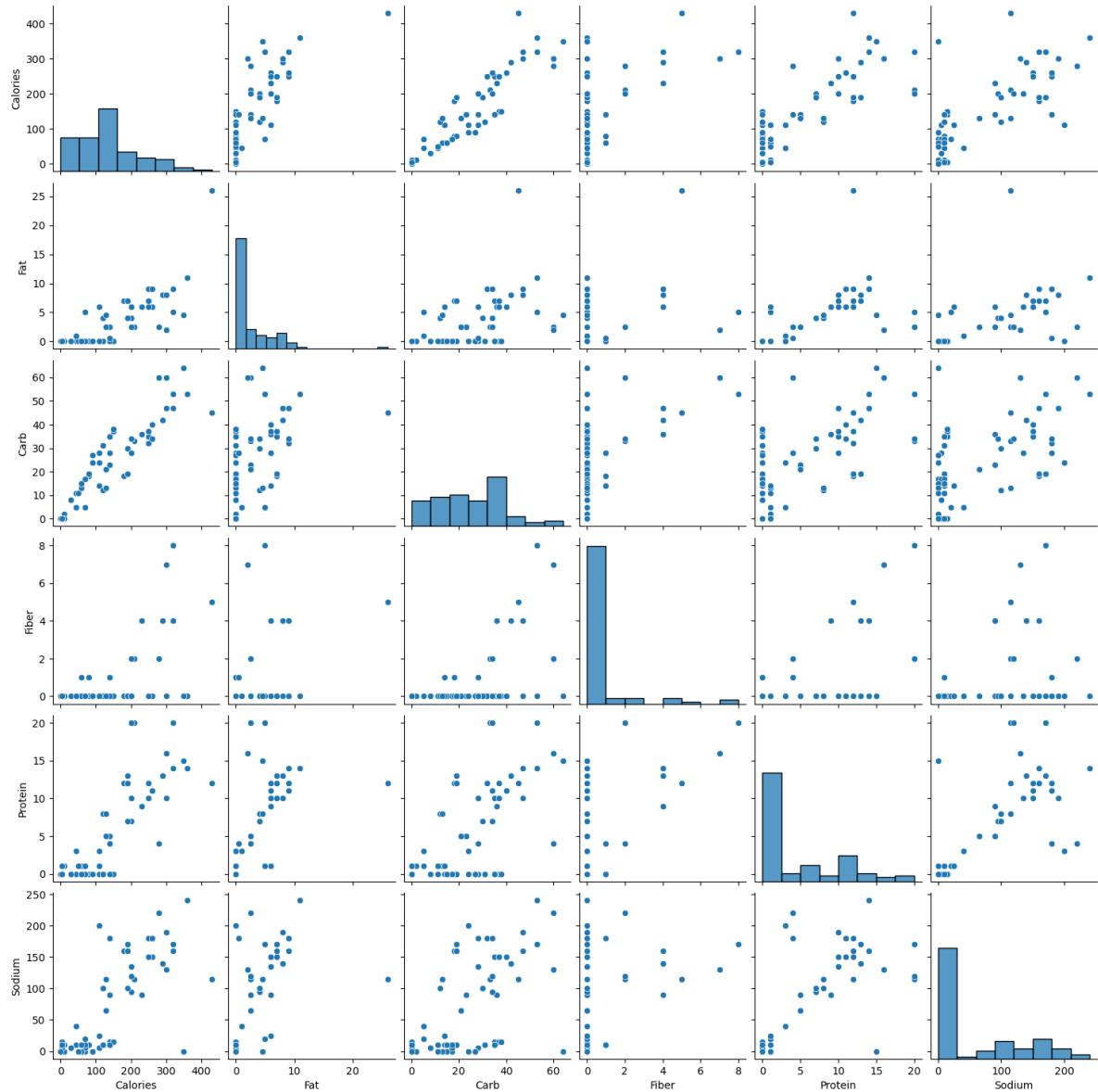
Let us visualize the pair-wise association of tips variables in the tips dataset

```
sns.pairplot(tips_df);
```



Let us visualize the pair-wise association of nutrition variables in the starbucks drinks data.

```
starbucks_drinks = pd.read_csv('datasets/starbucks-menu-nutrition-drinks.csv')
sns.pairplot(starbucks_drinks);
```



In the above pairplot, note that:

- The histograms on the diagonal of the grid show the distribution of each of the variables.
- Instead of a histogram, we can visualize the density plot with the argument `kde = True`.
- The scatterplots in the rest of the grid are the pair-wise plots of all the variables.

## 11.4 Geospatial Plotting

There are several widely used Python packages specifically designed for working with geospatial datasets. In this lesson, we will cover:

- GeoPandas
- Folium

Let's import them

```
import geopandas as gpd
import geopandas
import folium
import geodatasets
```

```
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\paramiko\transport.py:219: Cryptog
  "class": algorithms.Blowfish,
```

### 11.4.1 Static Plots with GeoPandas

A **shapefile** is a widely-used format for storing geographic information system (GIS) data, specifically vector data. It contains geometries (like points, lines, and polygons) that represent features on the earth's surface, along with associated attributes for each feature, such as names, populations, or other data relevant to the feature.

#### 11.4.1.1 Components of a Shapefile

A shapefile isn't a single file but a collection of files with the same name and different extensions, which work together to store geographic and attribute data:

- **.shp**: Stores the geometry (shapes of features, like points, lines, polygons).
- **.shx**: Contains an index to quickly access geometries in the .shp file.
- **.dbf**: A table storing attributes associated with each feature.

There may also be other optional files (e.g., .prj for projection information).

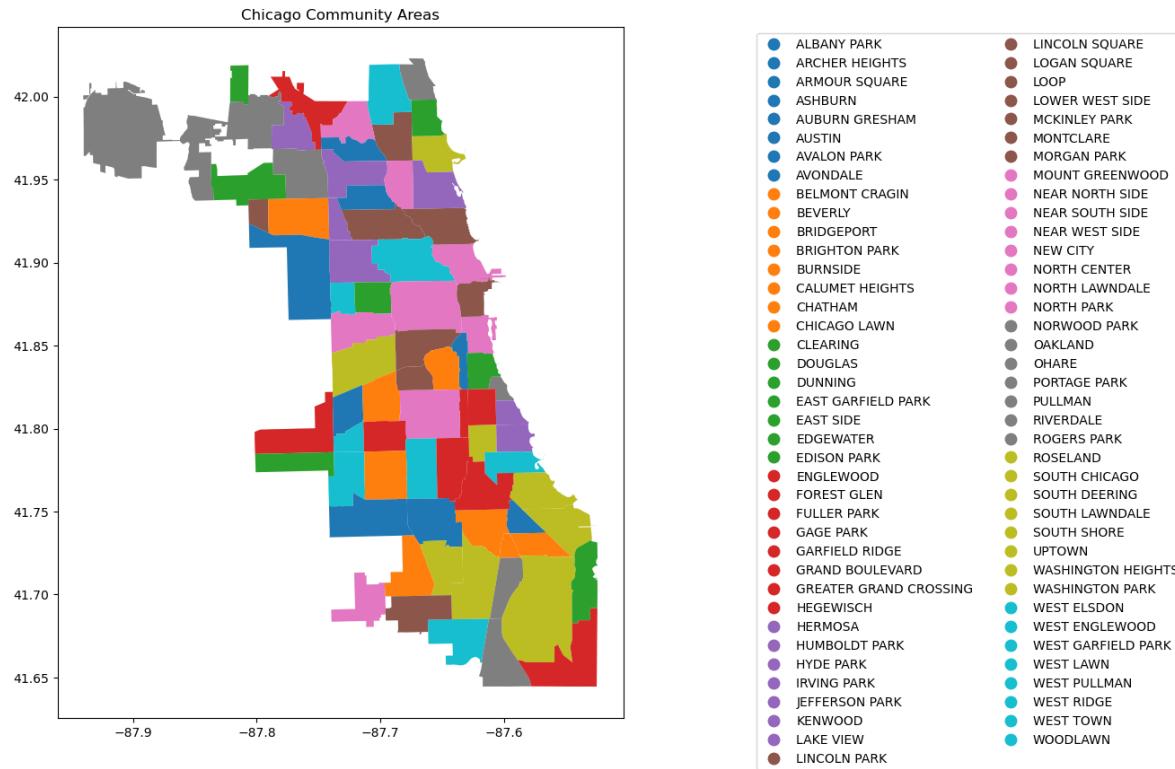
```
# Create figure and axis
fig, ax = plt.subplots(figsize=(15, 10))

# Plot your GeoDataFrame
chicago = gpd.read_file(r'datasets/chicago_boundaries\geo_export_26bce2f2-c163-42a9-9329-9ca
```

```

chicago.plot(column='community', ax=ax, legend=True, legend_kwds={'ncol': 2, 'bbox_to_anchor':
    # Add title (optional)
    plt.title('Chicago Community Areas');

```



Let's print out the information in the shapefile

```
chicago.head()
```

	area	area_num_1	area_numbe	comarea	comarea_id	community	perimeter	shape
0	0.0	35	35	0.0	0.0	DOUGLAS	0.0	4.6004
1	0.0	36	36	0.0	0.0	OAKLAND	0.0	1.6913
2	0.0	37	37	0.0	0.0	FULLER PARK	0.0	1.9916
3	0.0	38	38	0.0	0.0	GRAND BOULEVARD	0.0	4.8492
4	0.0	39	39	0.0	0.0	KENWOOD	0.0	2.9071

```
chicago['geometry'].head()

0    POLYGON ((-87.60914 41.84469, -87.60915 41.844...
1    POLYGON ((-87.59215 41.81693, -87.59231 41.816...
2    POLYGON ((-87.6288 41.80189, -87.62879 41.8017...
3    POLYGON ((-87.60671 41.81681, -87.6067 41.8165...
4    POLYGON ((-87.59215 41.81693, -87.59215 41.816...
Name: geometry, dtype: geometry
```

```
# Check the column names to see available data fields
print("Columns in the shapefile:", chicago.columns)

# Check the data types of each column
print("Data types:", chicago.dtypes)

# View the spatial extent (bounding box) of the shapes
print("Bounding box:", chicago.total_bounds)

# Check the coordinate reference system (CRS)
print("CRS:", chicago.crs)
```

```
Columns in the shapefile: Index(['area', 'area_num_1', 'area_numbe', 'comarea', 'comarea_id',
       'community', 'perimeter', 'shape_area', 'shape_len', 'geometry'],
      dtype='object')
Data types: area           float64
area_num_1        object
area_numbe        object
comarea          float64
comarea_id        float64
community        object
perimeter         float64
shape_area        float64
shape_len         float64
geometry          geometry
dtype: object
Bounding box: [-87.94011408  41.64454312 -87.5241371   42.02303859]
CRS: EPSG:4326
```

To enhance the geospatial plot, we'll use the shapefile as a background to provide context for Chicago's community areas. On top of that, we'll layer points of interest, such as restaurants,

and shops, to illustrate the city's amenities. This approach will make the map more informative and visually engaging, with community boundaries as the foundation and key locations overlayed to highlight areas of interest.

Next, we will add the Divvy bicycle stations on top of the chicago shapefile

#### 11.4.2 Dataset: Bicycle Sharing in Chicago

Divvy is Chicagoland's bike share system (in collaboration with Chicago Department of Transportation), with 6,000 bikes available at 570+ stations across Chicago and Evanston. Divvy provides residents and visitors with a convenient, fun and affordable transportation option for getting around and exploring Chicago.

Divvy, like other bike share systems, consists of a fleet of specially designed, sturdy and durable bikes that are locked into a network of docking stations throughout the region. The bikes can be unlocked from one station and returned to any other station in the system. People use bike share to explore Chicago, commute to work or school, run errands, get to appointments or social engagements, and more.

Divvy is available for use 24 hours/day, 7 days/week, 365 days/year, and riders have access to all bikes and stations across the system.

We will be using divvy trips in the year of 2013

```
# read the csv file'divvy_2013.csv' into pandas pandas dataframe
data = pd.read_csv('datasets/divvy_2013.csv')
data.head()
```

	trip_id	usertype	gender	starttime	stoptime	tripduration	from_station_id
0	3940	Subscriber	Male	2013-06-27 01:06:00	2013-06-27 09:46:00	31177	91
1	4095	Subscriber	Male	2013-06-27 12:06:00	2013-06-27 12:11:00	301	85
2	4113	Subscriber	Male	2013-06-27 11:09:00	2013-06-27 11:11:00	140	88
3	4118	Customer	NaN	2013-06-27 12:11:00	2013-06-27 12:16:00	316	85
4	4119	Subscriber	Male	2013-06-27 11:12:00	2013-06-27 11:13:00	87	88

In the Divvy dataset, each trip record includes the latitude and longitude coordinates of both the pickup and drop-off locations, which correspond to Divvy bike stations. These coordinates allow us to map the precise locations of each station, making it possible to visually display the network of Divvy stations across the city. By plotting these stations on a map, we can better understand the geographic distribution and accessibility of Divvy's bike-sharing network.

Below are the basic data cleaning steps to extract the coordinates of the Divvy stations.

```
# drop the duplicates in the column 'to_station_id', 'to_station_name', 'latitude_end', 'longitude_end'
# data_station_same = data[['from_station_id', 'from_station_name', 'latitude_start', 'longitude_start']]
# data_station_same.shape
```

### 11.4.3 Adding the divvy station to the plot

Once the coordinates are prepared, we'll add them as scatter plots on top of the Chicago shapefile

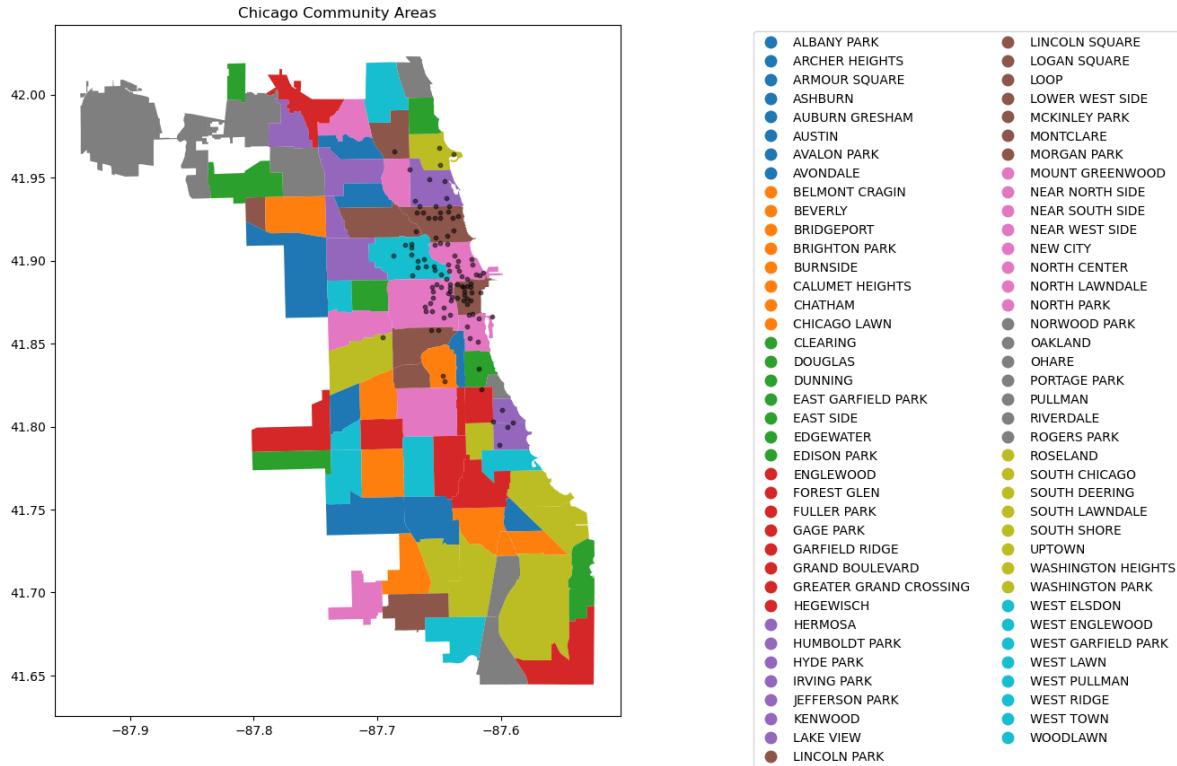
```
# Adding the stations to the plot
fig, ax = plt.subplots(figsize=(15, 10))

chicago = gpd.read_file(r'datasets/chicago_boundaries\geo_export_26bce2f2-c163-42a9-9329-9ca
chicago.plot(column='community', ax=ax, legend=True, legend_kwds={'ncol': 2, 'bbox_to_anchor': [0.5, 0.5, 1.0, 1.0]})

# Plot the stations
longlat_df = data[['latitude_start', 'longitude_start']].drop_duplicates()

plt.scatter(longlat_df['longitude_start'], longlat_df['latitude_start'], s=10, alpha=0.5, color='red')

# Add title (optional)
plt.title('Chicago Community Areas');
```



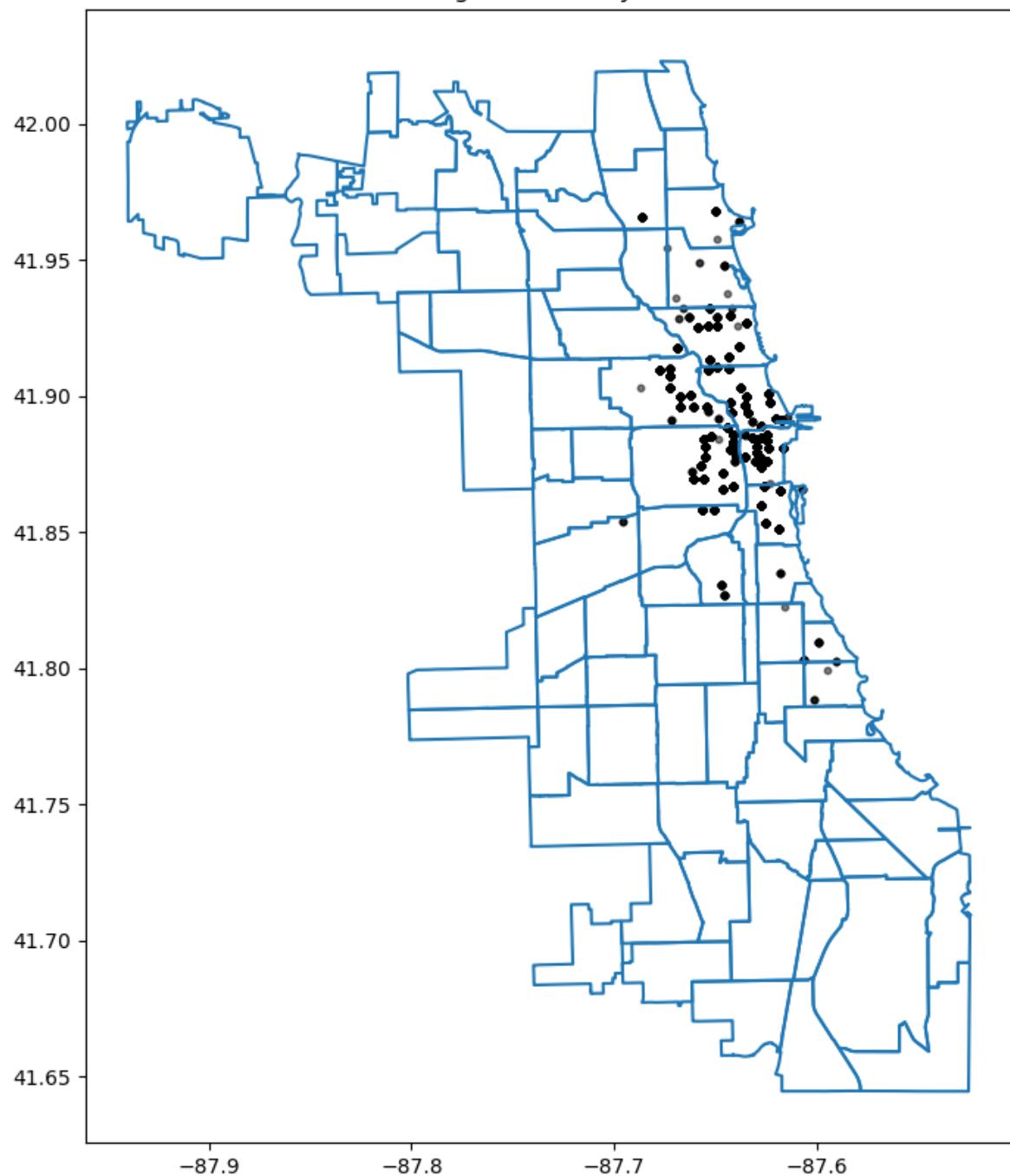
#### 11.4.4 Change the chicago shapefile

Using a different Chicago shapefile from GeoDa is a great way to observe how geographic boundaries or data details may vary

```
chicago = gpd.read_file(geodatasets.get_path("geoda.chicago_commpop"))

# Plot the data
fig, ax = plt.subplots(figsize=(15, 10))
chicago.boundary.plot(ax=ax)
plt.scatter(data['longitude_start'], data['latitude_start'], s=10, alpha=0.5, color='black',
plt.title('Chicago Community Areas');
```

Chicago Community Areas



## 11.4.5 Interactive Plotting

Alongside static plots, `geopandas` can create interactive maps based on the `folium` library.

Creating maps for interactive exploration mirrors the API of static `plots` in an `explore()` method of a GeoSeries or GeoDataFrame.

Here's an explanation of how `explore()` works and its key features:

Key Features of `explore()`:

1. Interactive Map Display:

- When you call `explore()` on a Geodataframe (gdf), it launches an interactive map widget directly within your Jupyter notebook.
- This map allows you to pan, zoom, and interact with the geometries (points, lines, polygons) in your Geodataframe.

2. Layer Control:

- `explore()` automatically adds the geometries from your Geodataframe as layers on the map.
- Each geometry type (points, lines, polygons) is displayed with appropriate styling and markers.

3. Tooltip Information:

- When you hover over a geometry in the map, `explore()` displays tooltip information that typically includes attribute data associated with that geometry.
- This feature is useful for inspecting specific details or properties of individual features in your geospatial dataset.

4. Search and Filter:

- `explore()` provides basic search and filter functionalities directly on the map.
- You can search for specific attribute values or filter the displayed features based on attribute criteria defined in your Geodataframe.

5. Customization:

- Although `explore()` provides default styling and interaction behaviors, you can customize the map further using parameters or by manipulating the Geodataframe before calling `explore()`.

```
# use the geopandas explore default settings
chicago = gpd.read_file(geodatasets.get_path("geoda.chicago_commpop"))

chicago.explore()
```

```
<folium.folium.Map at 0x1f6a6f672c0>
```

Adding the population layer

```
# Customerize the explore settings
chicago = gpd.read_file(geodatasets.get_path("geoda.chicago_commpop"))

m = chicago.explore(
    column="POP2010", # make choropleth based on "POP2010" column
    scheme="naturalbreaks", # use mapclassify's natural breaks scheme
    legend=True, # show legend
    k=10, # use 10 bins
    tooltip=False, # hide tooltip
    popup=["POP2010", "POP2000"], # show popup (on-click)
    legend_kwds=dict(colorbar=False), # do not use colorbar
    name="chicago", # name of the layer in the map
)
m
```

```
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1446: 
  warnings.warn(
```

```
<folium.folium.Map at 0x1f6a7436e40>
```

The `explore()` method returns a `folium.Map` object, which can also be passed directly (as you do with `ax` in `plot()`). You can then use folium functionality directly on the resulting map. Next, let's add the divvy station plot.

```
type(m)
```

```
folium.folium.Map
```

#### 11.4.6 Adding the divvy station on the interactive Folium.Map

We need to extract the station information from the trip dataset and add description to the station. You can skip this part

```
# Helper function for adding the description to the station
def row_to_html(row):
    row_df = pd.DataFrame(row).T
    row_df.columns = [col.capitalize() for col in row_df.columns]
    return row_df.to_html(index=False)

# Extracting the latitude, longitude, and station name for plotting, and also counting the number of trips
grouped_df = data.groupby(['from_station_name', 'latitude_start', 'longitude_start'])['trip_id'].count()
display(grouped_df.sort_values('trip_id', ascending=False).head())
grouped_df.rename(columns={'from_station_name':'title', 'latitude_start':'latitude', 'longitude_start':'longitude'})
grouped_df['description'] = grouped_df.apply(lambda row: row_to_html(row), axis=1)
geometry = gpd.points_from_xy(grouped_df['longitude'], grouped_df['latitude'])
geo_df = gpd.GeoDataFrame(grouped_df, geometry=geometry)
# Optional: Assign Coordinate Reference System (CRS)
geo_df.crs = "EPSG:4326" # WGS84 coordinate system
geo_df.head()
```

	from_station_name	latitude_start	longitude_start	trip_id
75	Millennium Park	41.881032	-87.624084	207
54	Lake Shore Dr & Monroe St	41.881050	-87.616970	191
72	Michigan Ave & Oak St	41.900960	-87.623777	186
68	McClurg Ct & Illinois St	41.891020	-87.617300	177
73	Michigan Ave & Pearson St	41.897660	-87.623510	127

	title	latitude	longitude	count	description
0	Aberdeen St & Jackson Blvd	41.877726	-87.654787	28	<table border="1" class="dataframe">\ <tr><td>Aberdeen St & Jackson Blvd</td><td>41.877726</td><td>-87.654787</td><td>28</td><td><table border="1" class="dataframe">\ <tr><td>Aberdeen St & Madison St</td><td>41.881487</td><td>-87.654752</td><td>28</td><td><table border="1" class="dataframe">\ <tr><td>Adler Planetarium</td><td>41.866095</td><td>-87.607267</td><td>6</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Armitage Ave</td><td>41.917859</td><td>-87.668919</td><td>20</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Augusta Blvd</td><td>41.899643</td><td>-87.667700</td><td>27</td><td>
1	Aberdeen St & Madison St	41.881487	-87.654752	28	<table border="1" class="dataframe">\ <tr><td>Aberdeen St & Madison St</td><td>41.881487</td><td>-87.654752</td><td>28</td><td><table border="1" class="dataframe">\ <tr><td>Adler Planetarium</td><td>41.866095</td><td>-87.607267</td><td>6</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Armitage Ave</td><td>41.917859</td><td>-87.668919</td><td>20</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Augusta Blvd</td><td>41.899643</td><td>-87.667700</td><td>27</td><td>
2	Adler Planetarium	41.866095	-87.607267	6	<table border="1" class="dataframe">\ <tr><td>Adler Planetarium</td><td>41.866095</td><td>-87.607267</td><td>6</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Armitage Ave</td><td>41.917859</td><td>-87.668919</td><td>20</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Augusta Blvd</td><td>41.899643</td><td>-87.667700</td><td>27</td><td>
3	Ashland Ave & Armitage Ave	41.917859	-87.668919	20	<table border="1" class="dataframe">\ <tr><td>Ashland Ave & Armitage Ave</td><td>41.917859</td><td>-87.668919</td><td>20</td><td><table border="1" class="dataframe">\ <tr><td>Ashland Ave & Augusta Blvd</td><td>41.899643</td><td>-87.667700</td><td>27</td><td>
4	Ashland Ave & Augusta Blvd	41.899643	-87.667700	27	<table border="1" class="dataframe">\ <tr><td>Ashland Ave & Augusta Blvd</td><td>41.899643</td><td>-87.667700</td><td>27</td><td>

We can add a hover tooltip (sometimes referred to as a tooltip or tooltip popup) for each point on your Folium map. This tooltip will appear when you hover over the markers on the map, providing additional information without needing to click on them. Here's how you can modify your existing code to include hover tooltips:

```

chicago = gpd.read_file(geodatasets.get_path("geoda.chicago_commpop"))

m = chicago.explore(
    column="POP2010", # make choropleth based on "POP2010" column
    scheme="naturalbreaks", # use mapclassify's natural breaks scheme
    legend=True, # show legend
    k=10, # use 10 bins
    tooltip=False, # hide tooltip
    popup=["POP2010", "POP2000"], # show popup (on-click)
    legend_kwds=dict(colorbar=False), # do not use colorbar
    name="chicago", # name of the layer in the map
)

geo_df.explore(
    m=m, # pass the map object
    color="red", # use red color on all points
    marker_kwds=dict(radius=5, fill=True), # make marker radius 10px with fill
    tooltip="description", # show "name" column in the tooltip
    tooltip_kwds=dict(labels=False), # do not show column label in the tooltip
    name="divstation", # name of the layer in the map
)

m

```

c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\\_kmeans.py:1446: UserWarning: Optimal results were not found because fewer than 2 clusters were found.

<folium.folium.Map at 0x1f6a369ddf0>

## 11.5 Independent Study

### 11.5.1 Multiple plots in a single figure using Seaborn

**Purpose:** Histogram and density plots visualize the distribution of a continuous variable.

A histogram plots the number of observations occurring within discrete, evenly spaced bins of a random variable, to visualize the distribution of the variable. It may be considered a special case of a bar plot as bars are used to plot the observation counts.

A density plot uses a kernel density estimate to approximate the distribution of random variable.

Using the tips\_df dataset

**11.5.1.1**

Make a histogram showing the distributions of total bill on each day of the week

**11.5.1.2**

Make a density plot showing the distributions of total bills on each day.

## **Part IV**

# **From messy to insight**

# 12 Data Cleaning and Preparation

## 12.1 Handling missing data

Missing values in a dataset can occur due to several reasons such as breakdown of measuring equipment, accidental removal of observations, lack of response by respondents, error on the part of the researcher, etc.

Let us read the dataset *GDP\_missing\_data.csv*, in which we have randomly removed some values, or put missing values in some of the columns.

We'll also read *GDP\_complete\_data.csv*, in which we have not removed any values. We'll use this data later to assess the accuracy of our guess or estimate of missing values in *GDP\_missing\_data.csv*.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
import seaborn as sns

sns.set(font_scale=1.5)

%matplotlib inline

gdp_missing_values_data = pd.read_csv('./Datasets/GDP_missing_data.csv')
gdp_complete_data = pd.read_csv('./Datasets/GDP_complete_data.csv')

gdp_missing_values_data.head()
```

	economicActivityFemale	country	lifeMale	infantMortality	gdpPerCapita	economicActivityMale
0	7.2	Afghanistan	45.0	154.0	2474.0	87.5
1	7.8	Algeria	67.5	44.0	11433.0	76.4
2	41.3	Argentina	69.6	22.0	NaN	76.2
3	52.0	Armenia	67.2	25.0	13638.0	65.0

	economicActivityFemale	country	lifeMale	infantMortality	gdpPerCapita	economicActivityMale
4	53.8	Australia	NaN	6.0	54891.0	NaN

Observe that the `gdp_missing_values_data` dataset consists of some missing values shown as `NaN` (Not a Number).

### 12.1.1 Identifying missing values in a dataframe

There are multiple ways to identify missing values in a dataframe

#### 12.1.1.1 `describe()` Method

Note that the descriptive statistics methods associated with Pandas objects ignore missing values by default. Consider the summary statistics of `gdp_missing_values_data`:

```
gdp_missing_values_data.describe()
```

	economicActivityFemale	lifeMale	infantMortality	gdpPerCapita	economicActivityMale	illit
count	145.000000	145.000000	145.000000	145.000000	145.000000	145.000000
mean	45.935172	65.491724	37.158621	24193.482759	76.563448	13.5
std	16.875922	9.099256	34.465699	22748.764444	7.854730	16.4
min	1.900000	36.000000	3.000000	772.000000	51.200000	0.00
25%	35.500000	62.900000	10.000000	6837.000000	72.000000	1.00
50%	47.600000	67.800000	24.000000	15184.000000	77.300000	6.60
75%	55.900000	72.400000	54.000000	35957.000000	81.600000	19.5
max	90.600000	77.400000	169.000000	122740.000000	93.000000	70.5

Observe that the `count` statistics report the number of non-missing values of each column in the data, as the number of rows in the data (see code below) is more than the number of non-missing values of all the variables in the above table. Similarly, for the rest of the statistics, such as `mean`, `std`, etc., the missing values are ignored.

```
#The dataset gdp_missing_values_data has 155 rows
gdp_missing_values_data.shape[0]
```

### 12.1.1.2 info() Method

Shows the count of non-null entries in each column, helping you quickly identify columns with missing values.

```
gdp_missing_values_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 155 entries, 0 to 154
Data columns (total 12 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   economicActivityFemale  145 non-null    float64
 1   country              155 non-null    object  
 2   lifeMale              145 non-null    float64
 3   infantMortality      145 non-null    float64
 4   gdpPerCapita          145 non-null    float64
 5   economicActivityMale  145 non-null    float64
 6   illiteracyMale         145 non-null    float64
 7   illiteracyFemale       145 non-null    float64
 8   lifeFemale             145 non-null    float64
 9   geographic_location   155 non-null    object  
 10  contraception          84 non-null    float64
 11  continent              155 non-null    object  
dtypes: float64(9), object(3)
memory usage: 14.7+ KB
```

### 12.1.1.3 isnull() Method

This is one of the most direct methods. Using `df.isnull()` returns a DataFrame of Boolean values where True indicates a missing value. To get a summary, you can use `df.isnull().sum()` to see the count of missing values in each column.

For finding the number of missing values in each column of `gdp_missing_values_data`, we will sum up the missing values in each column of the dataset:

```
gdp_missing_values_data.isnull().sum()
```

```
economicActivityFemale    10
country                  0
lifeMale                 10
```

```

infantMortality          10
gdpPerCapita              10
economicActivityMale      10
illiteracyMale             10
illiteracyFemale           10
lifeFemale                 10
geographic_location        0
contraception               71
continent                   0
dtype: int64

```

### 12.1.2 Types of Missing Values

In data science, missing values typically fall into three main types, each requiring different handling strategies:

#### 12.1.2.1 Missing Completely at Random (MCAR)

- **Definition:** Missing values are entirely independent of any variables in the dataset.
- **Example:** A respondent accidentally skips a question on a survey.
- **Impact:** MCAR data can usually be ignored or imputed without biasing the analysis.
- **Handling:** Simple imputation methods, like filling with mean or median values, are often appropriate.

#### 12.1.2.2 Missing at Random (MAR)

- **Definition:** The likelihood of a value being missing is related to other observed variables but not to the missing data itself.
- **Example:** People with higher incomes may be less likely to report their spending, but income data itself is not missing.
- **Impact:** Ignoring MAR values may bias results, so careful imputation based on related variables is recommended.
- **Handling:** More complex imputation methods, like conditional mean imputation or predictive modeling, are suitable.

#### 12.1.2.3 Missing Not at Random (MNAR)

- **Definition:** The probability of missingness is related to the missing data itself, meaning the value is systematically missing.

- **Example:** Patients with severe health conditions might be less likely to report their health status, or students with low scores may be less likely to submit their grades.
- **Impact:** MNAR is the most challenging type, as missing values may introduce significant bias.
- **Handling:** Solutions often include sensitivity analysis, data augmentation, or modeling techniques that account for the missing mechanism, though sometimes domain-specific approaches are necessary.

Understanding the type of missing data helps in selecting the right imputation method and mitigating potential biases in the analysis.

#### **12.1.2.4 Questions**

##### **12.1.2.4.1**

Why can we ignore observations with missing values without risking skewing the analysis or trends in the case of **Missing Completely at Random (MCAR)**?

##### **12.1.2.4.2**

Why could ignoring missing values lead to biased results for **Missing at Random (MAR)** and **Missing Not at Random (MNAR)** data?

##### **12.1.2.4.3**

For the dataset consisting of GDP per capita, think of hypothetical scenarios in which the missing values of GDP per capita can correspond to **MCAR / MAR / MNAR**.

#### **12.1.3 Methods for Handling missing values**

##### **12.1.3.1 Removing Missing values**

- **Row/Column Removal:** Use [df.dropna()] (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html>)
  - When to Use: if the missing values are few or the rows/columns are not critical.
  - Risks: Can reduce the dataset's size, potentially losing valuable information.

Let us drop the rows containing even a single value from gdp\_missing\_values\_data.

```
gdp_no_missing_data = gdp_missing_values_data.dropna()
```

By default, `df.dropna()` will drop any row that contains at least one missing value, leaving only the rows that are completely free of missing values.

```
#Shape of gdp_no_missing_data  
gdp_no_missing_data.shape
```

(42, 12)

Using `df.dropna()` to remove rows with missing values can sometimes lead to a significant reduction in data, which can be problematic if much of the data is valuable and non-missing. For example:

- **Impact of Default Behavior:** Dropping rows with even a single missing value reduced the number of rows from 155 to 42! This drastic reduction happens because, by default, `dropna()` removes any row with at least one missing value, keeping only rows that are completely complete.
- **Loss of Non-Missing Data:** Even though some columns may have very few missing values (e.g., at most 10), using `dropna()` without modification results in losing all non-missing data in affected rows. This is typically a poor choice, especially when valuable, non-missing data is removed unnecessarily.

To avoid losing too much data, you can adjust the behavior of `dropna()` using these parameters:

- **how Parameter**
  - Controls the criteria for dropping rows or columns.
  - `how='any'` (default): Drops rows or columns if any values are missing.
  - `'how='all'`: Drops rows or columns only if all values are missing
- **thresh Parameter**
  - Sets a minimum number of non-missing values required to retain the row or column.
  - Useful when you want to keep rows or columns with substantial, but not complete, data.

If a few values of a column are missing, we can possibly estimate them using the rest of the data, so that we can (hopefully) maximize the information that can be extracted from the data. However, if most of the values of a column are missing, it may be harder to estimate its values.

In this dataset, we see that around 50% values of the `contraception` column is missing. Thus, we'll drop the column as it may be hard to impute its values based on a relatively small number of non-missing values.

```
#Deleting column with missing values in almost half of the observations
gdp_missing_values_data.drop(['contraception'],axis=1,inplace=True)
gdp_missing_values_data.shape
```

(155, 11)

### 12.1.3.2 Imputing Missing values

There are an unlimited number of ways to impute missing values. Some imputation methods are provided in the [Pandas documentation](#).

The best way to impute them will depend on the problem:

- **For MCAR:** Simple imputation is generally acceptable.
- **For MAR:** Imputation should consider relationships with other variables, such as using conditional mean imputation.
- **For MNAR:** Imputation requires careful analysis, and domain knowledge is essential to avoid bias.

Below are some of the most common methods. Recall that we randomly introduced missing values in `gdp_missing_values_data`, while the actual values are preserved in `gdp_complete_data`.

We will apply these methods to `gdp_missing_values_data`. To evaluate each method's effectiveness in imputing missing values, we'll compare the imputed `gdpPerCapita` values with the actual values and calculate the Root Mean Square Error (RMSE).

To visualize the imputed vs actual values, let's define a helper function

```
#Index of rows with missing values for GDP per capita
null_ind_gdpPC = gdp_missing_values_data.index[gdp_missing_values_data.gdpPerCapita.isnull()]

#Defining a function to plot the imputed values vs actual values
def plot_actual_vs_predicted(y):
    fig, ax = plt.subplots(figsize=(8, 6))
    plt.rc('xtick', labelsize=15)
    plt.rc('ytick', labelsize=15)
    x = gdp_complete_data.loc>null_ind_gdpPC,'gdpPerCapita']
    # y = y.loc>null_ind_gdpPC,'gdpPerCapita']
    plt.scatter(x,y.loc>null_ind_gdpPC,'gdpPerCapita'])
    z=np.polyfit(x,y.loc>null_ind_gdpPC,'gdpPerCapita'],1)
    p=np.poly1d(z)
    plt.plot(x,x,color='orange')
```

```

plt.xlabel('Actual GDP per capita', fontsize=15)
plt.ylabel('Imputed GDP per capita', fontsize=15)
ax.xaxis.set_major_formatter('${x:.0f}')
ax.yaxis.set_major_formatter('${x:.0f}')
plt.rc('axes', labelsize=10) # Set all axis labels to fontsize 10
# plt.title('Actual vs Imputed values for GDP per capita', fontsize=20)
rmse = np.sqrt(np.mean((y.loc>null_ind_gdpPC, 'gdpPerCapita'])-x)**2))
plt.text(10000, 50000, 'RMSE = %.2f' % rmse, fontsize=15)

```

### 12.1.3.2.1 Mean/Median/Mode Imputation

This approach replaces missing values with the mean or median of the column.

Let's impute missing values in the column by substituting them with the average of the non-missing values. Imputing with the mean tends to minimize the sum of squared differences between actual values and imputed values, especially in cases where data is Missing Completely at Random (MCAR). However, this might not hold true for other types of missing data, such as Missing at Random (MAR) or Missing Not at Random (MNAR).

Let us impute missing values in the column as the average of the non-missing values of the column. The sum of squared differences between actual values and the imputed values is likely to be smaller if we impute using the mean. However, this may not be true in cases other than MCAR (Missing completely at random).

```

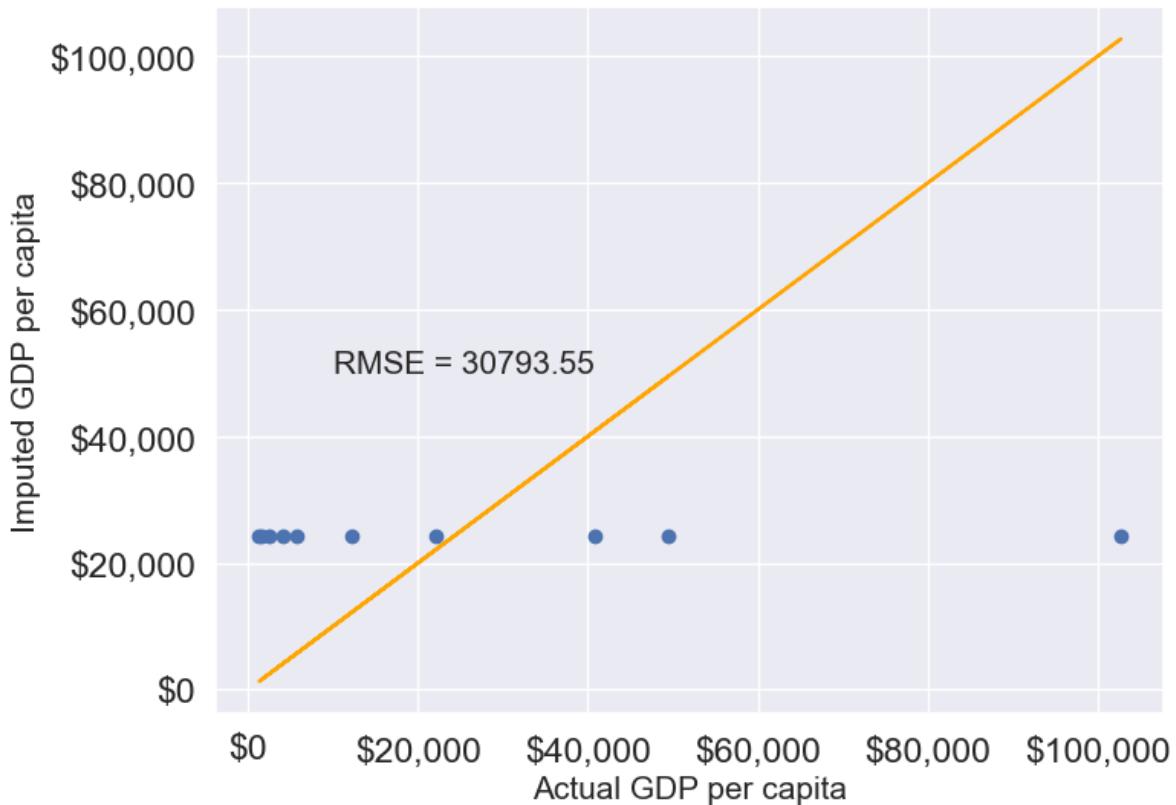
# Extracting the columns with missing values
columns_with_missing = [col for col in gdp_missing_values_data.columns if gdp_missing_values_
columns_with_missing

['economicActivityFemale',
 'lifeMale',
 'infantMortality',
 'gdpPerCapita',
 'economicActivityMale',
 'illiteracyMale',
 'illiteracyFemale',
 'lifeFemale']

# Imputing missing values using the mean
gdp_imputed_data_mean = gdp_missing_values_data[columns_with_missing].fillna(gdp_missing_val

```

```
plot_actual_vs_predicted(gdp_imputed_data_mean)
```



Here, since all columns with missing values are numerical, we could use the mean to impute these values. Using the mean is generally suitable only for numerical data, as it represents a central tendency specific to numbers. For categorical data, however, mean imputation would be inappropriate. Instead, the mode, which identifies the most frequently occurring value, is a more suitable choice for imputing missing values in categorical columns.

#### 12.1.3.2.2 Conditional Imputation: Use other related variables to predict missing values

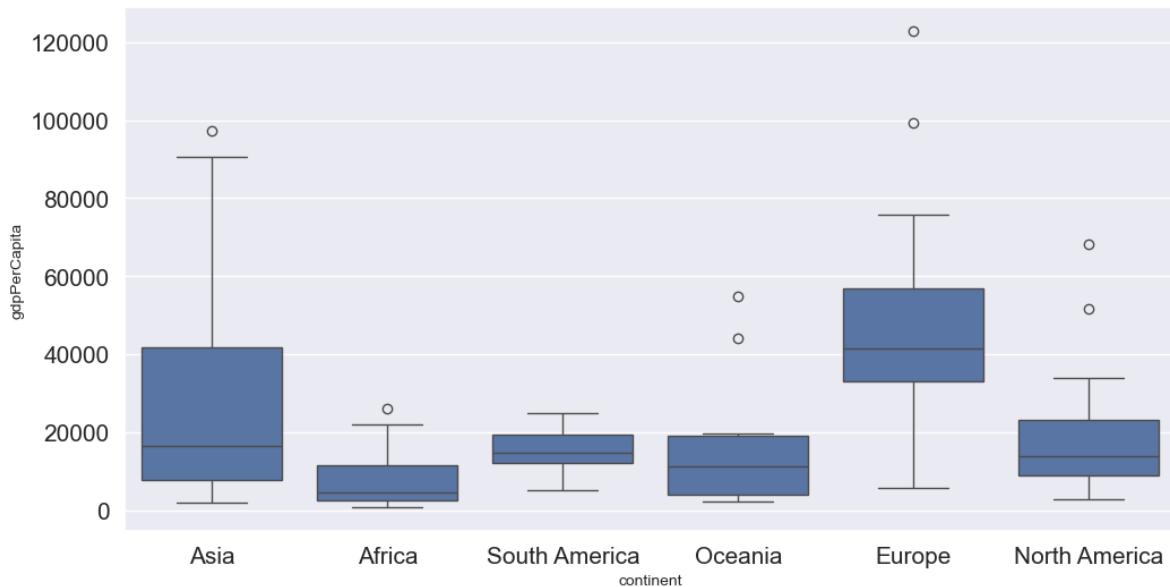
There are three approaches within this category; please see the details below.

- **Imputing missing values based on correlated variables in the data**

If a variable is highly correlated with another variable in the dataset, we can approximate its missing values using the trendline with the highly correlated variable.

Let us visualize the distribution of GDP per capita for different continents.

```
plt.rcParams["figure.figsize"] = (12,6)
sns.boxplot(x = 'continent',y='gdpPerCapita',data = gdp_missing_values_data);
```



We observe that there is a distinct difference between the GDPs per capita of some of the continents. Let us impute the missing GDP per capita of a country as the mean GDP per capita of the corresponding continent. This imputation should be better than imputing the missing GDP per capita as the mean of all the non-missing values, as the GDP per capita of a country is likely to be closer to the mean GDP per capita of the continent, rather than the mean GDP per capita of the whole world.

```
#Finding the mean GDP per capita of the continent - please defer the understanding of this code
avg_gdpPerCapita = gdp_missing_values_data['gdpPerCapita'].groupby(gdp_missing_values_data['continent']).mean()
```

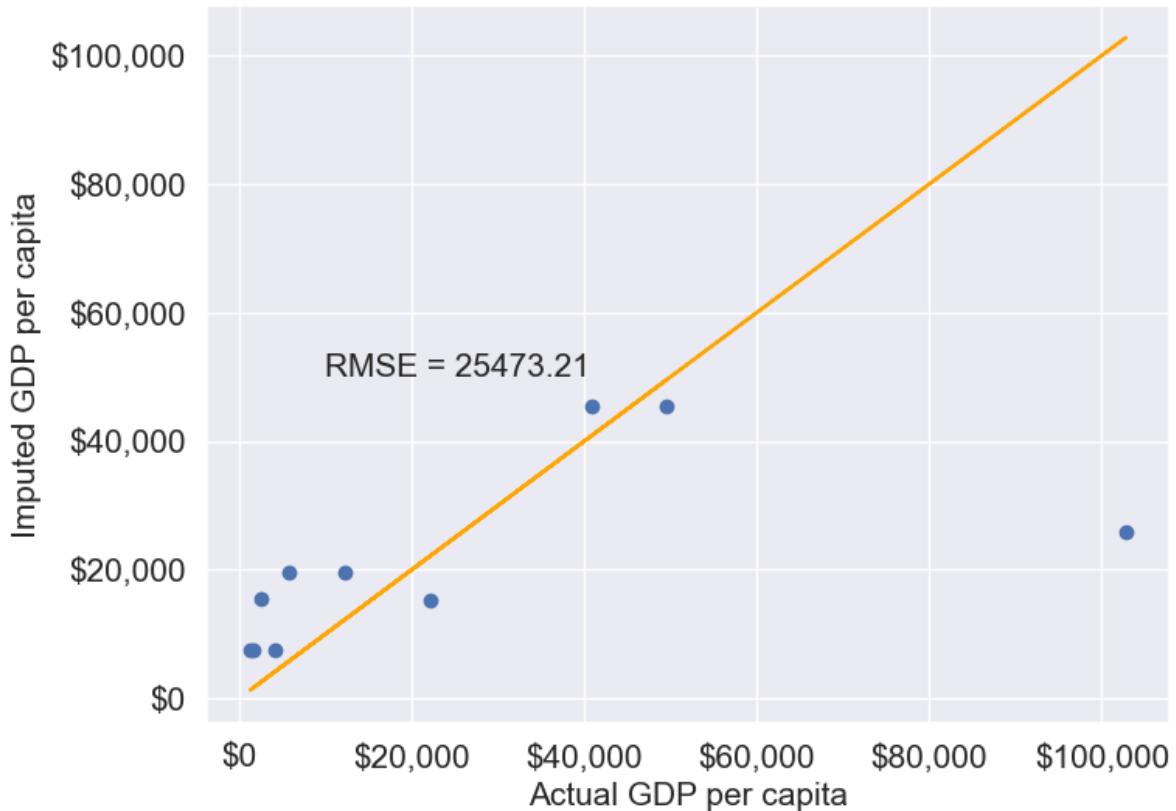
continent	avg_gdpPerCapita
Africa	7638.178571
Asia	25922.750000
Europe	45455.303030
North America	19625.210526
Oceania	15385.857143
South America	15360.909091

Name: gdpPerCapita, dtype: float64

```
#Creating a copy of missing data to impute missing values
gdp_imputed_data_group_mean = gdp_missing_values_data.copy()
```

```
#Replacing missing GDP per capita with the mean GDP per capita for the corresponding continent
```

```
gdp_imputed_data_group_mean.gdpPerCapita = \
gdp_imputed_data_group_mean['gdpPerCapita'].fillna(gdp_imputed_data_group_mean['continent'].mean())
plot_actual_vs_predicted(gdp_imputed_data_group_mean)
```



Note that the imputed values are closer to the actual values, and the RMSE has further reduced as expected.

Suppose we wish to impute the missing values of each numeric column with the average of the non-missing values of the respective column corresponding to the continent of the observation. The above logic can be extended to each column as shown in the code below.

```
all_columns_imputed_data = gdp_missing_values_data.iloc[:,[0,2,3,4,5,6,7,8]].apply(lambda x:
x.fillna(gdp_imputed_data_group_mean['continent']).map(x.groupby(gdp_missing_values_data['continent'])))
```

## Using Regression

```
#Let us identify the variable highly correlated with GDP per capita.
```

```
gdp_missing_values_data.select_dtypes(include='number').corrwith(gdp_missing_values_data.gdpP
```

```
economicActivityFemale    0.078332
lifeMale                  0.579850
infantMortality          -0.572201
gdpPerCapita              1.000000
economicActivityMale      -0.134108
illiteracyMale             -0.479143
illiteracyFemale           -0.448273
lifeFemale                 0.615954
dtype: float64
```

```
#The variable *lifeFemale* has the strongest correlation with GDP per capita. Let us use it to impute missing values.
```

```
# Extract the variables lifeFemale and GDP per capita
```

```
x = gdp_missing_values_data.lifeFemale
y = gdp_missing_values_data.gdpPerCapita
```

```
# Identify non-missing indices
```

```
idx_non_missing = np.isfinite(x) & np.isfinite(y)
```

```
# Fit a linear regression model (degree=1) to predict GDP per capita given lifeFemale
```

```
slope_intercept_trendline = np.polyfit(x[idx_non_missing],y[idx_non_missing],1) #Finding the slope and intercept of the trendline
```

```
compute_y_given_x = np.poly1d(slope_intercept_trendline)
```

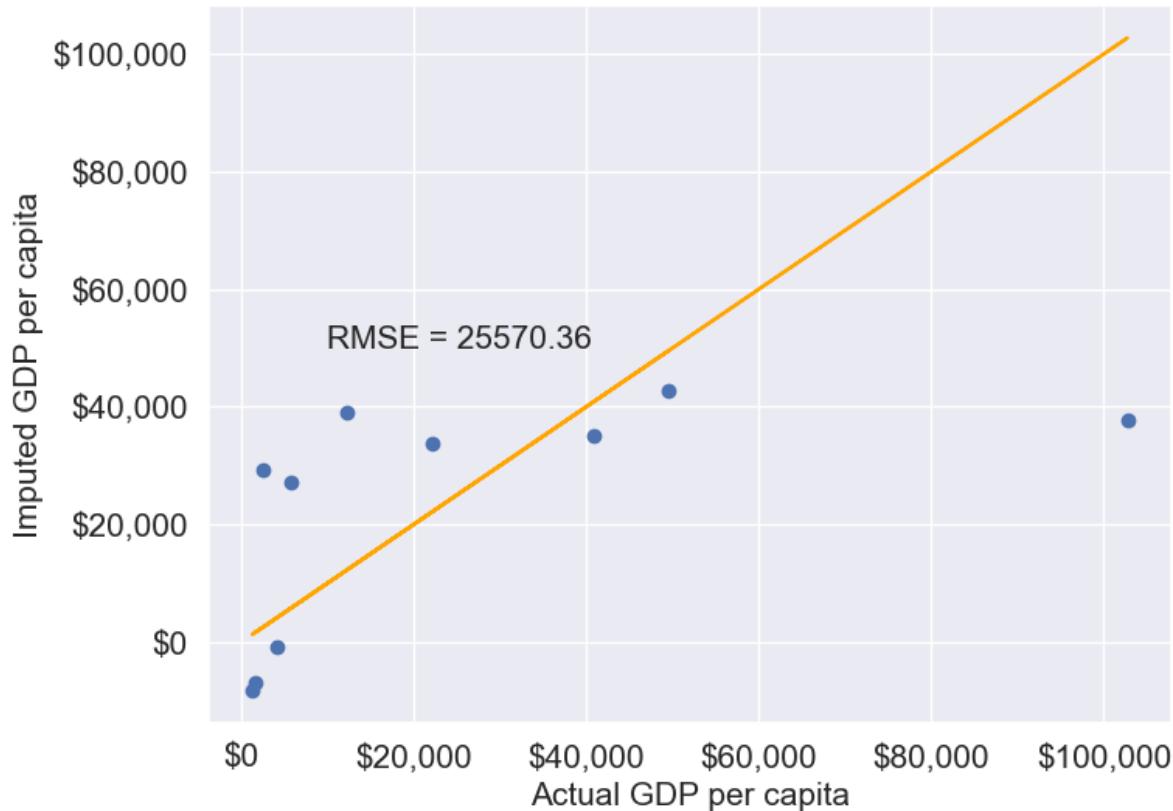
```
#Creating a copy of missing data to impute missing values
```

```
gdp_imputed_data_lr = gdp_missing_values_data.copy()
```

```
#Imputing missing values of GDP per capita using the linear regression model
```

```
gdp_imputed_data_lr.loc>null_ind_gdpPC,'gdpPerCapita']=compute_y_given_x(gdp_missing_values_data_lr.loc>null_ind_gdpPC,'lifeFemale')
```

```
plot_actual_vs_predicted(gdp_imputed_data_lr)
```



### KNN: K-nearest neighbor

In this method, we'll impute the missing value of the variable as the mean value of the  $K$ -nearest neighbors having non-missing values for that variable. The neighbors to a data-point are identified based on their Euclidean distance to the point in terms of the standardized values of rest of the variables in the data.

Let's consider a toy example to understand missing value imputation by KNN. Suppose we have to impute missing values in a toy dataset, named as `toy_data` having 4 observations and 3 variables.

```
#Toy example - A 4x3 array with missing values
nan = np.nan
toy_data = np.array([[1, 2, nan], [3, 4, 3], [nan, 6, 5], [8, 8, 7]])
toy_data
```

```
array([[ 1.,  2., nan],
       [ 3.,  4.,  3.],
       [nan,  6.,  5.],
       [ 8.,  8.,  7.]])
```

We'll use some functions from the *sklearn* library to perform the KNN imputation. It is much easier to directly use the algorithm from *sklearn*, instead of coding it from scratch.

```
#Library to compute pair-wise Euclidean distance between all observations in the data
from sklearn import metrics

#Library to impute missing values with the KNN algorithm
from sklearn import impute
```

We'll use the *sklearn* function `nan_euclidean_distances()` to compute the Euclidean distance between all pairs of observations in the data.

```
#This is the distance matrix containing the distance of the ith observation from the jth observation
metrics.pairwise.nan_euclidean_distances(toy_data,toy_data)
```

```
array([[ 0.          ,  3.46410162,  6.92820323, 11.29158979],
       [ 3.46410162,  0.          ,  3.46410162,  7.54983444],
       [ 6.92820323,  3.46410162,  0.          ,  3.46410162],
       [11.29158979,  7.54983444,  3.46410162,  0.          ]])
```

Note that the size of the above matrix is 4x4. This is because the  $(i, j)^{th}$  element of the matrix is the distance of the  $i^{th}$  observation from the  $j^{th}$  observation. The matrix is symmetric because the distance of  $i^{th}$  observation to the  $j^{th}$  observation is the same as the distance of the  $j^{th}$  observation to the  $i^{th}$  observation.

We'll use the *sklearn* function `KNNImputer()` to impute the missing value of a column in `toy_data` as the mean of the values of the  $K$  nearest neighbors to the observation that have non-missing values for that column.

Let us impute the missing values in `toy_data` using the values of  $K = 2$  nearest neighbors from the corresponding observation.

```
#imputing missing values with 2 nearest neighbors, where the neighbors have equal weights

#define an object of type KNNImputer
imputer = impute.KNNImputer(n_neighbors=2)

#Use the object method 'fit_transform' to impute missing values
imputer.fit_transform(toy_data)
```

```
array([[1. , 2. , 4. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

The third observation was the closest to the *2nd* and *4th* observations based on the Euclidean distance matrix. Thus, the missing value in the *3rd* row of the `toy_data` has been imputed as the mean of the values in the *2nd* and *4th* observations for the corresponding column. Similarly, the *1st* observation is the closest to the *2nd* and *3rd* observations. Thus the missing value in the *1st* row of `toy_data` has been imputed as the mean of the values in the *1st* and *2nd* observations for the corresponding column.

Let us use KNN to impute the missing values of `gdpPerCapita` in `gdp_missing_values_data`. We'll use only the numeric columns of the data in imputing the missing values. Also, we'll ignore `contraception` as it has a lot of missing values, and thus may not be useful.

```
#Considering numeric columns in the data to use KNN
num_cols = list(range(0,1))+list(range(2,9))
num_cols
```

```
[0, 2, 3, 4, 5, 6, 7, 8]
```

Before computing the pair-wise Euclidean distance of observations, we must standardize the data so that all columns are at the same scale. This will avoid columns with a higher magnitude of values having a higher weight in determining the Euclidean distance. Unless there is a reason to give a higher weight to a column, we assume all columns to have the same weight in the Euclidean distance computation.

We can use the code below to scale the data. However, after imputing the missing values, the data is to be scaled back to the original scale, so that each variable is in the same units as in the original dataset. However, if the code below is used, we'll lose the orginal scale of each of the columns.

```
#Scaling data to compute equally weighted distances from the 'k' nearest neighbors
scaled_data = gdp_missing_values_data.iloc[:,num_cols].apply(lambda x:(x-x.min())/(x.max()-x
```

To alleviate the problem of losing the orginal scale of the data, we'll use the `MinMaxScaler` object of the `sklearn` library. The object will store the original scale of the data, which will help transform the data back to the original scale once the missing values have been imputed in the standardized data.

```

# Scaling data - using sklearn

#Create an object of type MinMaxScaler
scaler = sk.preprocessing.MinMaxScaler()

#Use the object method 'fit_transform' to scale the values to a standard uniform distribution
scaled_data = pd.DataFrame(scaler.fit_transform(gdp_missing_values_data.iloc[:,num_cols]))

#Imputing missing values with KNNImputer

#Define an object of type KNNImputer
imputer = impute.KNNImputer(n_neighbors=3, weights="uniform")

#Use the object method 'fit_transform' to impute missing values
imputed_arr = imputer.fit_transform(scaled_data)

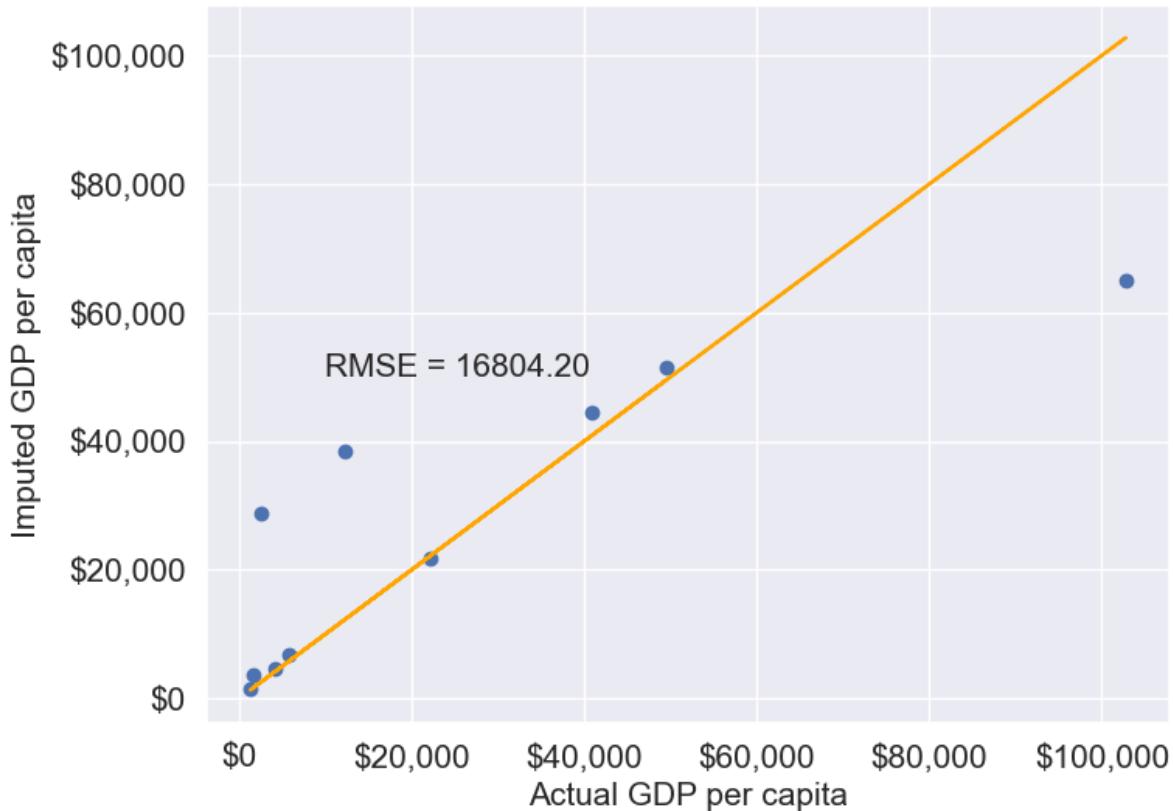
#Scaling back the scaled array to obtain the data at the original scale

#Use the object method 'inverse_transform' to scale back the values to the original scale of
unscaled_data = scaler.inverse_transform(imputed_arr)

#Note the method imputes the missing value of all the columns
#However, we are interested in imputing the missing values of only the 'gdpPerCapita' column
gdp_imputed_data_knn = gdp_missing_values_data.copy()
gdp_imputed_data_knn.loc[:, 'gdpPerCapita'] = unscaled_data[:,3]

#Visualizing the accuracy of missing value imputation with KNN
plot_actual_vs_predicted(gdp_imputed_data_knn)

```



Note that the RMSE is the lowest in this method. It is because this method imputes missing values as the average of the values of “similar” observations, which is smarter and more robust than the previous methods.

We chose  $K = 3$  in the missing value imputation for GDP per capita. However, the value of  $K$  is typically chosen using a method known as cross validation. We’ll learn about cross-validation in the next course of the sequence.

#### 12.1.3.2.3 Forward Fill/Backward Fill

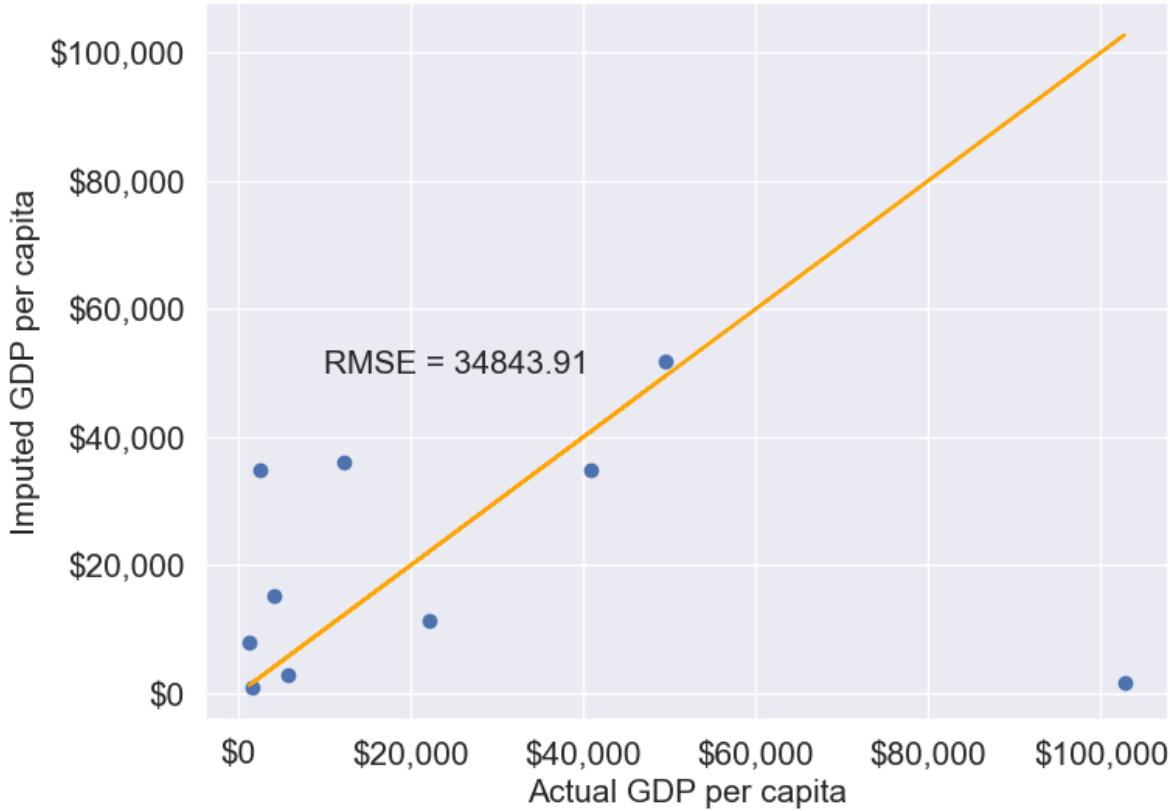
To fill missing values in a column by copying the value of the previous or next non-missing observation, we can use forward fill (propagating the last valid observation forward) or backward fill (propagating the next valid observation backward).

Below, we demonstrate forward fill. To perform backward fill instead, simply replace `ffill` with `bfill`

```
#Filling missing values: Method 1- Naive way
gdp_imputed_data_ffill = gdp_missing_values_data.ffill()
```

Let us next check how good is this method in imputing missing values.

```
plot_actual_vs_predicted(gdp_imputed_data_ffill)
```



We observe that the accuracy of imputation is poor as GDP per capita can vary a lot across countries, and the data is not sorted by GDP per capita. There is no reason why the GDP per capita of a country should be close to the GDP per capita of the country in the observation above it.

```
#Checking if any missing values are remaining
gdp_imputed_data_ffill.isnull().sum()
```

```
economicActivityFemale      0
country                      0
lifeMale                     0
infantMortality              0
gdpPerCapita                  0
economicActivityMale          0
```

```
illiteracyMale          1  
illiteracyFemale        0  
lifeFemale              0  
geographic_location    0  
continent               0  
dtype: int64
```

After imputing missing values, note there is still one missing value for *illiteracyMale*. Can you guess why one missing value remained?

## 12.2 Outlier detection and handling

An outlier is an observation that is significantly different from the rest of the data. Outlier detection and handling are important in data science because outliers can significantly impact the quality, accuracy, and interpretability of data analysis and model performance. Here are the main reasons why it's crucial to address outliers:

- **Preventing Skewed Results:** Outliers can distort statistical measures, such as the mean and standard deviation, leading to misleading interpretations of the data. For example, a few extreme values can inflate the mean, making the data seem larger than it is in reality.
- **Improving Model Accuracy:** Many machine learning algorithms are sensitive to outliers and can perform poorly if outliers are present. For instance, in linear regression, outliers can disproportionately affect the model by pulling the line of best fit toward them, reducing predictive accuracy.
- **Enhancing Robustness of Models:** Identifying and handling outliers can make models more robust and stable. By minimizing the influence of extreme values, models become less sensitive to noise, which improves generalization and reduces overfitting.

### 12.2.1 Outlier detection

Outlier detection is crucial for identifying unusual values in your data that may need to be investigated, removed, or transformed. Here are several popular methods for detecting outliers:

We will use *College.csv* that contains information about US universities as our dataset in this section. The description of variables of the dataset can be found on page 65 of this [book](#).

```
# Load the College dataset  
college = pd.read_csv('./Datasets/College.csv')  
college.head()
```

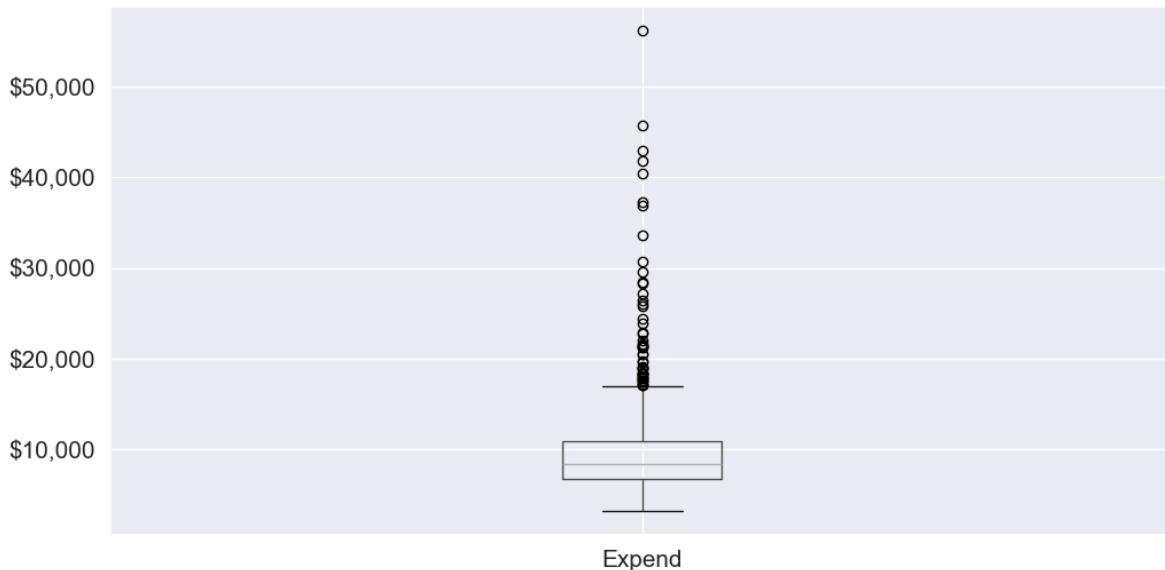
	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad
0	Abilene Christian University	Yes	1660	1232	721	23	52	2885
1	Adelphi University	Yes	2186	1924	512	16	29	2683
2	Adrian College	Yes	1428	1097	336	22	50	1036
3	Agnes Scott College	Yes	417	349	137	60	89	510
4	Alaska Pacific University	Yes	193	146	55	16	44	249

### 12.2.1.1 Visualization Methods

- **Box Plot:** A box plot is a quick and visual way to detect outliers, where points outside the “whiskers” are considered outliers.

Let us visualize outliers in average instructional expenditure per student given by the variable `Expend`.

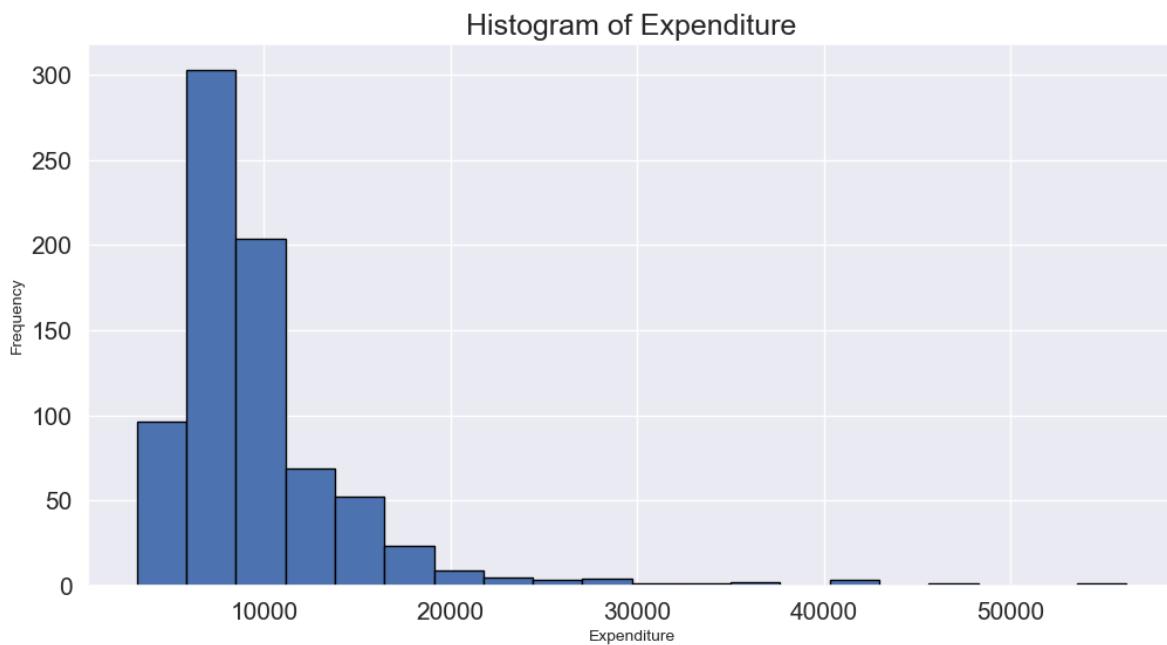
```
ax=college.boxplot(column = 'Expend')
ax.yaxis.set_major_formatter('${x:,.0f}')
```



There are several outliers (shown as circles in the above boxplot), which correspond to high values of average instructional expenditure per student.

- **Histogram:** Histograms can reveal unusual values as isolated bars, helping to identify outliers in a single feature.

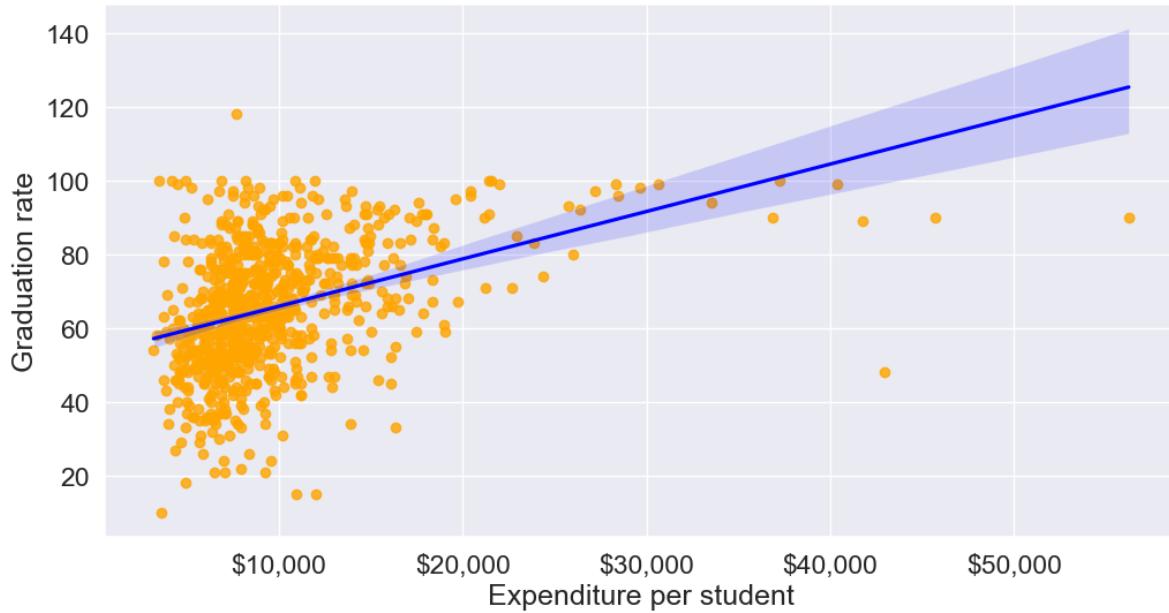
```
# create a histogram of expend
college['Expend'].plot(kind='hist', edgecolor='black', bins=20)
plt.xlabel('Expenditure')
plt.ylabel('Frequency')
plt.title('Histogram of Expenditure')
plt.show()
```



- **Scatter Plot:** A scatter plot can reveal outliers, especially in two-dimensional data. Outliers will often appear as points separated from the main cluster.

Let's make a scatterplot of 'Grad.Rate' vs 'Expend' with a trendline, to visualize potential outliers

```
sns.set(font_scale=1.5)
ax=sns.regplot(data = college, x = "Expend", y = "Grad.Rate", scatter_kws={"color": "orange"})
ax.xaxis.set_major_formatter('${x:,.0f}')
ax.set_xlabel('Expenditure per student')
ax.set_ylabel('Graduation rate')
plt.show()
```



The trendline indicates a positive correlation between `Expend` and `Grad.Rate`. However, there seems to be a lot of noise and presence of outliers in the data.

### 12.2.1.2 Statistical Methods

Visualization helps us identify potential outliers. To address them effectively, we need to extract these outlier instances using a defined threshold. Two commonly used statistical methods for this purpose are the Z-Score method and the Interquartile Range (IQR) (Tukey's fences).

#### 12.2.1.2.1 Z-Score (Standard Score)

Calculates how many standard deviations a data point is from the mean. Commonly, data points with a Z-score greater than 3 (or less than -3) are considered outliers.

```
from scipy import stats

# Calculate Z-scores and identify outliers
college['z_score'] = stats.zscore(college['Expend'])
college['is_z_score_outlier'] = college['z_score'].abs() > 3

# Filter to show only the outliers
z_score_outliers = college[college['is_z_score_outlier']]
print(z_score_outliers.shape)
z_score_outliers.head()
```

(16, 21)

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Und
20	Antioch University	Yes	713	661	252	25	44	712	23
144	Columbia University	Yes	6756	1930	871	78	96	3376	55
158	Dartmouth College	Yes	8587	2273	1087	87	99	3918	32
174	Duke University	Yes	13789	3893	1583	90	98	6188	53
191	Emory University	Yes	8506	4168	1236	76	97	5544	192

Boxplot identifies outliers based on the [Tukey's fences](#) criterion:

#### 12.2.1.2.2 IQR method ([Tukey's fences](#))

John Tukey proposed that observations outside the range  $[Q1 - 1.5(Q3 - Q1), Q3 + 1.5(Q3 - Q1)]$  are outliers, where  $Q1$  and  $Q3$  are the lower (25%) and upper (75%) quartiles respectively. Let us detect outliers based on this threshold.

```
# Calculate the first and third quartiles, and the IQR
Q1 = np.percentile(college['Expend'], 25)
Q3 = np.percentile(college['Expend'], 75)
IQR = Q3 - Q1

# Define the lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Add a new column to indicate IQR outliers
college['is_IQR_outlier'] = (college['Expend'] < lower_bound) | (college['Expend'] > upper_bound)

# Filter to show only the IQR outliers
iqr_outliers = college[college['is_IQR_outlier']]
print(iqr_outliers.shape)
iqr_outliers.head()
```

(48, 22)

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Und
3	Agnes Scott College	Yes	417	349	137	60	89	510	63

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad
16	Amherst College	Yes	4302	992	418	83	96	1593	5
20	Antioch University	Yes	713	661	252	25	44	712	23
60	Bowdoin College	Yes	3356	1019	418	76	100	1490	8
64	Brandeis University	Yes	4186	2743	740	48	77	2819	62

Using the IQR method, more instances in the college dataset are marked as outliers.

#### 12.2.1.2.3 When to use each:

- Use IQR when:
  - Data is skewed
  - You don't want to assume any distribution
  - You want a more sensitive detector
- Use Z-score when:
  - Data is approximately normal
  - You want a more conservative approach
  - You need a method that's easily interpretable

The actual number of outliers marked by each method will depend on your specific dataset's distribution and characteristics.

#### 12.2.2 Common Methods for Handling outliers

Once we identify outlier instances using statistical methods, the next step is to handle them. Below are some commonly used approaches:

- **Removing Outliers:** Discarding extreme values if they are likely due to errors or are irrelevant for the analysis.
- **Winsorizing:** Capping/flooring outliers to a specific percentile to reduce their influence without removing them completely.
- **Replacing with the median:** Replacing outlier values with the median of the remaining data. The median is less affected by extreme values than the mean, making it an ideal choice for imputation when dealing with outliers
- **Transforming Data:** Applying transformations (e.g., log, square root) to reduce the impact of outliers.

```

def method_removal():
    """Method 1: Remove outliers"""
    data_cleaned = college[~college['is_outlier']] ['Expend']
    return data_cleaned

def method_capping():
    """Method 2: Capping (Winsorization)"""
    data_capped = college['Expend'].copy()
    data_capped[data_capped < lower_bound] = lower_bound
    data_capped[data_capped > upper_bound] = upper_bound
    return data_capped

def method_mean_replacement():
    """Method 3: Replace with median"""
    data_median = college['Expend'].copy()
    median_value = college[~college['is_outlier']] ['Expend'].median()
    data_median[college['is_outlier']] = median_value
    return data_median

def method_log_transformation():
    """Method 4: Log transformation"""
    data_log = np.log1p(college['Expend'])
    return data_log

```

**Your Practice:** Try each method and compare their results to see how they differ in handling outliers.

In real-world applications, handling outliers should be approached on a case-by-case basis. However, here are some general recommendations:

#### Recommendations for Different Scenarios:

- Data Removal:
  - Use when: You have a large dataset and can afford to lose data
  - Pros: Simple, removes all outlier influence
  - Cons: Loss of data, potential loss of important information
- Capping (Winsorization):
  - Use when: You want to preserve data count while limiting extreme values
  - Pros: Preserves data size, reduces impact of outliers
  - Cons: Artificial boundary creation, potential loss of genuine extreme events

- Median Replacement:
  - Use when: The data is normally distributed, and outliers are verified as errors or anomalies not representing true values
  - Pros: Maintains data size, simple to implement
  - Cons: Reduces variance, may not represent the true distribution
- Log Transformation:
  - Use when: Data has a right-skewed distribution, and you want to reduce the impact of large outliers.
  - Pros: Reduces skewness, minimizes the effect of extreme values, can make data more normal-like.
  - Cons: Only applicable to positive values, may not be effective for extremely high outliers.

## 12.3 Data binning

Data binning is a powerful technique in data analysis where continuous numerical data is grouped into discrete intervals or “bins.” Imagine it as sorting items into distinct categories or “buckets” based on their values, making it easier to observe trends, patterns, and groupings within large datasets. We encounter binning in everyday life without even realizing it.

For example, consider age groups: instead of listing every individual age, we often refer to broader categories like “under 20,” “20-30,” “30-40,” or “40 and over.” This binning approach helps simplify data interpretation and comparison across different age groups. Another common example is tax brackets. Tax rates are often applied to ranges of income, like “up to \$11,000,” “\$11,001 to \$44,725,” and so forth. Here, income values are grouped into bins to determine applicable tax rates, making complex tax data easier to manage and understand.

```
<IPython.core.display.HTML object>
```

### 12.3.1 Key reasons for binning:

- (i) Better interpretation of data
- (ii) Making better recommendations
- (iii) Smooth data, reduce noise

Examples:

**Binning to better interpret data**

1. The number of flu cases everyday may be binned to seasons such as fall, spring, winter and summer, to understand the effect of season on flu.

#### **Binning to make recommendations:**

2. A doctor may like to group patient age into bins. Grouping patient ages into categories such as Age <=12, 12<Age<=18, 18<Age<=65, Age>65 may help recommend the kind/doses of covid vaccine a patient needs.
3. A credit card company may want to bin customers based on their spend, as “High spenders”, “Medium spenders” and “Low spenders”. Binning will help them design customized marketing campaigns for each bin, thereby increasing customer response (or revenue). On the other hand, they use the same campaign for customers within the same bin, thus minimizing marketing costs.

#### **Binning to smooth data, and reduce noise**

4. A sales company may want to bin their total sales to a weekly / monthly / yearly level to reduce the noise in day-to-day sales.

Using our college dataset, let's apply binning to better interpret the relationship between instructional expenditure per student (`Expend`) and graduation rate (`Grad.Rate`) for U.S. universities, and make relevant recommendations

Here is the previous scatterplot of ‘Grad.Rate’ vs ‘Expend’ with a trendline.

```
# scatter plot of Expend vs Grad.Rate
ax=sns.regplot(data = college, x = "Expend", y = "Grad.Rate",scatter_kws={"color": "orange"})
ax.xaxis.set_major_formatter('${x:.0f}')
ax.set_xlabel('Expenditure per student')
ax.set_ylabel('Graduation rate')
plt.show()
```



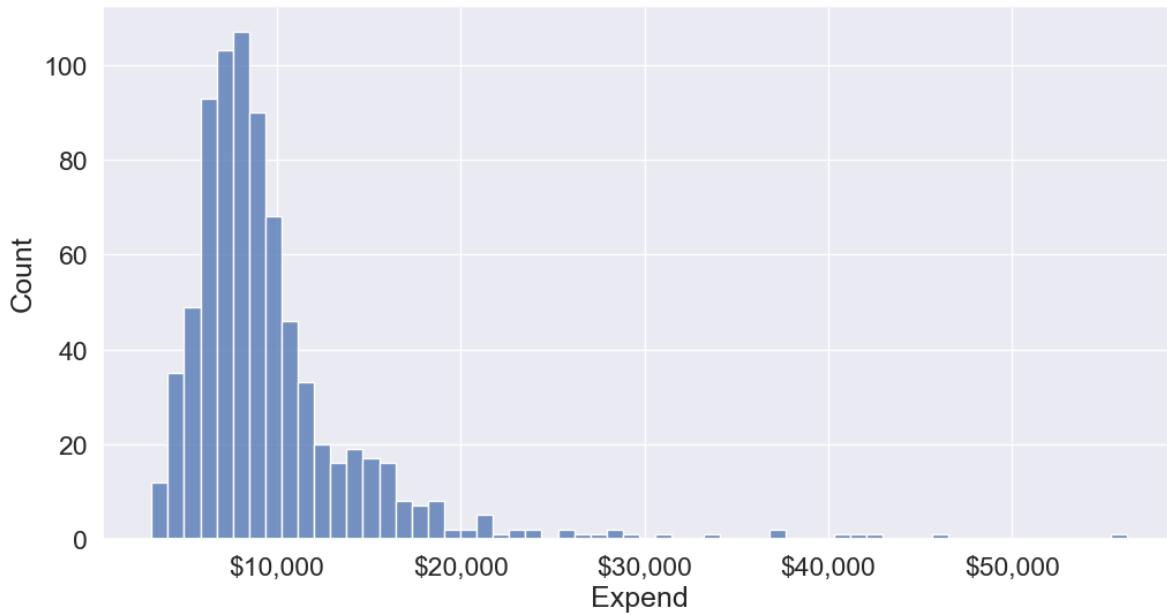
The trendline indicates a positive correlation between `Expend` and `Grad.Rate`. However, there seems to be a lot of noise and presence of outliers in the data, which makes it hard to interpret the overall trend.

### 12.3.2 Common binning methods:

- **Equal-width:** Divides range into equal-sized intervals
- **Equal-size:** Places equal number of observations in each bin
- **Custom:** Based on domain knowledge (like age groups above)

We'll bin `Expend` to see if we can better analyze its association with `Grad.Rate`. However, let us first visualize the distribution of `Expend`.

```
#Visualizing the distribution of expend
ax=sns.histplot(data = college, x= 'Expend')
ax.xaxis.set_major_formatter('${x:.0f}')
```



The distribution of `Expend` is right skewed with potentially some extremely high outlying values.

#### 12.3.2.1 Binning with equal width bins

We'll use the Pandas function `cut()` to bin `Expend`. This function creates bins such that all bins have the same width.

```
#Using the cut() function in Pandas to bin "Expend"
Binned_expend = pd.cut(college['Expend'], 3, retbins = True)
Binned_expend
```

(0	(3132.953, 20868.333]
1	(3132.953, 20868.333]
2	(3132.953, 20868.333]
3	(3132.953, 20868.333]
4	(3132.953, 20868.333]
	...
772	(3132.953, 20868.333]
773	(3132.953, 20868.333]
774	(3132.953, 20868.333]
775	(38550.667, 56233.0]
776	(3132.953, 20868.333]

```
Name: Expend, Length: 777, dtype: category  
Categories (3, interval[float64, right]): [(3132.953, 20868.333] < (20868.333, 38550.667] <  
array([ 3132.953       , 20868.3333333, 38550.6666667, 56233.       ]))
```

The `cut()` function returns a tuple of length 2. The first element of the tuple are the bins, while the second element is an array containing the cut-off values for the bins.

```
type(Binned_expend)
```

```
tuple
```

```
len(Binned_expend)
```

```
2
```

Once the bins are obtained, we'll add a column in the dataset that indicates the bin for `Expend`.

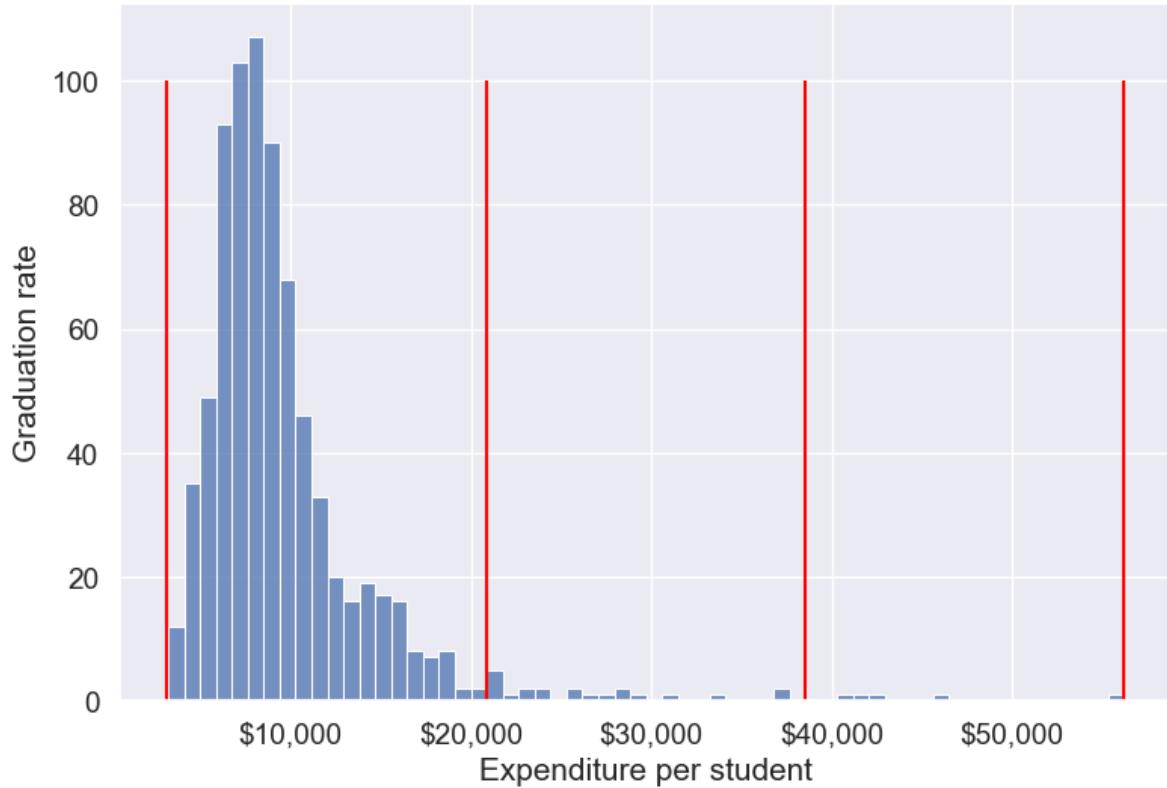
```
#Creating a categorical variable to store the level of expenditure on a student  
college['Expend_bin'] = Binned_expend[0]  
college.head()
```

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad
0	Abilene Christian University	Yes	1660	1232	721	23	52	2885
1	Adelphi University	Yes	2186	1924	512	16	29	2683
2	Adrian College	Yes	1428	1097	336	22	50	1036
3	Agnes Scott College	Yes	417	349	137	60	89	510
4	Alaska Pacific University	Yes	193	146	55	16	44	249

See the variable `Expend_bin` in the above dataset.

Let us visualize the `Expend` bins over the distribution of the `Expend` variable.

```
#Visualizing the bins for instructional expenditure on a student  
sns.set(font_scale=1.25)  
plt.rcParams["figure.figsize"] = (9,6)  
ax=sns.histplot(data = college, x= 'Expend')  
plt.vlines(Binned_expend[1], 0,100,color='red')  
plt.xlabel('Expenditure per student');  
plt.ylabel('Graduation rate');  
ax.xaxis.set_major_formatter('${x:.0f}')
```



By default, the bins created have equal width. They are created by dividing the range between the maximum and minimum value of `Expend` into the desired number of equal-width intervals. We can label the bins as well as follows.

```
college['Expend_bin'] = pd.cut(college['Expend'], 3, labels = ['Low expend', 'Med expend', 'High expend'])
college['Expend_bin']
```

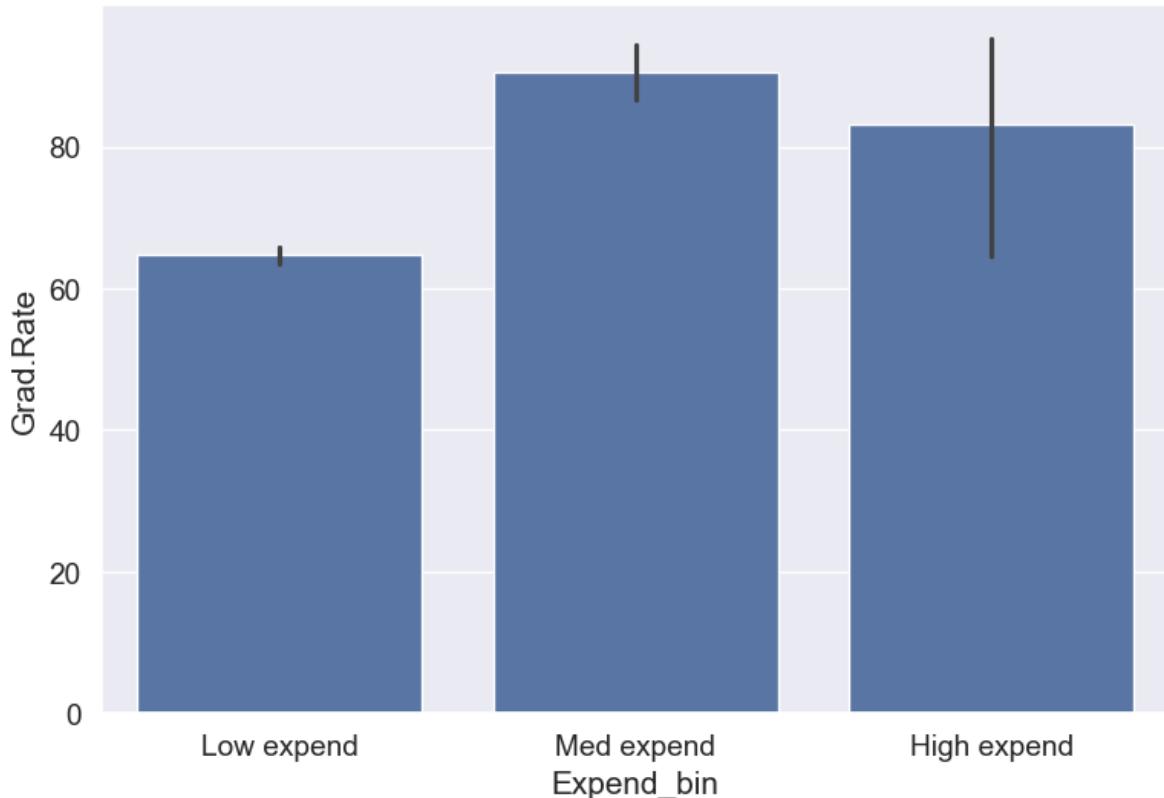
0	Low expend
1	Low expend
2	Low expend
3	Low expend
4	Low expend
	...
772	Low expend
773	Low expend
774	Low expend
775	High expend
776	Low expend

Name: Expend\_bin, Length: 777, dtype: category

```
Categories (3, object): ['Low expend' < 'Med expend' < 'High expend']
```

Now that we have binned the variable `Expend`, let us see if we can better visualize the association of graduation rate with expenditure per student using `Expend_bin`.

```
#Visualizing average graduation rate vs categories of instructional expenditure per student  
sns.barplot(x = 'Expend_bin', y = 'Grad.Rate', data = college)
```



It seems that the graduation rate is the highest for universities with medium level of expenditure per student. This is different from the trend we saw earlier in the scatter plot. Let us investigate.

Let us find the number of universities in each bin.

```
college['Expend_bin'].value_counts()
```

```
Expend_bin  
Low expend    751
```

```
Med expend      21
High expend      5
Name: count, dtype: int64
```

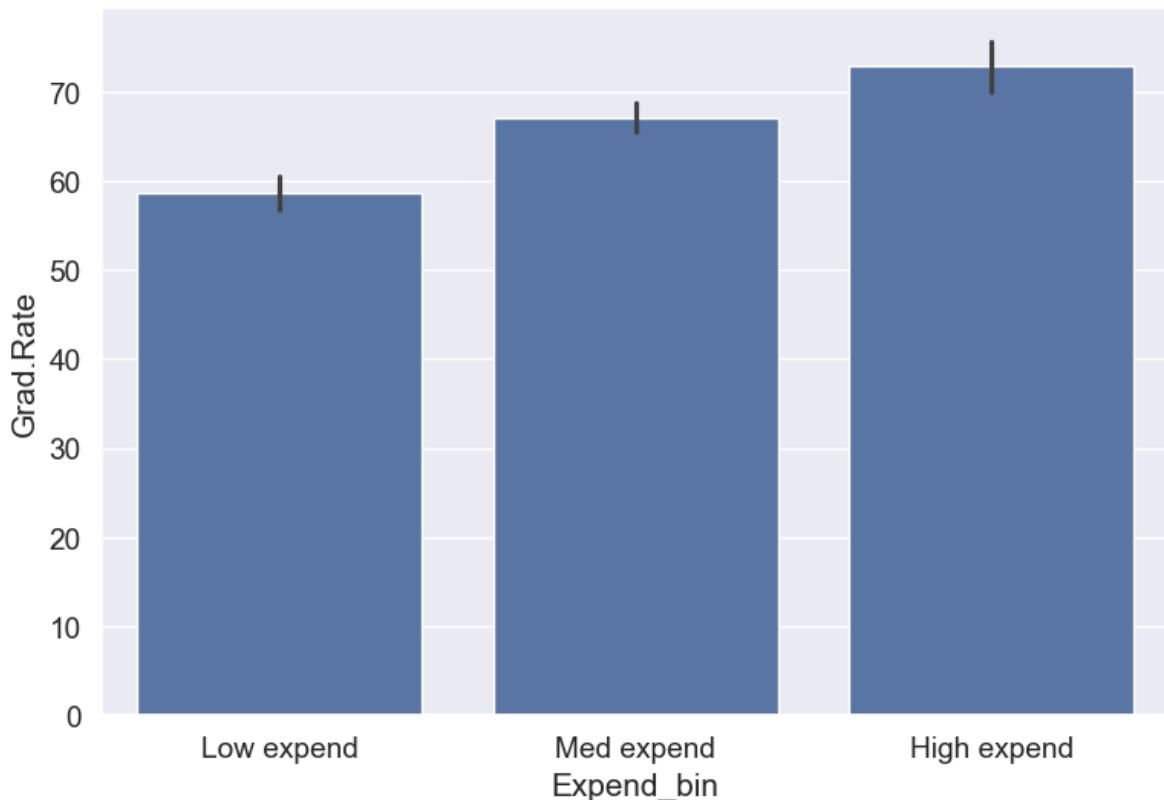
The bin `High expend` consists of only 5 universities, or 0.6% of all the universities in the dataset. These universities may be outliers that are skewing the trend (as also evident in the histogram above).

Let us see if we get the correct trend with the outliers removed from the data.

```
#Data without outliers
college_data_without_outliers = college[((college.Expend>=lower_bound) & (college.Expend<=upper_bound))

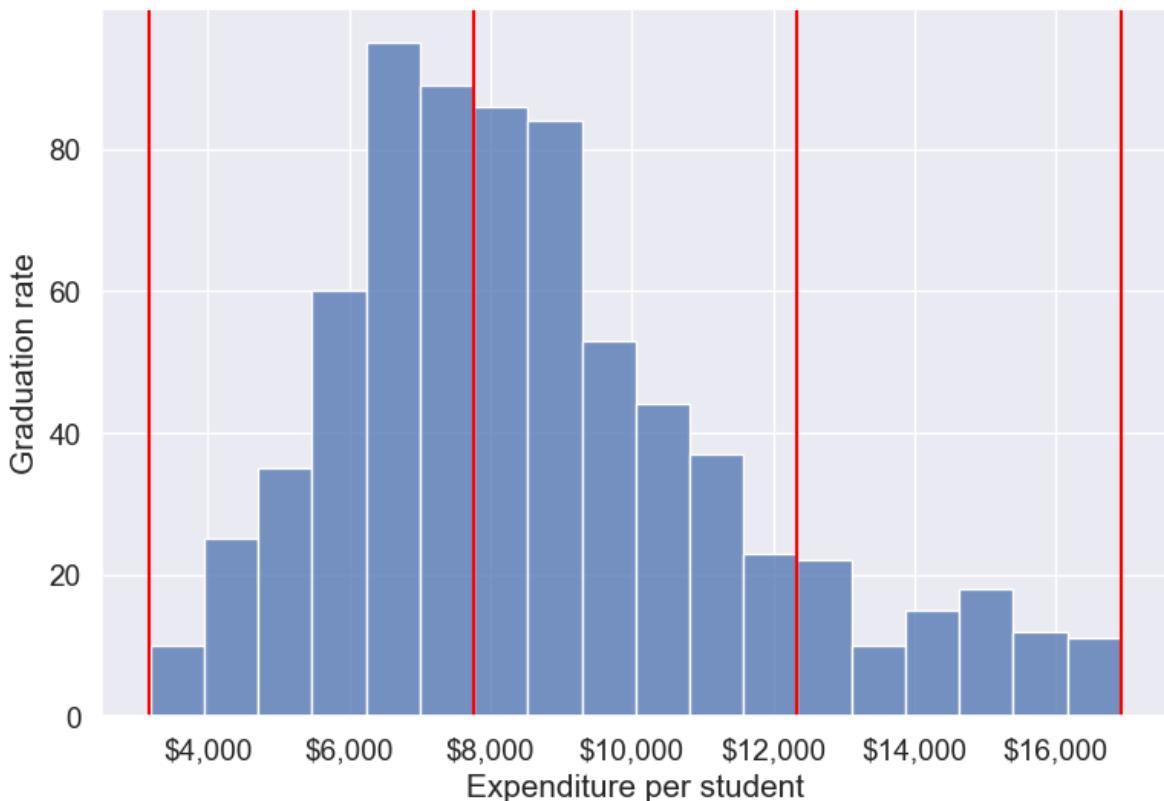
Binned_data = pd.cut(college_data_without_outliers['Expend'],3,labels = ['Low expend','Med expend','High expend'])
college_data_without_outliers.loc[:, 'Expend_bin'] = Binned_data[0]

sns.barplot(x = 'Expend_bin', y = 'Grad.Rate', data = college_data_without_outliers)
```



With the outliers removed, we obtain the correct overall trend, even in the case of equal-width bins. Note that these bins have unequal number of observations as shown below.

```
ax=sns.histplot(data = college_data_without_outliers, x= 'Expend')
for i in range(4):
    plt.axvline(Binned_data[1][i], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:,.0f}')
```



Note that the right tail of the histogram has disappeared since we removed outliers.

```
college_data_without_outliers['Expend_bin'].value_counts()
```

```
Expend_bin
Med expend      327
Low expend      314
High expend     88
Name: count, dtype: int64
```

Instead of removing outliers, we can address the issue by binning observations into equal-sized bins, ensuring each bin has the same number of observations. Let's explore this approach next.

### 12.3.2.2 Binning with equal sized bins

Let us bin the variable `Expend` such that each bin consists of the same number of observations.

We'll use the Pandas function `qcut()` to make equal-sized bins (in contrast to equal-width bins in the previous section).

```
#Using the Pandas function qcut() to create bins with the same number of observations
Binned_expend = pd.qcut(college['Expend'], 3, retbins = True)
college['Expend_bin'] = Binned_expend[0]
```

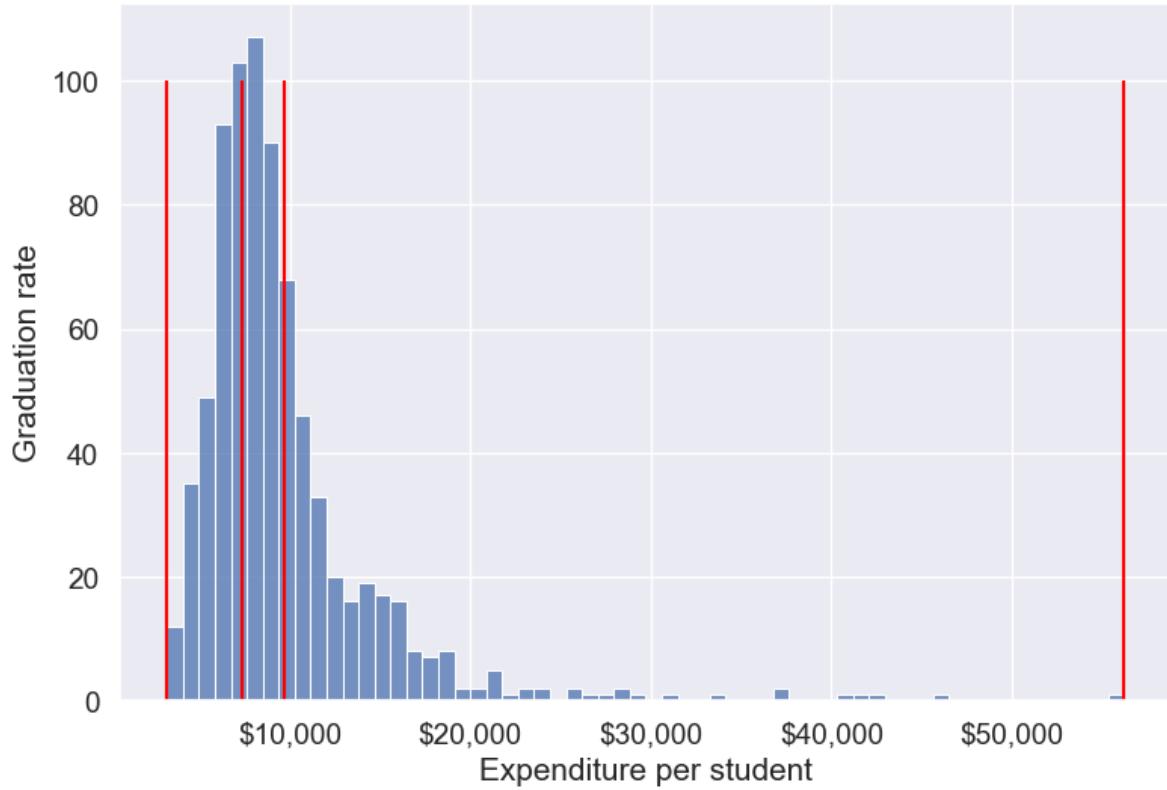
Each bin has the same number of observations with `qcut()`:

```
college['Expend_bin'].value_counts()
```

```
Expend_bin
(3185.999, 7334.333]    259
(7334.333, 9682.333]    259
(9682.333, 56233.0]     259
Name: count, dtype: int64
```

Let us visualize the `Expend` bins over the distribution of the `Expend` variable.

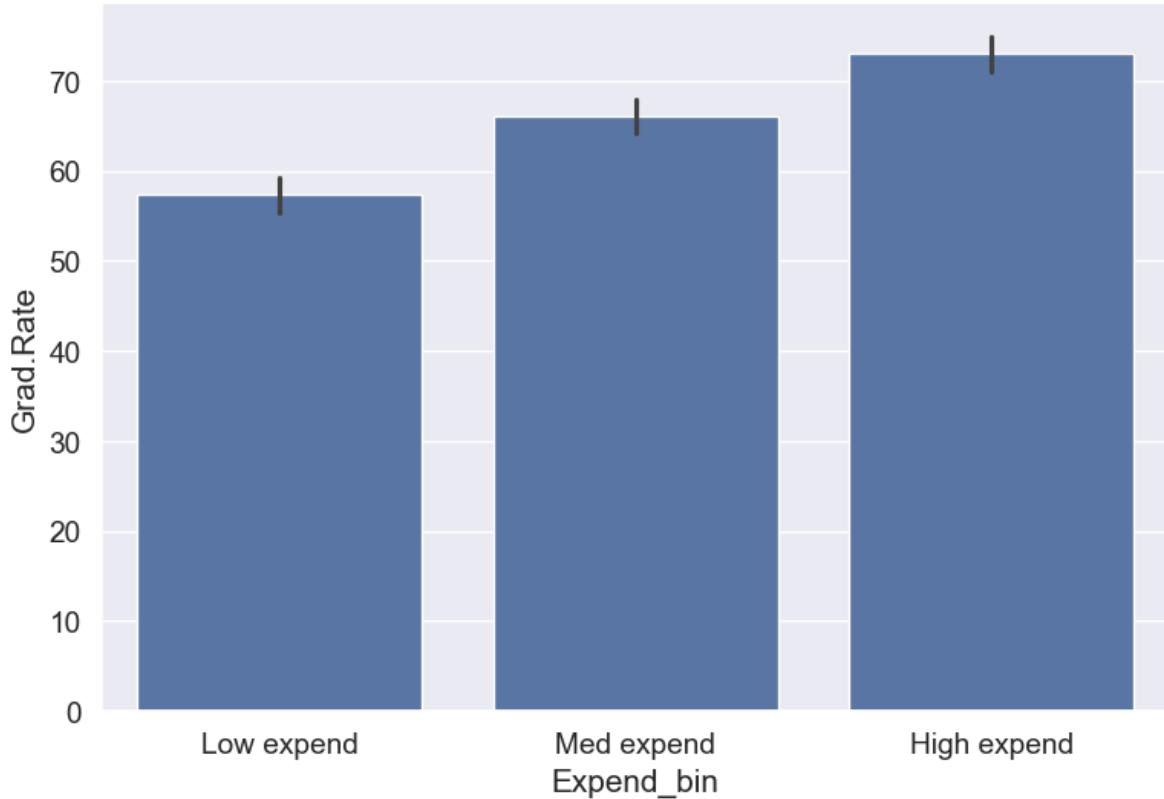
```
#Visualizing the bins for instructional expenditure on a student
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
ax=sns.histplot(data = college, x= 'Expend')
plt.vlines(Binned_expend[1], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:.0f}')
```



Note that the bin-widths have been adjusted to have the same number of observations in each bin. The bins are narrower in domains of high density, and wider in domains of sparse density.

Let us again make the barplot visualizing the average graduate rate with level of instructional expenditure per student.

```
college['Expend_bin'] = pd.qcut(college['Expend'], 3, labels = ['Low expend', 'Med expend', 'High expend'])
a=sns.barplot(x = 'Expend_bin', y = 'Grad.Rate', data = college)
```



Now we see the same trend that we saw in the scatterplot, but without the noise. We have smoothed the data. Note that making equal-sized bins helps reduce the effect of outliers in the overall trend.

Suppose this analysis was done to provide recommendations to universities for increasing their graduation rate. With binning, we can provide one recommendation to '*Low expend*' universities, and another one to '*Med expend*' universities. For example, the recommendations can be:

1. '*Low expend*' universities can expect an increase of 9 percentage points in `Grad.Rate`, if they migrate to the '*Med expend*' category.
2. '*Med expend*' universities can expect an increase of 7 percentage points in `Grad.Rate`, if they migrate to the '*High expend*' category.

The numbers in the above recommendations are based on the table below.

```
college.groupby(college.Expend_bin, observed=False)[['Grad.Rate']].mean()
```

Expend_bin	Grad.Rate
Low expend	57.343629

```

Med expend      66.057915
High expend     72.988417
Name: Grad.Rate, dtype: float64

```

We can also provide recommendations based on the confidence intervals of the mean graduation rate (`Grad.Rate`). The confidence intervals, calculated below, are derived using a method known as bootstrapping. For a detailed explanation of bootstrapping, please refer to [this article on Wikipedia](#).

```

# Bootstrapping to find 95% confidence intervals of Graduation Rate of US universities based
confidence_intervals = {}

# Loop through each expenditure bin
for expend_bin, data_sub in college.groupby('Expend_bin', observed=False):
    # Generate bootstrap samples for the graduation rate
    samples = np.random.choice(data_sub['Grad.Rate'], size=(10000, len(data_sub)), replace=True)

    # Calculate the mean of each sample
    sample_means = samples.mean(axis=1)

    # Calculate the 95% confidence interval
    lower_bound = np.percentile(sample_means, 2.5)
    upper_bound = np.percentile(sample_means, 97.5)

    # Store the result in the dictionary
    confidence_intervals[expend_bin] = (round(lower_bound, 2), round(upper_bound, 2))

# Print the results
for expend_bin, ci in confidence_intervals.items():
    print(f"95% Confidence interval of Grad.Rate for {expend_bin} universities = [{ci[0]}, {ci[1]}]")

```

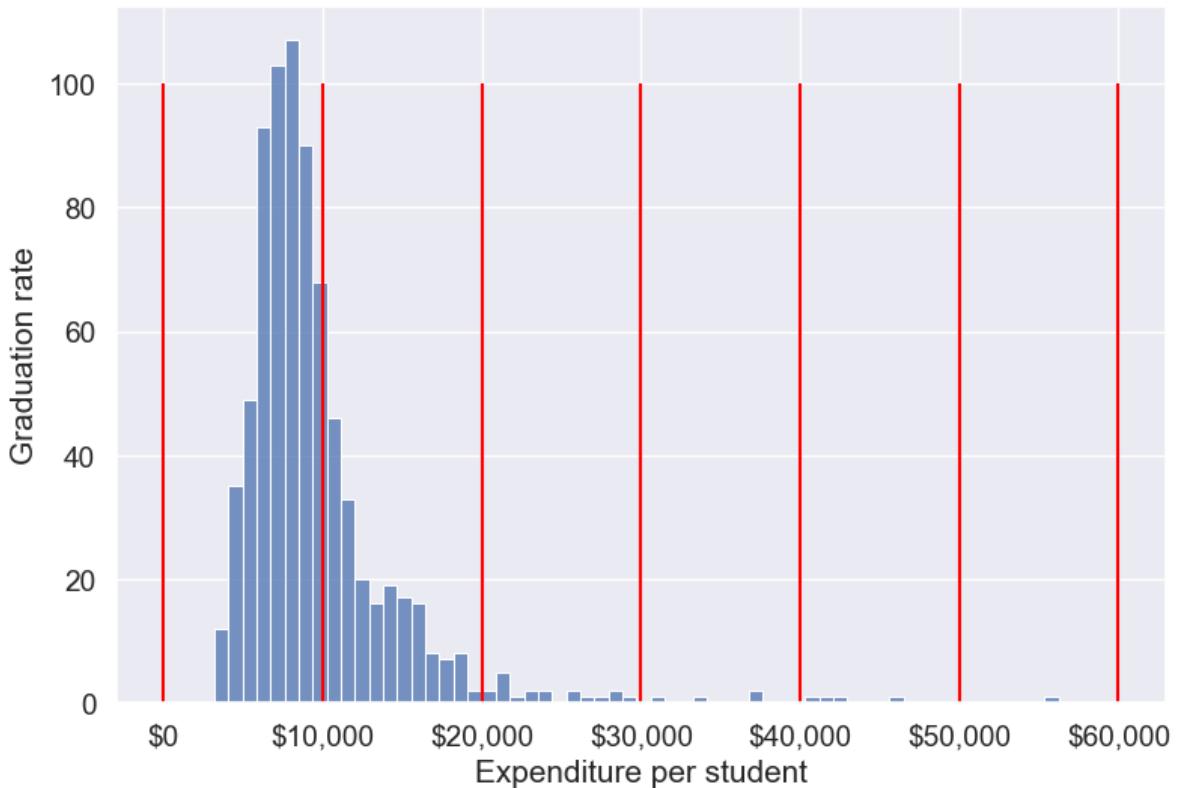
95% Confidence interval of Grad.Rate for Low expend universities = [55.36, 59.36]  
95% Confidence interval of Grad.Rate for Med expend universities = [64.15, 67.95]  
95% Confidence interval of Grad.Rate for High expend universities = [71.03, 74.92]

Apart from equal-width and equal-sized bins, custom bins can be created using the `bins` argument. Suppose, bins are to be created for `Expend` with cutoffs \$10,000, \$20,000, \$30,000...\$60,000. Then, we can use the `bins` argument as in the code below:

### 12.3.2.3 Binning with custom bins

```
Binned_expend = pd.cut(college.Expend,bins = list(range(0,70000,10000)),retbins=True)

#Visualizing the bins for instructional expenditure on a student
ax=sns.histplot(data = college, x= 'Expend')
plt.vlines(Binned_expend[1], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:,0f}')
```



Next, let's create a custom bin with unequal sizes and widths.

```
# Create bins and labels
bins = [college['Expend'].min(), 10000, 40000, college['Expend'].max()]
labels = ['Low Expenditure', 'Medium Expenditure', 'High Expenditure']

# Binning the 'Expend' column
```

```

college['Expend_Binned'] = pd.cut(college['Expend'], bins=bins, labels=labels)

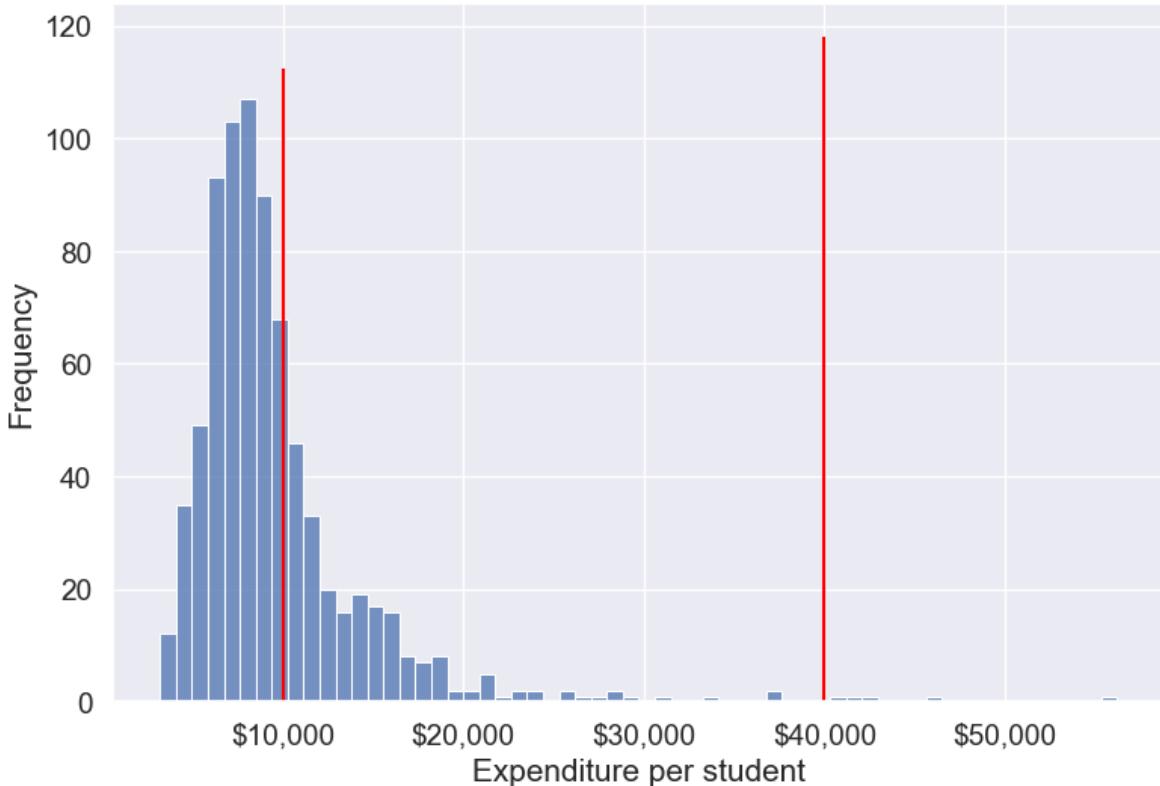
# Visualizing the bins using a histogram
ax = sns.histplot(data=college, x='Expend', kde=False)

# Add vertical lines for the bin edges (the bin edges are the values in 'bins')
for bin_edge in bins[1:-1]: # skip the first and last bins since they are the edges
    plt.vlines(bin_edge, 0, ax.get_ylim()[1], color='red')

# Add labels and formatting
plt.xlabel('Expenditure per student')
plt.ylabel('Frequency')
ax.xaxis.set_major_formatter('${x:,.0f}')

# Display the plot
plt.show()

```



As custom bin-cutoffs can be specified with the `cut()` function, custom bin quantiles can be specified with the `qcut()` function.

## 12.4 Dummy / Indicator variables

Dummy or indicator variables are binary variables created to represent categorical data in numerical form, typically used in regression and machine learning models. Each category is represented by a separate dummy variable with values of 0 or 1.

### 12.4.1 Purpose:

- **Enables categorical data** to be used in quantitative analysis: Models like linear regression and many machine learning algorithms require numerical input, making dummy variables essential for handling categorical data.
- **Preserves information**: Converts categorical information into a format that models can interpret without losing meaning.

### 12.4.2 When to use dummy variables

Dummy variables are especially useful in:

- **Regression models**: For categorical predictors, such as region, gender, or type of product.
- **Machine learning algorithms**: Many models, including tree-based methods, can benefit from dummy variables for categorical features.

### 12.4.3 How to Create Dummy variables

The pandas library in Python has a built-in function, `pd.get_dummies()`, to generate dummy variables. If a column in a DataFrame has  $k$  distinct values, we will get a DataFrame with  $k$  columns containing 0s and 1s

Let us make dummy variables with the equal-sized bins we created for the average instruction expenditure per student.

```
#Using the Pandas function qcut() to create bins with the same number of observations
Binned_expend = pd.qcut(college['Expend'], 3, retbins = True, labels = ['Low_expend', 'Med_expend'])
college['Expend_bin'] = Binned_expend[0]
```

```
#Making dummy variables based on the levels (categories) of the 'Expend_bin' variable
dummy_Expend = pd.get_dummies(college['Expend_bin'], dtype=int)
```

The dummy data `dummy_Expend` has a value of 1 if the observation corresponds to the category referenced by the column name.

```
dummy_Expend.head()
```

	Low_expend	Med_expend	High_expend
0	1	0	0
1	0	0	1
2	0	1	0
3	0	0	1
4	0	0	1

Adding the dummy columns back to the original dataframe

```
#Concatenating the dummy variables to the original data
college = pd.concat([college,dummy_Expend],axis=1)
print(college.shape)
college.head()
```

(777, 26)

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad
0	Abilene Christian University	Yes	1660	1232	721	23	52	2885
1	Adelphi University	Yes	2186	1924	512	16	29	2683
2	Adrian College	Yes	1428	1097	336	22	50	1036
3	Agnes Scott College	Yes	417	349	137	60	89	510
4	Alaska Pacific University	Yes	193	146	55	16	44	249

We can find the correlation between the dummy variables and graduation rate to identify if any of the dummy variables will be useful to estimate graduation rate (**Grad.Rate**).

```
#Finding if dummy variables will be useful to estimate 'Grad.Rate'
dummy_Expend.corrwith(college['Grad.Rate'])
```

```
Low_expend    -0.334456
Med_expend     0.024492
High_expend    0.309964
dtype: float64
```

The dummy variables **Low\_expend** and **High\_expend** may contribute in explaining **Grad.Rate** in a regression model.

#### 12.4.4 Using drop\_first to Avoid Multicollinearity

In cases like linear regression, you may want to drop one dummy variable to avoid multicollinearity (perfect correlation between variables).

```
# create dummy variables, dropping the first to avoid multicollinearity
dummy_Expend = pd.get_dummies(college['Expend_bin'], dtype=int, drop_first=True)

# add the dummy variables to the original dataset
college = pd.concat([college, dummy_Expend], axis=1)

college.head()
```

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad
0	Abilene Christian University	Yes	1660	1232	721	23	52	2885
1	Adelphi University	Yes	2186	1924	512	16	29	2683
2	Adrian College	Yes	1428	1097	336	22	50	1036
3	Agnes Scott College	Yes	417	349	137	60	89	510
4	Alaska Pacific University	Yes	193	146	55	16	44	249

## 12.5 Independent Study

### 12.5.1 Practice exercise 1

Read `survey_data_clean.csv`. Split the columns of the dataset, such that all columns with categorical values transform into dummy variables with each category corresponding to a column of 0s and 1s. Leave the *Timestamp* column.

As all categorical columns are transformed to dummy variables, all columns have numeric values.

What is the total number of columns in the transformed data? What is the total number of columns of the original data?

Find the:

1. Top 5 variables having the highest positive correlation with NU\_GPA.
2. Top 5 variables having the highest negative correlation with NU\_GPA.

### 12.5.2 Practice exercise 2

Consider the dataset *survey\_data\_clean.csv*. Find the number of outliers in each column of the dataset based on both the Z-score and IQR criteria. Do not use a `for` loop.

Which column(s) have the maximum number of outliers using Z-score?

Which column(s) have the maximum number of outliers using IQR?

Do you think the outlying observations identified for those columns(s) should be considered as outliers? If not, then which type of columns should be considered when finding outliers?

# 13 Data Grouping and Aggregation

Grouping and aggregation are essential in data science because they enable meaningful analysis and insights from complex and large-scale datasets. Here are key reasons why they are important:

- **Data Summarization:** Grouping allows you to condense large amounts of data into concise summaries. Aggregation provides statistical summaries (e.g., mean, sum, count), making it easier to understand data trends and characteristics without needing to examine every detail.
- **Insights Across Categories:** Grouping by categories, like regions, demographics, or time periods, lets you analyze patterns within each group. For example, aggregating sales data by region can reveal which areas are performing better, or grouping customer data by age group can highlight consumer trends.
- **Time Series Analysis:** In time-series data, grouping by time intervals (e.g., day, month, quarter) allows you to observe trends over time, which can be crucial for forecasting, seasonal analysis, and trend detection.
- **Reducing Data Complexity:** By summarizing data at a higher level, you reduce complexity, making it easier to visualize and interpret, especially in large datasets where working with raw data could be overwhelming.

In this chapter, we are going to see grouping and aggregating using pandas. Grouping and aggregating will help to achieve data analysis easily using various functions. These methods will help us to the group and summarize our data and make complex analysis comparatively easy.

Throughout this chapter, we will use *gdp\_lifeExpectancy.csv*, let's read the csv file to pandas dataframe first

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

```
# Load the data
gdp_lifeExp_data = pd.read_csv('./Datasets/gdp_lifeExpectancy.csv')
gdp_lifeExp_data.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

## 13.1 Grouping by a single column

`groupby()` allows you to split a DataFrame based on the values in one or more columns. First, we'll explore grouping by a single column.

### 13.1.1 Syntax of `groupby()`

- `DataFrame.groupby(by="column_name")` – Groups data by the specified column.

Consider the life expectancy dataset. suppose we want to group by the observations by `continent` by passing it as an argument to the `groupby()` method.

```
#Creating a GroupBy object
grouped = gdp_lifeExp_data.groupby('continent')
#This will split the data into groups that correspond to values of the column 'continent'
```

The `groupby()` method returns a `GroupBy` object.

```
#A 'GroupBy' objects is created with the `groupby()` function
type(grouped)
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

The `GroupBy` object `grouped` contains the information of the groups in which the data is distributed. Each observation has been assigned to a specific group of the column(s) used to group the data. However, note that the dataset is not physically split into different DataFrames. For example, in the above case, each observation is assigned to a particular group depending on the value of the `continent` for that observation. However, all the observations are still in the same DataFrame `data`.

## 13.1.2 Attributes and methods of the *GroupBy* object

### 13.1.2.1 keys

The object(s) grouping the data are called *key(s)*. Here `continent` is the group key. The keys of the *GroupBy* object can be seen using Its `keys` attribute.

```
#Key(s) of the GroupBy object  
grouped.keys
```

```
'continent'
```

### 13.1.2.2 ngroups

The number of groups in which the data is distributed based on the keys can be seen with the `ngroups` attribute.

```
#The number of groups based on the key(s)  
grouped.ngroups
```

```
5
```

The group names are the *keys* of the dictionary, while the row labels are the corresponding *values*

```
#Group names  
grouped.groups.keys()
```

```
dict_keys(['Africa', 'Americas', 'Asia', 'Europe', 'Oceania'])
```

### 13.1.2.3 groups

The `groups` attribute of the *GroupBy* object contains the group labels (or names) and the row labels of the observations in each group, as a dictionary.

```
#The groups (in the dictionary format)  
grouped.groups
```

```
{'Africa': [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
```

#### 13.1.2.4 groups.values

The `groups.values` attribute of the `GroupBy` object contains the row labels of the observations in each group, as a dictionary.

```
#Group values are the row labels corresponding to a particular group
grouped.groups.values()
```

```
dict_values([Index([ 24,    25,    26,    27,    28,    29,    30,    31,    32,    33,
       ...,
       1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1703],
      dtype='int64', length=624), Index([ 48,    49,    50,    51,    52,    53,    54,    55,    56,
       ...,
       1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643],
      dtype='int64', length=300), Index([  0,    1,    2,    3,    4,    5,    6,    7,    8,
       ...,
       1670, 1671, 1672, 1673, 1674, 1675, 1676, 1677, 1678, 1679],
      dtype='int64', length=396), Index([ 12,    13,    14,    15,    16,    17,    18,    19,    20,
       ...,
       1598, 1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607],
      dtype='int64', length=360), Index([ 60,    61,    62,    63,    64,    65,    66,    67,    68,
       ...,
       1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101, 1102, 1103],
      dtype='int64')])
```

#### 13.1.2.5 size()

The `size()` method of the `GroupBy` object returns the number of observations in each group.

```
#Number of observations in each group
grouped.size()
```

```
continent
Africa      624
Americas    300
Asia        396
Europe      360
Oceania     24
dtype: int64
```

### 13.1.2.6 `first()`

The first non missing element of each group is returned with the `first()` method of the `GroupBy` object.

```
#The first element of the group can be printed using the first() method  
grouped.first()
```

continent	country	year	lifeExp	pop	gdpPercap
Africa	Algeria	1952	43.077	9279525	2449.008185
Americas	Argentina	1952	62.485	17876956	5911.315053
Asia	Afghanistan	1952	28.801	8425333	779.445314
Europe	Albania	1952	55.230	1282697	1601.056136
Oceania	Australia	1952	69.120	8691212	10039.595640

### 13.1.2.7 `get_group()`

This method returns the observations for a particular group of the `GroupBy` object.

```
#Observations for individual groups can be obtained using the get_group() function  
grouped.get_group('Asia')
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1675	Yemen, Rep.	Asia	1987	52.922	11219340	1971.741538
1676	Yemen, Rep.	Asia	1992	55.599	13367997	1879.496673
1677	Yemen, Rep.	Asia	1997	58.020	15826497	2117.484526
1678	Yemen, Rep.	Asia	2002	60.308	18701257	2234.820827
1679	Yemen, Rep.	Asia	2007	62.698	22211743	2280.769906

## 13.2 Data aggregation within groups

### 13.2.1 Common Aggregation Functions

Aggregation functions are essential for summarizing and analyzing data in pandas. These functions allow you to compute summary statistics for your data, making it easier to identify trends and patterns.

Below are some of the most commonly used aggregation functions when working with grouped data in pandas:

- `mean()` – Calculates the **average** value of the group.
- `sum()` – Computes the **total** value by summing all elements in the group.
- `min()` – Finds the **minimum** value in the group.
- `max()` – Finds the **maximum** value in the group.
- `count()` – Returns the **number of occurrences** or entries in the group.
- `median()` – Finds the **middle value** in the sorted group.
- `std()` – Calculates the **standard deviation**, measuring the spread or variation in the values of the group.

Each of these functions can help summarize and provide insights into different aspects of the grouped data.

Consider the life expectancy dataset, let's find the mean life expectancy, population and GDP per capita for each country during the period of 1952 -2007.

Next, we'll find the mean statistics for each group with the `mean()` method. The method will be applied on all columns of the DataFrame and all groups.

```
#Grouping the observations by 'country'  
grouped_country = gdp_lifeExp_data.drop(['continent', 'year'], axis = 1).groupby('country')  
  
#Finding the mean stastistic of all columns of the DataFrame and all groups  
grouped_country.mean()
```

country	lifeExp	pop	gdpPercap
Afghanistan	37.478833	1.582372e+07	802.674598
Albania	68.432917	2.580249e+06	3255.366633
Algeria	59.030167	1.987541e+07	4426.025973
Angola	37.883500	7.309390e+06	3607.100529
Argentina	69.060417	2.860224e+07	8955.553783
...	...	...	...

country	lifeExp	pop	gdpPercap
Vietnam	57.479500	5.456857e+07	1017.712615
West Bank and Gaza	60.328667	1.848606e+06	3759.996781
Yemen, Rep.	46.780417	1.084319e+07	1569.274672
Zambia	45.996333	6.353805e+06	1358.199409
Zimbabwe	52.663167	7.641966e+06	635.858042

Next, we'll find the standard deviation statistics for each group with the `std()` method.

```
grouped_country.std()
```

country	lifeExp	pop	gdpPercap
Afghanistan	5.098646	7.114583e+06	108.202929
Albania	6.322911	8.285855e+05	1192.351513
Algeria	10.340069	8.613355e+06	1310.337656
Angola	4.005276	2.672281e+06	1165.900251
Argentina	4.186470	7.546609e+06	1862.583151
...	...	...	...
Vietnam	12.172331	2.052585e+07	567.482251
West Bank and Gaza	11.000069	1.023057e+06	1716.840614
Yemen, Rep.	11.019302	5.590408e+06	609.939160
Zambia	4.453246	3.096949e+06	247.494984
Zimbabwe	7.071816	3.376895e+06	133.689213

Alternatively, you can compute other statistical metrics.

### 13.2.2 Multiple aggregations and Custom aggregation using `agg()`

#### 13.2.2.1 Multiple aggregations

Directly applying the aggregate methods of the `GroupBy` object such as `mean`, `count`, etc., lets us apply only one function at a time. Also, we may wish to apply an aggregate function of our own, which is not there in the set of methods of the `GroupBy` object, such as the range of values of a column.

The `agg()` function of a `GroupBy` object lets us aggregate data using:

1. Multiple aggregation functions
2. Custom aggregate functions (in addition to in-built functions like mean, std, count etc.)

Consider the life expectancy dataset, Let us use the agg() method of the GroupBy object to simultaneously find the mean and standard deviation of the *gdpPercap* for each country.

For aggregating by multiple functions, we pass a list of strings to agg(), where the strings are the function names.

```
grouped_country['gdpPercap'].agg(['mean','std']).sort_values(by = 'mean',ascending = False).
```

	mean	std
country		
Kuwait	65332.910472	33882.139536
Switzerland	27074.334405	6886.463308
Norway	26747.306554	13421.947245
United States	26261.151347	9695.058103
Canada	22410.746340	8210.112789

### 13.2.2.2 Custom aggregation

In addition to the mean and standard deviation of the gdpPercap of each country, let us also include the range of gdpPercap in the table above using lambda function

```
grouped_country['gdpPercap'].agg(lambda x: x.max() - x.min())
```

country	
Afghanistan	342.670088
Albania	4335.973390
Algeria	3774.359280
Angola	3245.635491
Argentina	6868.064587
	...
Vietnam	1836.509912
West Bank and Gaza	5595.075290
Yemen, Rep.	1499.052330
Zambia	705.723500
Zimbabwe	392.478061
Name: gdpPercap, Length: 142, dtype: float64	

```
# define a function that calculates the range
def range_func(x):
    return x.max() - x.min()
```

```
# apply the range function to the 'gdpPercap' column besides the mean and standard deviation
grouped_country['gdpPercap'].agg(['mean', 'std', range_func]).sort_values(by = 'range_func',
```

country	mean	std	range_func
Kuwait	65332.910472	33882.139536	85404.702920
Singapore	17425.382267	14926.147774	44828.041413
Norway	26747.306554	13421.947245	39261.768450
Hong Kong, China	16228.700865	12207.329731	36670.557461
Ireland	15758.606238	11573.311022	35465.716022
...	...	...	...
Rwanda	675.669043	142.229906	388.246772
Senegal	1533.121694	105.399353	344.572767
Afghanistan	802.674598	108.202929	342.670088
Ethiopia	509.115155	96.427627	328.659296
Burundi	471.662990	99.329720	292.403419

For aggregating by multiple functions & changing the column names resulting from those functions, we pass a list of tuples to `agg()`, where each tuple is of length two, and contains the new column name & the function to be applied.

```
#Simultaneous renaming of columns while grouping
grouped_country['gdpPercap'].agg([('Average','mean'),('Standard Deviation','std'),('90th Per
```

country	Average	Standard Deviation	90th Percentile
Kuwait	65332.910472	33882.139536	109251.315590
Norway	26747.306554	13421.947245	44343.894158
United States	26261.151347	9695.058103	38764.132898
Singapore	17425.382267	14926.147774	35772.742520
Switzerland	27074.334405	6886.463308	34246.394240
...	...	...	...
Liberia	604.814141	98.988329	706.275527
Malawi	575.447212	122.999953	689.590541

country	Average	Standard Deviation	90th Percentile
Burundi	471.662990	99.329720	615.597260
Myanmar	439.333333	175.401531	592.300000
Ethiopia	509.115155	96.427627	577.448804

### 13.2.3 Multiple aggregate functions on multiple columns

Let us find the mean and standard deviation of lifeExp and pop for each country

```
# find the meand and standard deviation of the 'lifeExp' and 'pop' column for each country
grouped_country[['lifeExp', 'pop']].agg(['mean', 'std']).sort_values(by = ('lifeExp', 'mean'))
```

country	lifeExp		pop	
	mean	std	mean	std
Iceland	76.511417	3.026593	2.269781e+05	4.854168e+04
Sweden	76.177000	3.003990	8.220029e+06	6.365660e+05
Norway	75.843000	2.423994	4.031441e+06	4.107955e+05
Netherlands	75.648500	2.486363	1.378680e+07	2.005631e+06
Switzerland	75.565083	4.011572	6.384293e+06	8.582009e+05
...	...	...	...	...
Mozambique	40.379500	4.599184	1.204670e+07	4.457509e+06
Guinea-Bissau	39.210250	4.937369	8.820084e+05	3.132917e+05
Angola	37.883500	4.005276	7.309390e+06	2.672281e+06
Afghanistan	37.478833	5.098646	1.582372e+07	7.114583e+06
Sierra Leone	36.769167	3.937828	3.605425e+06	1.270945e+06

### 13.2.4 Distinct aggregate functions on multiple columns

For aggregating by multiple functions, we pass a list of strings to `agg()`, where the strings are the function names.

For aggregating by multiple functions & changing the column names resulting from those functions, we pass a list of tuples to `agg()`, where each tuple is of length two, and contains the new column name as the first object and the function to be applied as the second object of the tuple.

For aggregating by multiple functions such that a distinct set of functions is applied to each column, we pass a dictionary to `agg()`, where the keys are the column names on which the

function is to be applied, and the values are the list of strings that are the function names, or a list of tuples if we also wish to name the aggregated columns.

```
# We can use a list to apply multiple aggregation functions to a single column, and a dictionary to name the resulting columns
# Use string names for the aggregation functions
grouped_country.agg({"gdpPercap": ["mean", "std"], "lifeExp": ["median", "std"], "pop": ["max", "min"]})
```

country	gdpPercap		lifeExp		pop	
	mean	std	median	std	max	min
Afghanistan	802.674598	108.202929	39.1460	5.098646	31889923	8425333
Albania	3255.366633	1192.351513	69.6750	6.322911	3600523	1282697
Algeria	4426.025973	1310.337656	59.6910	10.340069	33333216	9279525
Angola	3607.100529	1165.900251	39.6945	4.005276	12420476	4232095
Argentina	8955.553783	1862.583151	69.2115	4.186470	40301927	17876956
...	...	...	...	...	...	...
Vietnam	1017.712615	567.482251	57.2900	12.172331	85262356	26246839
West Bank and Gaza	3759.996781	1716.840614	62.5855	11.000069	4018332	1030585
Yemen, Rep.	1569.274672	609.939160	46.6440	11.019302	22211743	4963829
Zambia	1358.199409	247.494984	46.0615	4.453246	11746035	2672000
Zimbabwe	635.858042	133.689213	53.1765	7.071816	12311143	3080907

Next, for each country, find the mean and standard deviation of the `lifeExp`, and the minimum and maximum values of `gdpPercap`.

```
#Specifying arguments to the function as a dictionary if distinct functions are to be applied to each column
grouped_country.agg({'lifeExp':[('Average','mean'),('Standard deviation','std')], 'gdpPercap':[('min','min'), ('max','max')])
```

country	lifeExp		gdpPercap	
	Average	Standard deviation	min	max
Afghanistan	37.478833	5.098646	635.341351	978.011439
Albania	68.432917	6.322911	1601.056136	5937.029526
Algeria	59.030167	10.340069	2449.008185	6223.367465
Angola	37.883500	4.005276	2277.140884	5522.776375
Argentina	69.060417	4.186470	5911.315053	12779.379640
...	...	...	...	...
Vietnam	57.479500	12.172331	605.066492	2441.576404
West Bank and Gaza	60.328667	11.000069	1515.592329	7110.667619

country	lifeExp			gdpPercap	
	Average	Standard deviation	min	max	
Yemen, Rep.	46.780417	11.019302	781.717576	2280.769906	
Zambia	45.996333	4.453246	1071.353818	1777.077318	
Zimbabwe	52.663167	7.071816	406.884115	799.362176	

## 13.3 Grouping by Multiple Columns

Above, we demonstrated grouping by a single column, which is useful for summarizing data based on one categorical variable. However, in many cases, we need to group by multiple columns. Grouping by multiple columns allows us to create more detailed summaries by accounting for multiple categorical variables. This approach enables us to analyze data at a finer granularity, revealing insights that might be missed with single-column grouping alone.

### 13.3.1 Basic Syntax for Grouping by Multiple Columns

Use `groupby()` with a list of column names to group data by multiple columns.

- `DataFrame.groupby(by=["col1", "col2"])`

Consider the life expectancy dataset, we can group by both country and continent to analyze `gdpPercap`, `lifeExp`, and `pop` trends for each country within each continent, providing a more comprehensive view of the data.

```
#Grouping by multiple columns
grouped_continent_ctry = gdp_lifeExp_data.groupby(['continent', 'country'])["lifeExp"].agg
```

```
grouped_continent_ctry
```

continent	country
	Iceland
	Sweden
Europe	Norway
	Netherlands
	Switzerland
...	...

---

continent	country
Africa	Mozambique
Asia	Guinea-Bissau
Africa	Angola
	Afghanistan
	Sierra Leone

### 13.3.2 Understanding Hierarchical (Multi-Level) Indexing

- Grouping by multiple columns creates a hierarchical index (also called a multi-level index).
- This index allows each level (e.g., continent, country) to act as an independent category that can be accessed individually.

In the above output, `continent` and `country` form a two-level hierarchical index, allowing us to drill down from continent-level to country-level summaries.

```
grouped_continent_country.index.levels
```

2

```
# get the first level of the index
grouped_continent_country.index.levels[0]
```

```
Index(['Africa', 'Americas', 'Asia', 'Europe', 'Oceania'], dtype='object', name='continent')
```

```
# get the second level of the index
grouped_continent_country.index.levels[1]
```

```
Index(['Afghanistan', 'Albania', 'Algeria', 'Angola', 'Argentina', 'Australia',
       'Austria', 'Bahrain', 'Bangladesh', 'Belgium',
       ...
       'Uganda', 'United Kingdom', 'United States', 'Uruguay', 'Venezuela',
       'Vietnam', 'West Bank and Gaza', 'Yemen, Rep.', 'Zambia', 'Zimbabwe'],
       dtype='object', name='country', length=142)
```

### 13.3.3 Subsetting Data in a Hierarchical Index

`grouped_continent_country` is still a DataFrame with hierarchical indexing. You can use `.loc[]` for subsetting, just as you would with a single-level index.

```
# get the observations for the 'Americas' continent
grouped_continent_country.loc['Americas'].head()
```

country	mean	std	max	min
Canada	74.902750	3.952871	80.653	68.750
United States	73.478500	3.343781	78.242	68.440
Puerto Rico	72.739333	3.984267	78.746	64.280
Cuba	71.045083	6.022798	78.273	59.421
Uruguay	70.781583	3.342937	76.384	66.071

```
# get the mean life expectancy for the 'Americas' continent
grouped_continent_country.loc['Americas']['mean'].head()
```

```
country
Canada      74.902750
United States 73.478500
Puerto Rico   72.739333
Cuba         71.045083
Uruguay      70.781583
Name: mean, dtype: float64
```

```
# another way to get the mean life expectancy for the 'Americas' continent
grouped_continent_country.loc['Americas', 'mean'].head()
```

```
country
Canada      74.902750
United States 73.478500
Puerto Rico   72.739333
Cuba         71.045083
Uruguay      70.781583
Name: mean, dtype: float64
```

You can use a tuple to access data for specific levels in a multi-level index.

```
# get the observations for the 'United States' country  
grouped_continent_ctry.loc[('Americas', 'United States')]
```

```
mean    73.478500  
std     3.343781  
max    78.242000  
min    68.440000  
Name: (Americas, United States), dtype: float64
```

```
grouped_continent_ctry.loc[('Americas', 'United States'), ['mean', 'std']]
```

```
mean    73.478500  
std     3.343781  
Name: (Americas, United States), dtype: float64
```

```
gdp_lifeExp_data.columns
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpPercap'], dtype='object')
```

Finally, you can use `reset_index()` to convert the hierarchical index into a regular index, allowing you to apply the standard subsetting and filtering methods covered in previous chapters

```
grouped_continent_ctry.reset_index().head()
```

	continent	country	mean	std	max	min
0	Europe	Iceland	76.511417	3.026593	81.757	72.49
1	Europe	Sweden	76.177000	3.003990	80.884	71.86
2	Europe	Norway	75.843000	2.423994	80.196	72.67
3	Europe	Netherlands	75.648500	2.486363	79.762	72.13
4	Europe	Switzerland	75.565083	4.011572	81.701	69.62

### 13.3.4 Grouping by multiple columns and aggregating multiple variables

```
#Grouping by multiple columns  
grouped_continent_ctry_multi = gdp_lifeExp_data.groupby(['continent', 'country', 'year'])[[  
grouped_continent_ctry_multi
```

---

continent	country
Africa	Algeria
...	...
Oceania	New Zealand
...	...

---

## Breaking Down Grouping and Aggregation

- **Grouping by Multiple Columns:**

In this example, we are grouping the data by three columns: `continent`, `country`, and `year`. This creates groups based on unique combinations of these columns.

- **Aggregating Multiple Variables:**

We apply multiple aggregation functions (`mean`, `std`, `max`, and `min`) to multiple variables (`lifeExp`, `pop`, and `gdpPercap`).

This type of operation is commonly referred to as “**multi-column grouping with multiple aggregations**” in pandas. It’s a powerful approach because it allows us to obtain a detailed statistical summary for each combination of grouping columns across several variables.

```
# its columns are also two levels deep
grouped_continent_country_multi.columns.nlevels
```

2

```
# pass a tuple to the loc() method to access the values of the multi-level columns with a multi-level index
grouped_continent_country_multi.loc[('Americas','United States'), ('lifeExp', 'mean')]
```

year	
1952	68.440
1957	69.490
1962	70.210

```
1967    70.760
1972    71.340
1977    73.380
1982    74.650
1987    75.020
1992    76.090
1997    76.810
2002    77.310
2007    78.242
Name: (lifeExp, mean), dtype: float64
```

## 13.4 Advanced Operations within groups: apply(), transform(), and filter()

### 13.4.1 Using apply() on groups

The `apply()` function applies a custom function to each group, allowing for flexible operations. The function can return either a scalar, Series, or DataFrame.

**Example:** Consider the life expectancy dataset, find the top 3 life expectancy values for each continent

We'll first define a function that sorts a dataset by decreasing life expectancy and returns the top 3 rows. Then, we'll apply this function on each group using the `apply()` method of the `GroupBy` object.

```
# Define a function to get the top 3 rows based on life expectancy for each group
def top_3_life_expectancy(group):
    return group.nlargest(3, 'lifeExp')

#Defining the groups in the data
grouped_gdpcapital_data = gdp_lifeExp_data.groupby('continent')
```

Now we'll use the `apply()` method to apply the `top_3_life_expectancy()` function on each group of the object `grouped_gdpcapital_data`.

```
# Apply the function to each continent group
top_life_expectancy = gdp_lifeExp_data.groupby('continent')[['continent', 'country', 'year']]

# Display the result
top_life_expectancy.head()
```

	continent	country	year	lifeExp	gdpPercap
0	Africa	Reunion	2007	76.442	7670.122558
1	Africa	Reunion	2002	75.744	6316.165200
2	Africa	Reunion	1997	74.772	6071.941411
3	Americas	Canada	2007	80.653	36319.235010
4	Americas	Canada	2002	79.770	33328.965070

The `top_3_life_expectancy()` function is applied to each group, and the results are concatenated internally with the `concat()` function. The output therefore has a hierarchical index whose outer level indices are the group keys.

We can also use a lambda function instead of separately defining the function `top_3_life_expectancy()`:

```
# Use a lambda function to get the top 3 life expectancy values for each continent
top_life_expectancy = (
    gdp_lifeExp_data
    .groupby('continent')[['continent', 'country', 'year', 'lifeExp', 'gdpPercap']] # Avoid
    .apply(lambda x: x.nlargest(3, 'lifeExp'))
    .reset_index(drop=True)
)

# Display the result
top_life_expectancy.head()
```

	continent	country	year	lifeExp	gdpPercap
0	Africa	Reunion	2007	76.442	7670.122558
1	Africa	Reunion	2002	75.744	6316.165200
2	Africa	Reunion	1997	74.772	6071.941411
3	Americas	Canada	2007	80.653	36319.235010
4	Americas	Canada	2002	79.770	33328.965070

### 13.4.2 Using `transform()` on Groups

The `transform()` function applies a function to each group and returns a Series aligned with the original DataFrame's index. This makes it suitable for adding or modifying columns based on group-level calculations.

Recall that in the data cleaning and preparation chapter, we imputed missing values based on correlated variables in the dataset.

In the example provided, some countries had missing values for GDP per capita. To handle this, we imputed the missing GDP per capita for each country using the average GDP per capita of its corresponding continent.

Now, we'll explore an alternative approach using `groupby()` and `transform()` to perform this imputation.

Let us read the datasets and the function that makes a visualization to compare the imputed values with the actual values.

```
#Importing data with missing values
gdp_missing_data = pd.read_csv('./Datasets/GDP_missing_data.csv')

#Importing data with all values
gdp_complete_data = pd.read_csv('./Datasets/GDP_complete_data.csv')

#index of rows with missing values for GDP per capita
null_ind_gdpPC = gdp_missing_data.index[gdp_missing_data.gdpPerCapita.isnull()]

#Defining a function to plot the imputed values vs actual values
def plot_actual_vs_predicted():
    fig, ax = plt.subplots(figsize=(8, 6))
    plt.rc('xtick', labelsize=15)
    plt.rc('ytick', labelsize=15)
    x = gdp_complete_data.loc[null_ind_gdpPC, 'gdpPerCapita']
    y = gdp_imputed_data.loc[null_ind_gdpPC, 'gdpPerCapita']
    plt.scatter(x,y)
    z=np.polyfit(x,y,1)
    p=np.poly1d(z)
    plt.plot(x,x,color='orange')
    plt.xlabel('Actual GDP per capita', fontsize=20)
    plt.ylabel('Imputed GDP per capita', fontsize=20)
    ax.xaxis.set_major_formatter('${:.0f}')
    ax.yaxis.set_major_formatter('${:.0f}')
    rmse = np.sqrt(((x-y).pow(2)).mean())
    print("RMSE=",rmse)
```

**Approach 1:** Using the approach we used in the previous chapter

```
#Finding the mean GDP per capita of the continent
avg_gdpPerCapita = gdp_missing_data['gdpPerCapita'].groupby(gdp_missing_data['continent']).mean()

#Creating a copy of missing data to impute missing values
```

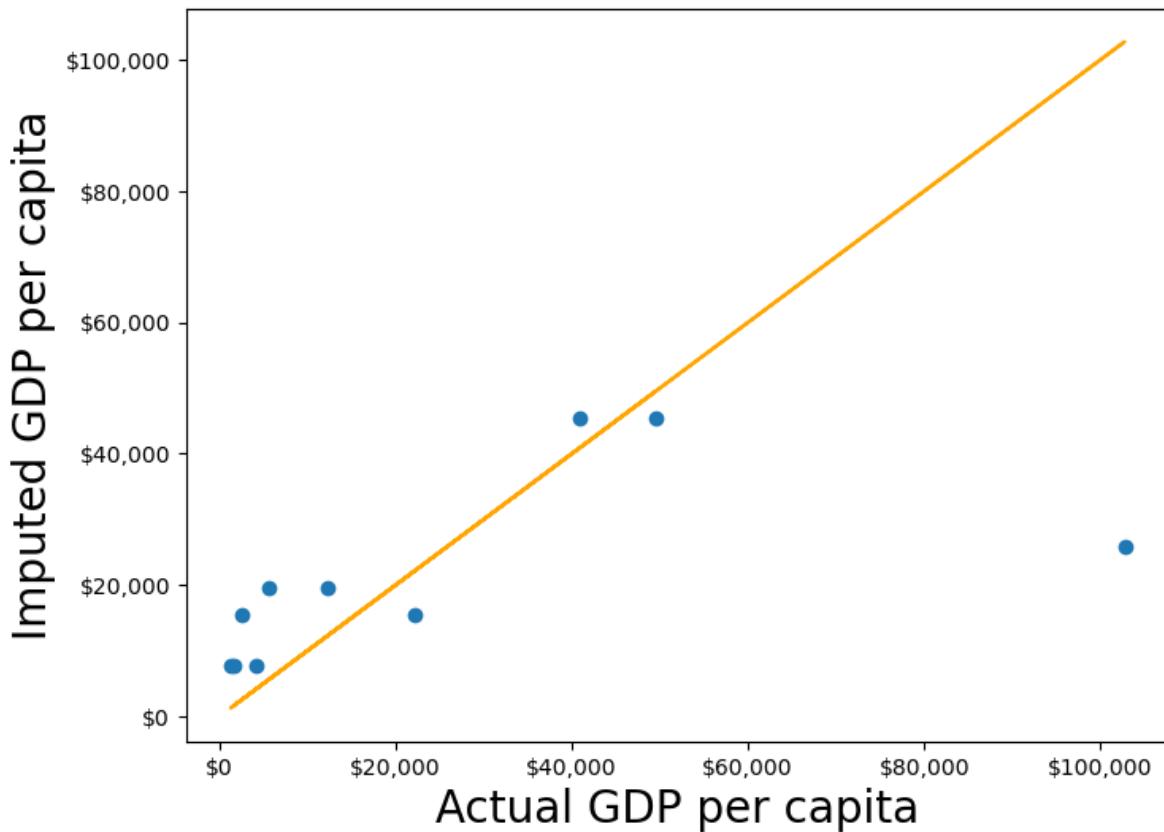
```

gdp_imputed_data = gdp_missing_data.copy()

#Replacing missing GDP per capita with the mean GDP per capita for the corresponding continent
for cont in avg_gdpPerCapita.index:
    gdp_imputed_data.loc[(gdp_imputed_data.continent==cont) & (gdp_imputed_data.gdpPerCapita.isnull()),'gdpPerCapita']=avg_gdpPerCapita[cont]
plot_actual_vs_predicted()

```

RMSE= 25473.20645170116



**Approach 2:** Using the `groupby()` and `transform()` methods.

The `transform()` function is a powerful tool for filling missing values in grouped data. It allows us to apply a function across each group and align the result back to the original DataFrame, making it perfect for filling missing values based on group statistics.

In this example, we use `transform()` to impute missing values in the `gdpPerCapita` column by filling them with the mean `gdpPerCapita` of each continent:

```

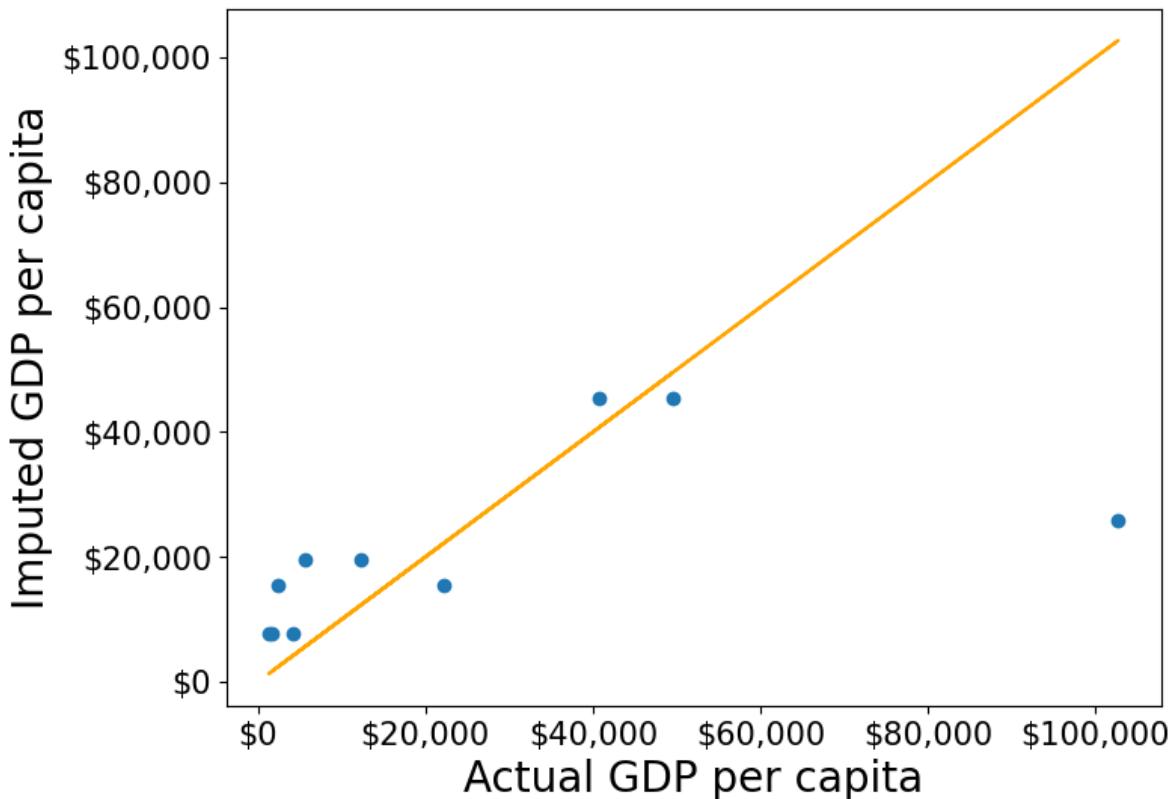
#Creating a copy of missing data to impute missing values
gdp_imputed_data = gdp_missing_data.copy()

#Grouping data by continent
grouped = gdp_missing_data.groupby('continent')

#Imputing missing values with the mean GDP per capita of the continent
gdp_imputed_data['gdpPerCapita'] = grouped['gdpPerCapita'].transform(lambda x: x.fillna(x.mean()))
plot_actual_vs_predicted()

```

RMSE= 25473.20645170116



Using the `transform()` function, missing values in `gdpPerCapita` for each group are filled with the group's mean `gdpPerCapita`. This approach is not only more convenient to write but also faster compared to using for loops. While a for loop imputes missing values one group at a time, `transform()` performs built-in operations (like mean, sum, etc.) in a way that is optimized internally, making it more efficient.

Let's use `apply()` instead of `transform()` with `groupby()`

Please copy the code below and run it in your notebook:

```
#Creating a copy of missing data to impute missing values
gdp_imputed_data = gdp_missing_data.copy()

#Grouping data by continent
grouped = gdp_missing_data.groupby('continent')

#Applying the lambda function on the 'gdpPerCapita' column of the groups
gdp_imputed_data['gdpPerCapita'] = grouped['gdpPerCapita'].apply(lambda x: x.fillna(x.mean()))

plot_actual_vs_predicted()
```

Why we ran into this error? and `apply()` doesn't work?

Here's a deeper look at why `apply()` doesn't work as expected here:

#### 13.4.2.1 Behavior of `groupby().apply()` vs. `groupby().transform()`

- **`groupby().apply()`:** This method applies a function to each group and returns the result with a hierarchical (multi-level) index by default. This hierarchical index can make it difficult to align the result back to a single column in the original DataFrame.
- **`groupby().transform()`:** In contrast, `transform()` is specifically designed to apply a function to each group and return a Series that is aligned with the original DataFrame's index. This alignment makes it directly compatible for assignment to a new or existing column in the original DataFrame.

#### 13.4.2.2 Why `transform()` Works for Imputation

When using `transform()` to fill missing values, it applies the function (e.g., `fillna(x.mean())`) based on each group's statistics, such as the mean, while keeping the result aligned with the original DataFrame's index. This allows for smooth assignment to a column in the DataFrame without any index mismatch issues.

Additionally, `transform()` applies the function to each element in a group independently and returns a result that has the same shape as the original data, making it ideal for adding or modifying columns.

### 13.4.3 Using `filter()` on Groups

The `filter()` function filters entire groups based on a condition. It evaluates each group and keeps only those that meet the specified criteria.

Example: Keep only the countries where the mean life expectancy is greater than 70

```
# keep only the continent where the mean life expectancy is greater than 74
gdp_lifeExp_data.groupby('continent').filter(lambda x: x['lifeExp'].mean() > 74)['continent']

array(['Oceania'], dtype=object)

# keep only the country where the mean life expectancy is greater than 74
gdp_lifeExp_data.groupby('country').filter(lambda x: x['lifeExp'].mean() > 74)['country'].un

array(['Australia', 'Canada', 'Denmark', 'France', 'Iceland', 'Italy',
       'Japan', 'Netherlands', 'Norway', 'Spain', 'Sweden', 'Switzerland'],
      dtype=object)
```

Using `.nunique()` get the number of countries that satisfy this condition

```
gdp_lifeExp_data.groupby('country').filter(lambda x: x['lifeExp'].mean() > 74)['country'].nu
```

12

## 13.5 Sampling data by group

If a dataset contains a large number of observations, operating on it can be computationally expensive. Instead, working on a sample of entire observations is a more efficient alternative. The `groupby()` method combined with `apply()` can be used for stratified random sampling from a large dataset.

Before taking the random sample, let us find the number of countries in each continent.

```
gdp_lifeExp_data.continent.value_counts()
```

```
continent
Africa      624
Asia        396
Europe      360
Americas    300
Oceania     24
Name: count, dtype: int64
```

Let us take a random sample of 650 observations from the entire dataset.

```
sample_lifeExp_data = gdp_lifeExp_data.sample(650)
```

Now, let us see the number of countries of each continent in our sample.

```
sample_lifeExp_data.continent.value_counts()
```

```
continent
Africa      241
Asia        149
Europe      142
Americas    109
Oceania     9
Name: count, dtype: int64
```

Some of the continent have a very low representation in the data. To rectify this, we can take a random sample of 130 observations from each of the 5 continents. In other words, we can take a random sample from each of the continent-based groups.

```
evenly_sampled_lifeExp_data = gdp_lifeExp_data.groupby('continent').apply(lambda x:x.sample(130))

group_sizes = evenly_sampled_lifeExp_data.groupby(level=0).size()
print(group_sizes)
```

```
continent
Africa      130
Americas    130
Asia        130
Europe      130
Oceania     130
dtype: int64
```

The above stratified random sample equally represents all the continent.

## 13.6 `corr()`: Correlation by group

The `corr()` method of the `GroupBy` object returns the correlation between all pairs of columns within each group.

**Example:** Find the correlation between `lifeExp` and `gdpPercap` for each continent-country level combination.

```
gdp_lifeExp_data.groupby(['continent','country']).apply(lambda x:x['lifeExp'].corr(x['gdpPer
```

```
continent    country
Africa        Algeria      0.904471
              Angola       -0.301079
              Benin        0.843949
              Botswana     0.005597
              Burkina Faso  0.881677
              ...
Europe        Switzerland  0.980715
              Turkey       0.954455
              United Kingdom 0.989893
Oceania       Australia    0.986446
              New Zealand   0.974493
Length: 142, dtype: float64
```

Life expectancy is closely associated with GDP per capita across most continent-country combinations.

## 13.7 `pivot_table()`

The `pivot_table()` function in pandas is a powerful tool for performing groupwise aggregation in a structured format, similar to Excel's pivot tables. It allows you to create a summary of data by grouping and aggregating it based on specified columns. Here's an overview of how `pivot_table()` works for groupwise aggregation:

Note that `pivot_table()` is the same as `pivot()` except that `pivot_table()` aggregates the data as well in addition to re-arranging it.

**Example:** Consider the life expectancy dataset, calculate the average life expectancy for each country and year combination

```
pd.pivot_table(data = gdp_lifeExp_data, values = 'lifeExp', index = 'country', columns = 'year')
```

year country	1952	1957	1962	1967	1972	1977	1982	1987
Afghanistan	28.80100	30.332000	31.997000	34.02000	36.088000	38.438000	39.854000	40.8
Albania	55.23000	59.280000	64.820000	66.22000	67.690000	68.930000	70.420000	72.0
Algeria	43.07700	45.685000	48.303000	51.40700	54.518000	58.014000	61.368000	65.7
Angola	30.01500	31.999000	34.000000	35.98500	37.928000	39.483000	39.942000	39.9
Argentina	62.48500	64.399000	65.142000	65.63400	67.065000	68.481000	69.942000	70.7
...	...	...	...	...	...	...	...	...
West Bank and Gaza	43.16000	45.671000	48.127000	51.63100	56.532000	60.765000	64.406000	67.0
Yemen, Rep.	32.54800	33.970000	35.180000	36.98400	39.848000	44.175000	49.113000	52.9
Zambia	42.03800	44.077000	46.023000	47.76800	50.107000	51.386000	51.821000	50.8
Zimbabwe	48.45100	50.469000	52.358000	53.99500	55.635000	57.674000	60.363000	62.8
All	49.05762	51.507401	53.609249	55.67829	57.647386	59.570157	61.533197	63.5

## Explanation

- **values:** Specifies the column to aggregate (e.g., `lifeExp` in our example).
- **index:** Groups by the rows based on the `country` column.
- **columns:** Groups by the columns based on the `year` column.
- **aggfunc:** Uses `mean` to calculate the average life expectancy.

## Common Aggregation Functions in `pivot_table()`

You can use various aggregation functions within `pivot_table()` to summarize data:

- **mean** – Calculates the average of values within each group.
- **sum** – Computes the total of values within each group.
- **count** – Counts the number of non-null entries within each group.
- **min** and **max** – Finds the minimum and maximum values within each group.

We can also use custom *GroupBy* aggregate functions with `pivot_table()`.

**Example:** Find the 90<sup>th</sup> percentile of life expectancy for each country and year combination

```
pd.pivot_table(data = gdp_lifeExp_data, values = 'lifeExp', index = 'country', columns = 'year')
```

year country	1952	1957	1962	1967	1972	1977	1982	1987	1992	1997
Afghanistan	28.801	30.332	31.997	34.020	36.088	38.438	39.854	40.822	41.674	41.763
Albania	55.230	59.280	64.820	66.220	67.690	68.930	70.420	72.000	71.581	72.950
Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744	69.152
Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647	40.963
Argentina	62.485	64.399	65.142	65.634	67.065	68.481	69.942	70.774	71.868	73.275
...	...	...	...	...	...	...	...	...	...	...
Vietnam	40.412	42.887	45.363	47.838	50.254	55.764	58.816	62.820	67.662	70.672
West Bank and Gaza	43.160	45.671	48.127	51.631	56.532	60.765	64.406	67.046	69.718	71.096
Yemen, Rep.	32.548	33.970	35.180	36.984	39.848	44.175	49.113	52.922	55.599	58.020
Zambia	42.038	44.077	46.023	47.768	50.107	51.386	51.821	50.821	46.100	40.238
Zimbabwe	48.451	50.469	52.358	53.995	55.635	57.674	60.363	62.351	60.377	46.809

## 13.8 crosstab()

### 13.8.1 Basic Usage of crosstab()

The `crosstab()` method is a special case of a pivot table for computing group frequencies (or size of each group).

```
# create a basic crosstab to see counts of countries in each continent
pd.crosstab(gdp_lifeExp_data['continent'], columns='count')
```

continent	col_0	count
Africa	624	
Americas	300	
Asia	396	
Europe	360	
Oceania	24	

Use the `margins=True` argument to add totals (row and column margins) to the table. The `All` row and column provide totals for each age group and gender category.

```
# create a crosstab to see counts of countries in each continent and year
pd.crosstab(gdp_lifeExp_data['continent'], gdp_lifeExp_data['year'], margins=True)
```

year continent	1952	1957	1962	1967	1972	1977	1982	1987	1992	1997	2002	2007	All
Africa	52	52	52	52	52	52	52	52	52	52	52	52	624
Americas	25	25	25	25	25	25	25	25	25	25	25	25	300
Asia	33	33	33	33	33	33	33	33	33	33	33	33	396
Europe	30	30	30	30	30	30	30	30	30	30	30	30	360
Oceania	2	2	2	2	2	2	2	2	2	2	2	2	24
All	142	142	142	142	142	142	142	142	142	142	142	142	1704

This table shows the count of each year in each continent group, helping us understand the year distribution across different continent groups. We may often use it to check if the data is representative of all groups that are of interest to us.

### 13.8.2 Using crosstab() with Aggregation Functions

You can specify a `values` column and an aggregation function (`aggfunc`) to summarize numerical data for each combination of categorical variables.

**Example:** find the mean life expectancy for each continent and year

```
# find the mean life expectancy for each country in each continent and year
pd.crosstab(gdp_lifeExp_data['continent'], gdp_lifeExp_data['year'], values=gdp_lifeExp_data['life_expectancy'])
```

year continent	1952	1957	1962	1967	1972	1977	1982	1987	1992	1997	2002	2007	2012
Africa	39.135500	41.266346	43.319442	45.334538	47.450942	49.580423	51.592865	53.344788	55.210500	57.177200	59.143900	61.110600	63.077300
Americas	53.279840	55.960280	58.398760	60.410920	62.394920	64.391560	66.228840	68.090720	70.058400	71.026100	72.993800	74.961500	76.929200
Asia	46.314394	49.318544	51.563223	54.663640	57.319269	59.610556	62.617939	64.851182	67.105500	69.353200	71.600900	73.848600	76.096300
Europe	64.408500	66.703067	68.539233	69.737600	70.775033	71.937767	72.806400	73.642167	75.494800	77.342500	79.290200	81.237900	83.185600
Oceania	69.255000	70.295000	71.085000	71.310000	71.910000	72.855000	74.290000	75.320000	76.767500	78.205200	80.642900	83.080600	85.518300

## 13.9 Independent Study

### 13.9.1 Practice exercise 1

Read the table consisting of GDP per capita of countries from the webpage: [https://en.wikipedia.org/wiki/List\\_of\\_countries\\_by\\_GDP\\_\(PPP\)\\_per\\_capita](https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(PPP)_per_capita).

To only read the relevant table, read the tables that contain the word '*Country*'.

Estimate the GDP per capita of each country as the average of the estimates of the three agencies - IMF, United Nations and World Bank.

We need to do a bit of data cleaning before we could directly use the `groupby()` function. Follow the steps below:

1. Drop the “Year” column
2. Drop the level 1 column name (innermost level column)
3. Apply the following function on all the columns of the “Estimate” to convert them to numeric: `f = lambda x:pd.to_numeric(x,errors = 'coerce')`
4. Set the Country/Territory column as the index
5. Convert all other columns into numeric and coerce errors using ‘errors=’coerce’‘
6. Drop rows with NaN values
7. Find the average GDP per capital across the three agencies

```
dfs = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_capita')
gdp_data = dfs[0]
gdp_data.head()
```

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
0	Monaco	—	—	240862	2022	240535	2022
1	Liechtenstein	—	—	187267	2022	197268	2022
2	Luxembourg	135321	2024	128259	2023	125897	2022
3	Bermuda	—	—	123091	2022	117568	2022
4	Switzerland	106098	2024	99995	2023	93636	2022

```
# Flatten the MultiIndex to access column names
gdp_data.columns = [' '.join(col).strip() if isinstance(col, tuple) else col for col in gdp_data.columns]

# Drop all columns containing "Year"
gdp_data = gdp_data.loc[:, ~gdp_data.columns.str.contains('Year', case=False)]

gdp_data = gdp_data.rename(columns={"Country/Territory Country/Territory": "Country/Territory"})

gdp_data.head()
```

	Country/Territory	IMF	World Bank	United Nations
0	Monaco	—	240862	240535
1	Liechtenstein	—	187267	197268

	Country/Territory	IMF	World Bank	United Nations
2	Luxembourg	135321	128259	125897
3	Bermuda	—	123091	117568
4	Switzerland	106098	99995	93636

```
import re
column_name_cleaner = lambda x:re.split(r'\[', x)[0]

gdp_data.columns = gdp_data.columns.map(column_name_cleaner)
gdp_data.head()
```

	Country/Territory	IMF	World Bank	United Nations
0	Monaco	—	240862	240535
1	Liechtenstein	—	187267	197268
2	Luxembourg	135321	128259	125897
3	Bermuda	—	123091	117568
4	Switzerland	106098	99995	93636

```
# set the country column as the index
gdp_data.set_index('Country/Territory', inplace=True)

# convert all other columns into numeric and coerce errors
gdp_data = gdp_data.apply(pd.to_numeric, errors='coerce')

# drop rows with NaN values
gdp_data.dropna(inplace=True)

#find the average GDP per capita
gdp_data['Average GDP per capita'] = gdp_data.mean(axis=1)
gdp_data.head()
```

Country/Territory	IMF	World Bank	United Nations	Average GDP per capita
Luxembourg	135321.0	128259.0	125897.0	129825.666667
Switzerland	106098.0	99995.0	93636.0	99909.666667
Ireland	103500.0	103685.0	105993.0	104392.666667
Norway	90434.0	87962.0	106623.0	95006.333333
Singapore	89370.0	84734.0	78115.0	84073.000000

### **13.9.2 Practice exercise 2**

Read the spotify dataset from `spotify_data.csv` that contains information about tracks and artists

#### **13.9.2.1**

Find the mean and standard deviation of the track popularity for each genre.

#### **13.9.2.2**

Create a new categorical column, `energy_lvl`, with two levels – ‘Low energy’ and ‘High energy’ – using equal-sized bins based on the track’s energy level. Then, calculate the mean, standard deviation, and 90th percentile of track popularity for each genre and energy level combination

#### **13.9.2.3**

Find the mean and standard deviation of track popularity and danceability for each genre and energy level. What insights you can gain from the generated table

#### **13.9.2.4**

For each genre and energy level, find the mean and standard deviation of the track popularity, and the minimum and maximum values of loudness.

#### **13.9.2.5**

Find the most popular artist from each genre.

#### **13.9.2.6**

Filter the first 4 columns of the spotify dataset. Drop duplicate observations in the resulting dataset using the Pandas DataFrame method `drop_duplicates()`. Find the top 3 most popular artists for each genre.

### **13.9.2.7**

The spotify dataset has more than 200k observations. It may be expensive to operate with so many observations. Take a random sample of 650 observations to analyze spotify data, such that all genres are equally represented.

### **13.9.2.8**

Find the correlation between `danceability` and `track popularity` for each genre-energy level combination.

### **13.9.2.9**

Find the mean of track popularity for each genre-energy lvl combination such that each row corresponds to a genre, and the energy levels correspond to columns.

Hints: using `pivot_table()`

### **13.9.2.10**

Find the 90<sup>th</sup> percentile of track popularity for each genre-energy lvl combination such that each row corresponds to a genre, and the energy levels correspond to columns.

Hints: using `pivot_table()`

### **13.9.2.11**

Find the number of observations in each group, where each groups corresponds to a distinct genre-energy lvl combination

### **13.9.2.12**

Find the percentage of observations in each group of the above table.

### **13.9.2.13**

What percentage of unique tracks are contributed by the top 5 artists of each genre?

**Hint:** Find the top 5 artists based on `artist_popularity` for each genre. Count the total number of unique tracks (`track_name`) contributed by these artists. Divide this number by the total number of unique tracks in the data. The `nunique()` function will be useful.

# 14 Data Reshaping and Enrichment

In this chapter, we will explore how to reshape and enrich data using pandas. Data reshaping and enriching are essential steps in data preprocessing because they help transform raw data into a more structured, useful, and meaningful format that can be effectively used for analysis or modeling.

```
<IPython.core.display.HTML object>
```

## 14.1 Data Reshaping

Data reshaping involves altering the structure of the data to suit the requirements of the analysis or modeling process. This is often needed when the data is in a format that is difficult to analyze or when it needs to be organized in a specific way.

### 14.1.1 Why Data Reshaping is Important:

- **Improves Data Organization:** Raw data may be in a long format, where each observation is recorded on a separate row, or it may be in a wide format, where the same variable is spread across multiple columns. Reshaping allows you to reorganize the data into a format that is more suitable for analysis, such as turning multiple columns into a single column (melting) or aggregating rows into summary statistics (pivoting).
- **Facilitates Analysis:** Some analysis techniques require specific data formats. For example, many machine learning algorithms require data in a matrix form, where each row represents a sample and each column represents a feature. Reshaping helps to convert data into this format.
- **Simplifies Visualization:** Visualizing data is easier when it is in the right shape. For instance, plotting categorical data across time is easier when the data is in a tidy format, rather than having multiple variables spread across columns.
- **Efficient Grouping and Aggregation:** In some cases, you may want to group data by certain categories or time periods, and reshaping can help aggregate data effectively for summarization.

### 14.1.2 Wide vs. Long Data Format in Data Reshaping

- **Wide Format:** Each variable in its own column; each row represents a unit, and data for multiple variables is spread across columns.
- **Long Format:** Variables are stacked into one column per variable, with additional columns used to indicate time, categories, or other distinctions, leading to a greater number of rows.

To reshape between these two formats, you can use pandas functions such as `pivot_table()`, `melt()`, `stack()`, and `unstack()`. Let's next take a look at each of these methods one by one.

Throughout this section, we will continue using the `gdp_lifeExpectancy.csv` dataset. Let's start by reading the CSV file into a pandas DataFrame.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline

# Load the data
gdp_lifeExp_data = pd.read_csv('./Datasets/gdp_lifeExpectancy.csv')
gdp_lifeExp_data.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

### 14.1.3 Pivoting “long” to “wide” using `pivot_table()`

In the last chapter, we learned that `pivot_table()` is a powerful tool for performing groupwise aggregation in a structured format. The `pivot_table()` function is used to reshape data by specifying columns for the index, columns, and values.

#### 14.1.3.1 Pivoting a single column

Consider the life expectancy dataset. Let's calculate the average life expectancy for each combination of country and year.

```
gdp_lifeExp_data_pivot = gdp_lifeExp_data.pivot_table(index=['continent','country'], columns=
```

	year
continent	country
	Algeria
Africa	Angola
	Benin
	Botswana
	Burkina Faso

#### Explanation

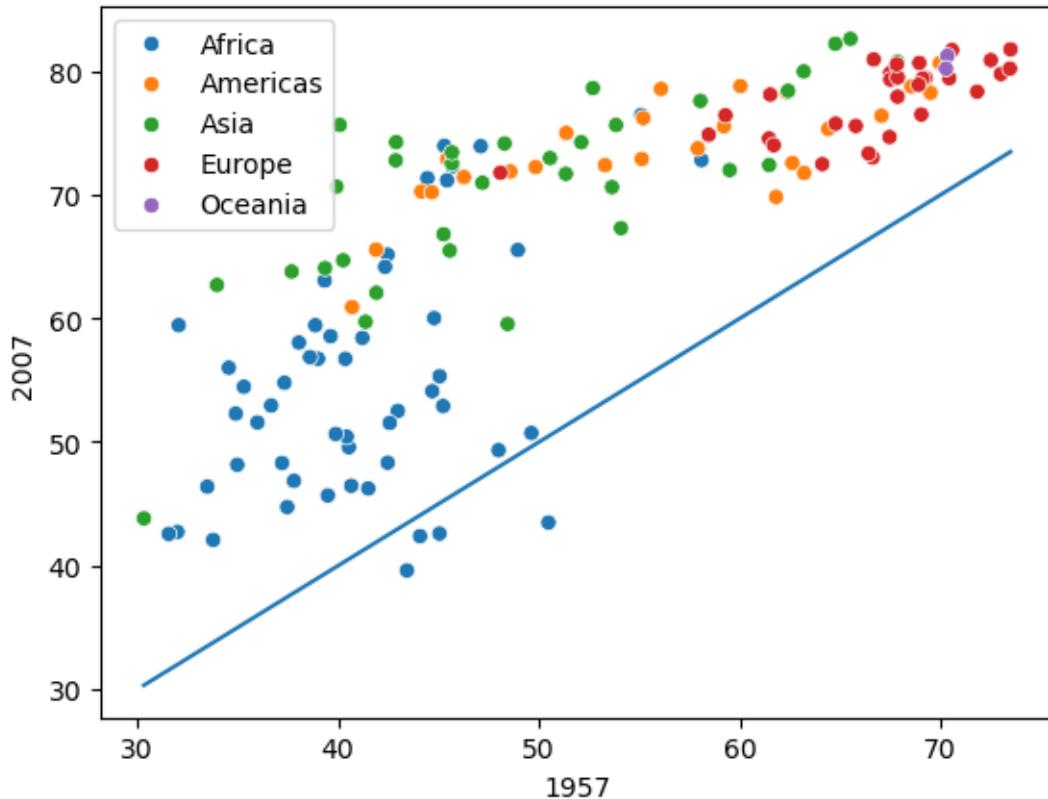
- **values:** Specifies the column to aggregate (e.g., `lifeExp` in our example).
- **index:** Groups by the rows based on the `continent` and `country` column.
- **columns:** Groups by the columns based on the `year` column.
- **aggfunc:** Uses `mean` to calculate the average life expectancy.

With values of `year` as columns, it is easy to compare any metric for different years.

```
#visualizing the change in life expectancy of all countries in 2007 as compared to that in 1957
```

```
sns.scatterplot(data = gdp_lifeExp_data_pivot, x = 1957,y=2007,hue = 'continent')
```

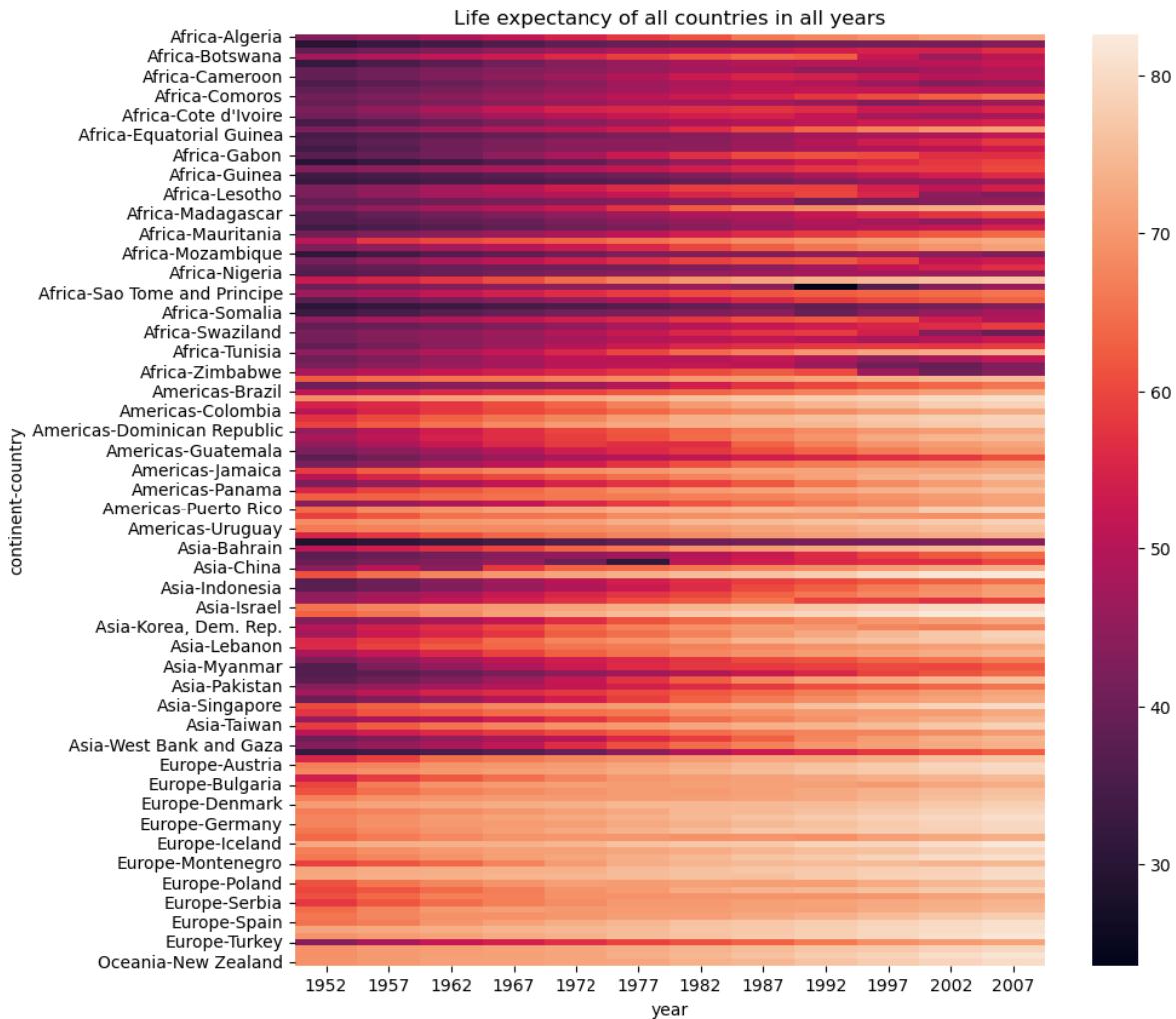
```
sns.lineplot(data = gdp_lifeExp_data_pivot, x = 1957,y = 1957);
```



Observe that for some African countries, the life expectancy has decreased after 50 years. It is worth investigating these countries to identify factors associated with the decrease.

Another way to visualize a pivot table is by using a heatmap, which represents the 2D table as a matrix of colors, making it easier to interpret patterns and trends.

```
# Plotting a heatmap to visualize the life expectancy of all countries in all years, figure size 10x10
plt.figure(figsize=(10,10))
plt.title("Life expectancy of all countries in all years")
sns.heatmap(gdp_lifeExp_data_pivot);
```



### Common Aggregation Functions in `pivot_table()`

You can use various aggregation functions within `pivot_table()` to summarize data:

- `mean` – Calculates the average of values within each group.
- `sum` – Computes the total of values within each group.
- `count` – Counts the number of non-null entries within each group.
- `min` and `max` – Finds the minimum and maximum values within each group.

We can also define our own custom aggregation function and pass it to the `aggfunc` parameter of the `pivot_table()` function

**For Example:** Find the 90<sup>th</sup> percentile of life expectancy for each country and year combination

```
pd.pivot_table(data = gdp_lifeExp_data, values = 'lifeExp', index = 'country', columns ='year')
```

year country	1952	1957	1962	1967	1972	1977	1982	1987	1992	1997
Afghanistan	28.801	30.332	31.997	34.020	36.088	38.438	39.854	40.822	41.674	41.763
Albania	55.230	59.280	64.820	66.220	67.690	68.930	70.420	72.000	71.581	72.950
Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744	69.152
Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647	40.963
Argentina	62.485	64.399	65.142	65.634	67.065	68.481	69.942	70.774	71.868	73.275
...	...	...	...	...	...	...	...	...	...	...
Vietnam	40.412	42.887	45.363	47.838	50.254	55.764	58.816	62.820	67.662	70.672
West Bank and Gaza	43.160	45.671	48.127	51.631	56.532	60.765	64.406	67.046	69.718	71.096
Yemen, Rep.	32.548	33.970	35.180	36.984	39.848	44.175	49.113	52.922	55.599	58.020
Zambia	42.038	44.077	46.023	47.768	50.107	51.386	51.821	50.821	46.100	40.238
Zimbabwe	48.451	50.469	52.358	53.995	55.635	57.674	60.363	62.351	60.377	46.809

#### 14.1.4 Melting “wide” to “long” using melt()

Melting is the **inverse** of pivoting. It transforms columns into rows. The `melt()` function is used when you want to unpivot a DataFrame, turning multiple columns into rows.

Let's first consider `gdp_lifeExp_data_pivot` created in the previous section

```
gdp_lifeExp_data_pivot.head()
```

continent	year country
	Algeria
	Angola
Africa	Benin
	Botswana
	Burkina Faso

```
gdp_lifeExp_data_pivot.melt(value_name='lifeExp', var_name='year', ignore_index=False)
```

---

continent	country
Africa	Algeria
	Angola
	Benin
	Botswana
	Burkina Faso
...	...
Europe	Switzerland
	Turkey
	United Kingdom
Oceania	Australia
	New Zealand

---

With the above DataFrame, we can visualize the mean life expectancy against year separately for each country in each continent.

#### 14.1.5 Stacking and Unstacking using `stack()` and `unstack()`

Stacking and unstacking are powerful techniques used to reshape DataFrames by moving data between rows and columns. These operations are particularly useful when working with **multi-level index** or **multi-level columns**, where the data is structured in a hierarchical format.

- `stack()`: Converts columns into rows, which means it “stacks” the data into a long format.
- `unstack()`: Converts rows into columns, reversing the effect of `stack()`.

Let’s use the GDP per capita dataset and create a DataFrame with a multi-level index (e.g., continent, country) and multi-level columns (e.g., `lifeExp`, `gdpPercap`, and `pop`).

```
# calculate the average and standard deviation of life expectancy, population, and GDP per capita
gdp_lifeExp_multilevel_data = gdp_lifeExp_data.groupby(['continent', 'country'])[['lifeExp', 'pop', 'gdpPercap']].agg([np.mean, np.std]).reset_index()
gdp_lifeExp_multilevel_data
```

---

continent	country
Africa	Algeria
	Angola
	...
	...
	...

---

continent	country
	Benin
	Botswana
	Burkina Faso
...	...
Europe	Switzerland
	Turkey
	United Kingdom
Oceania	Australia
	New Zealand

---

The resulting DataFrame has two levels of index and two levels of columns, as shown below:

```
# check the level of row and column index
gdp_lifeExp_multilevel_data.index.nlevels
```

2

```
gdp_lifeExp_multilevel_data.columns.nlevels
```

2

if you want to see the data along with the index and columns (in a more accessible way), you can directly print the `.index` and `.columns` attributes.

```
gdp_lifeExp_multilevel_data.index
```

```
MultiIndex([( 'Africa', 'Algeria'),
( 'Africa', 'Angola'),
( 'Africa', 'Benin'),
( 'Africa', 'Botswana'),
( 'Africa', 'Burkina Faso'),
( 'Africa', 'Burundi'),
( 'Africa', 'Cameroon'),
( 'Africa', 'Central African Republic'),
( 'Africa', 'Chad'),
( 'Africa', 'Comoros'),
```

```
...
('Europe', 'Serbia'),
('Europe', 'Slovak Republic'),
('Europe', 'Slovenia'),
('Europe', 'Spain'),
('Europe', 'Sweden'),
('Europe', 'Switzerland'),
('Europe', 'Turkey'),
('Europe', 'United Kingdom'),
('Oceania', 'Australia'),
('Oceania', 'New Zealand')],
names=['continent', 'country'], length=142)
```

```
gdp_lifeExp_multilevel_data.columns
```

```
MultiIndex([( 'lifeExp', 'mean'),
( 'lifeExp', 'std'),
( 'pop', 'mean'),
( 'pop', 'std'),
('gdpPercap', 'mean'),
('gdpPercap', 'std')],
)
```

As seen, **the index and columns each contain tuples**. This is the reason that when performing data subsetting on a DataFrame with multi-level index and columns, you need to supply a tuple to specify the exact location of the data.

```
gdp_lifeExp_multilevel_data.loc[('Americas', 'United States'), ('lifeExp', 'mean')]
```

73.4785

```
gdp_lifeExp_multilevel_data.columns.values
```

```
array([('lifeExp', 'mean'), ('lifeExp', 'std'), ('pop', 'mean'),
('pop', 'std'), ('gdpPercap', 'mean'), ('gdpPercap', 'std')],
dtype=object)
```

#### 14.1.5.1 Understanding the level in multi-level index and columns

The `gdp_lifeExp_multilevel_data` dataframe have two levels of index and two level of columns, You can reference these levels in various operations, such as grouping, subsetting, or performing aggregations. The `level` parameter helps identify which part of the multi-level structure you're working with.

```
gdp_lifeExp_multilevel_data.index.get_level_values(0)
```

```
Index(['Africa', 'Africa', 'Africa', 'Africa', 'Africa', 'Africa', 'Africa',
       'Africa', 'Africa', 'Africa',
       ...
       'Europe', 'Europe', 'Europe', 'Europe', 'Europe', 'Europe', 'Europe',
       'Europe', 'Oceania', 'Oceania'],
      dtype='object', name='continent', length=142)
```

```
gdp_lifeExp_multilevel_data.index.get_level_values(1)
```

```
gdp_lifeExp_multilevel_data.columns.get_level_values(0)
```

```
Index(['lifeExp', 'lifeExp', 'pop', 'pop', 'gdpPercap', 'gdpPercap'], dtype='object')
```

```
gdp_lifeExp_multilevel_data.columns.get_level_values(1)
```

```
Index(['mean', 'std', 'mean', 'std', 'mean', 'std'], dtype='object')
```

```
gdp_lifeExp_multilevel_data.columns.get_level_values(-1)
```

```
Index(['mean', 'std', 'mean', 'std', 'mean', 'std'], dtype='object')
```

As seen above, the outermost level corresponds to `level=0`, and it increases as we move inward to the inner levels. The innermost level can be referred to as `-1`.

Now, let's use the level number in the `droplevel()` method to see how it works. The syntax for this method is as follows:

```
DataFrame.droplevel(level, axis=0)
```

```
# drop the first level of row index
gdp_lifeExp_multilevel_data.droplevel(0, axis=0)
```

country	lifeExp		pop		gdpPercap	
	mean	std	mean	std	mean	std
Algeria	59.030167	10.340069	1.987541e+07	8.613355e+06	4426.025973	1310.337656
Angola	37.883500	4.005276	7.309390e+06	2.672281e+06	3607.100529	1165.900251
Benin	48.779917	6.128681	4.017497e+06	2.105002e+06	1155.395107	159.741306
Botswana	54.597500	5.929476	9.711862e+05	4.710965e+05	5031.503557	4178.136987
Burkina Faso	44.694000	6.845792	7.548677e+06	3.247808e+06	843.990665	183.430087
...	...	...	...	...	...	...
Switzerland	75.565083	4.011572	6.384293e+06	8.582009e+05	27074.334405	6886.463308
Turkey	59.696417	9.030591	4.590901e+07	1.667768e+07	4469.453380	2049.665102
United Kingdom	73.922583	3.378943	5.608780e+07	3.174339e+06	19380.472986	7388.189399
Australia	74.662917	4.147774	1.464931e+07	3.915203e+06	19980.595634	7815.405220
New Zealand	73.989500	3.559724	3.100032e+06	6.547108e+05	17262.622813	4409.009167

```
# drop the first level of column index
gdp_lifeExp_multilevel_data.droplevel(0, axis=1)
```

continent	country
Africa	Algeria
	Angola
	Benin
	Botswana
	Burkina Faso
...	...
Europe	Switzerland
	Turkey
	United Kingdom
	Australia
Oceania	New Zealand

#### 14.1.5.2 Stacking (Columns to Rows ): stack()

The `stack()` function pivots the columns into rows. You can specify which level you want to stack using the `level` parameter. By default, it will stack the innermost level of the columns.

```
gdp_lifeExp_multilevel_data.stack(future_stack=True)
```

---

continent	country
	Algeria
Africa	Angola
	Benin
...	...
Europe	United Kingdom
	Australia
Oceania	New Zealand

---

Let's change the default setting by specifying the `level=0`(the outmost level)

```
gdp_lifeExp_multilevel_data.stack(level=0, future_stack=True)
```

---

continent	country
	Algeria
Africa	Angola
	...
...	...
Oceania	Australia
	New Zealand

---

#### 14.1.5.3 Unstacking (Rows to Columns) : `unstack()`

The `unstack()` function pivots the rows back into columns. By default, it will unstack the innermost level of the index.

Let's reverse the previous dataframe to its original shape using `unstack()`.

```
gdp_lifeExp_multilevel_data.stack(level=0, future_stack=True).unstack()
```

continent	country
Africa	Algeria
	Angola
	Benin
	Botswana
	Burkina Faso
...	...
Europe	Switzerland
...	Turkey
Oceania	United Kingdom
	Australia
	New Zealand

#### 14.1.6 Transposing using .T attribute

If you simply want to swap rows and columns, you can use the `.T` attribute.

```
gdp_lifeExp_multilevel_data.T
```

	continent	Afr
	country	Alg
lifeExp	mean	5.90
	std	1.03
pop	mean	1.98
	std	8.61
gdpPercap	mean	4.41
	std	1.31

#### 14.1.7 Summary of Common Reshaping Functions

Function	Description	Example
<code>pivot()</code>	Reshape data by creating new columns from existing ones.	Pivot data on index/columns.
<code>melt()</code>	Unpivot data, turning columns into rows.	Convert wide format to long.
<code>stack()</code>	Convert columns to rows.	Move column data to rows.
<code>unstack()</code>	Convert rows back to columns.	Move row data to columns.
<code>.T</code>	Transpose the DataFrame (swap rows and columns).	Transpose the entire DataFrame.

### Practical Use Cases:

- **Pivoting:** Useful for time series analysis or when you want to group and summarize data by specific categories.
- **Melting:** Helpful when you need to reshape wide data for easier analysis or when preparing data for machine learning algorithms.
- **Stacking/Unstacking:** Useful when you are working with hierarchical index DataFrames.
- **Transpose:** Helpful for reorienting data for analysis or visualization.

#### 14.1.8 Converting from Multi-Level Index to Single-Level Index DataFrame

If you're uncomfortable working with DataFrames that have multi-level indexes and columns, you can convert them into single-level DataFrames using the methods you learned in the previous chapter:

- `reset_index()`
- Flattening the multi-level columns

```
# reset the index to convert the multi-level index to a single-level index
gdp_lifeExp_multilevel_data.reset_index()
```

	continent	country	lifeExp		pop		gdpPercap	
			mean	std	mean	std	mean	std
0	Africa	Algeria	59.030167	10.340069	1.987541e+07	8.613355e+06	4426.025973	1
1	Africa	Angola	37.883500	4.005276	7.309390e+06	2.672281e+06	3607.100529	1
2	Africa	Benin	48.779917	6.128681	4.017497e+06	2.105002e+06	1155.395107	1
3	Africa	Botswana	54.597500	5.929476	9.711862e+05	4.710965e+05	5031.503557	4
4	Africa	Burkina Faso	44.694000	6.845792	7.548677e+06	3.247808e+06	843.990665	1
...	...	...	...	...	...	...	...	...
137	Europe	Switzerland	75.565083	4.011572	6.384293e+06	8.582009e+05	27074.334405	6

	continent	country	lifeExp		pop		gdpPerCap	
			mean	std	mean	std	mean	std
138	Europe	Turkey	59.696417	9.030591	4.590901e+07	1.667768e+07	4469.453380	2
139	Europe	United Kingdom	73.922583	3.378943	5.608780e+07	3.174339e+06	19380.472986	7
140	Oceania	Australia	74.662917	4.147774	1.464931e+07	3.915203e+06	19980.595634	7
141	Oceania	New Zealand	73.989500	3.559724	3.100032e+06	6.547108e+05	17262.622813	4

```
# flatten the multi-level column
gdp_lifeExp_multilevel_data.columns = ['_'.join(col).strip() for col in gdp_lifeExp_multilevel_data.columns]
gdp_lifeExp_multilevel_data
```

continent	country
Africa	Algeria
Africa	Angola
Africa	Benin
Africa	Botswana
Africa	Burkina Faso
Africa	...
Europe	...
Europe	Switzerland
Europe	Turkey
Europe	United Kingdom
Oceania	Australia
Oceania	New Zealand

## 14.2 Data Enriching

Data enriching involves enhancing your existing dataset by adding additional data, often from external sources, to make your dataset more insightful.

left	right	right2	Result																																																																																																	
<table border="1"> <tr> <th></th> <th>A</th> <th>B</th> <th>key2</th> </tr> <tr> <td>K0</td> <td>A0</td> <td>B0</td> <td>K0</td> </tr> <tr> <td>K0</td> <td>A1</td> <td>B1</td> <td>K1</td> </tr> <tr> <td>K1</td> <td>A2</td> <td>B2</td> <td>K0</td> </tr> <tr> <td>K2</td> <td>A3</td> <td>B3</td> <td>K1</td> </tr> </table>		A	B	key2	K0	A0	B0	K0	K0	A1	B1	K1	K1	A2	B2	K0	K2	A3	B3	K1	<table border="1"> <tr> <th></th> <th>C</th> <th>D</th> <th>key2</th> </tr> <tr> <td>K0</td> <td>C0</td> <td>D0</td> <td>K0</td> </tr> <tr> <td>K1</td> <td>C1</td> <td>D1</td> <td>K0</td> </tr> <tr> <td>K2</td> <td>C2</td> <td>D2</td> <td>K0</td> </tr> <tr> <td>K2</td> <td>C3</td> <td>D3</td> <td>K1</td> </tr> </table>		C	D	key2	K0	C0	D0	K0	K1	C1	D1	K0	K2	C2	D2	K0	K2	C3	D3	K1	<table border="1"> <tr> <td>K1</td> <td>V</td> </tr> <tr> <td>K1</td> <td>7</td> </tr> <tr> <td>K1</td> <td>8</td> </tr> <tr> <td>K2</td> <td>9</td> </tr> </table>	K1	V	K1	7	K1	8	K2	9	<table border="1"> <tr> <th>A</th> <th>B</th> <th>key2_x</th> <th>C</th> <th>D</th> <th>key2_y</th> <th>V</th> </tr> <tr> <td>K0</td> <td>A0</td> <td>B0</td> <td>K0</td> <td>C0</td> <td>D0</td> <td>NaN</td> </tr> <tr> <td>K0</td> <td>A1</td> <td>B1</td> <td>K1</td> <td>C0</td> <td>D0</td> <td>NaN</td> </tr> <tr> <td>K1</td> <td>A2</td> <td>B2</td> <td>K0</td> <td>C1</td> <td>D1</td> <td>7.0</td> </tr> <tr> <td>K1</td> <td>A2</td> <td>B2</td> <td>K0</td> <td>C1</td> <td>D1</td> <td>8.0</td> </tr> <tr> <td>K2</td> <td>A3</td> <td>B3</td> <td>K1</td> <td>C2</td> <td>D2</td> <td>9.0</td> </tr> <tr> <td>K2</td> <td>A3</td> <td>B3</td> <td>K1</td> <td>C3</td> <td>D3</td> <td>9.0</td> </tr> </table>	A	B	key2_x	C	D	key2_y	V	K0	A0	B0	K0	C0	D0	NaN	K0	A1	B1	K1	C0	D0	NaN	K1	A2	B2	K0	C1	D1	7.0	K1	A2	B2	K0	C1	D1	8.0	K2	A3	B3	K1	C2	D2	9.0	K2	A3	B3	K1	C3	D3	9.0
	A	B	key2																																																																																																	
K0	A0	B0	K0																																																																																																	
K0	A1	B1	K1																																																																																																	
K1	A2	B2	K0																																																																																																	
K2	A3	B3	K1																																																																																																	
	C	D	key2																																																																																																	
K0	C0	D0	K0																																																																																																	
K1	C1	D1	K0																																																																																																	
K2	C2	D2	K0																																																																																																	
K2	C3	D3	K1																																																																																																	
K1	V																																																																																																			
K1	7																																																																																																			
K1	8																																																																																																			
K2	9																																																																																																			
A	B	key2_x	C	D	key2_y	V																																																																																														
K0	A0	B0	K0	C0	D0	NaN																																																																																														
K0	A1	B1	K1	C0	D0	NaN																																																																																														
K1	A2	B2	K0	C1	D1	7.0																																																																																														
K1	A2	B2	K0	C1	D1	8.0																																																																																														
K2	A3	B3	K1	C2	D2	9.0																																																																																														
K2	A3	B3	K1	C3	D3	9.0																																																																																														

There are 3 common methods for data enriching:

- **Merging using `merge()`:** Combining datasets based on a common key.
- **Joining using `join()`:** Using indices to combine data.
- **Concatenation using `concat()`:** Stacking DataFrames vertically or horizontally.

Let's first load the existing data

```
df_existing = pd.read_csv('./datasets/LOTR.csv')
print(df_existing.shape)
df_existing.head()
```

(4, 3)

	FellowshipID	FirstName	Skills
0	1001	Frodo	Hiding
1	1002	Samwise	Gardening
2	1003	Gandalf	Spells
3	1004	Pippin	Fireworks

Let's next load the external data we want to combine to the existing data

```
df_external = pd.read_csv("./datasets/LOTR_2.csv")
print(df_external.shape)
df_external.head()
```

(5, 3)

	FellowshipID	FirstName	Age
0	1001	Frodo	50
1	1002	Samwise	39
2	1006	Legolas	2931
3	1007	Elrond	6520
4	1008	Barromir	51

#### 14.2.1 Combining based on common keys: `merge()`

```
df_enriched_merge = pd.merge(df_existing, df_external)
print(df_enriched_merge.shape)
df_enriched_merge.head()
```

(2, 4)

	FellowshipID	FirstName	Skills	Age
0	1001	Frodo	Hiding	50
1	1002	Samwise	Gardening	39

You can also do this way

```
df_enriched_merge = df_existing.merge(df_external)
print(df_enriched_merge.shape)
df_enriched_merge.head()
```

(2, 4)

	FellowshipID	FirstName	Skills	Age
0	1001	Frodo	Hiding	50
1	1002	Samwise	Gardening	39

By default, the `merge()` function in pandas performs an **inner join**, which combines two DataFrames based on the **common keys shared by both**. Only the rows with matching keys in both DataFrames are retained in the result.

This behavior can be observed in the following code:

```
df_enriched_merge = pd.merge(df_existing, df_external, how='inner')
print(df_enriched_merge.shape)
df_enriched_merge.head()
```

(2, 4)

	FellowshipID	FirstName	Skills	Age
0	1001	Frodo	Hiding	50
1	1002	Samwise	Gardening	39

**When you may use inner join?** You should use inner join when you cannot carry out the analysis unless the observation corresponding to the *key column(s)* is present in both the tables.

**Example:** Suppose you wish to analyze the association between vaccinations and covid infection rate based on country-level data. In one of the datasets, you have the infection rate for each country, while in the other one you have the number of vaccinations in each country. The countries which have either the vaccination or the infection rate missing, cannot help analyze the association. In such a case you may be interested only in countries that have values for both the variables. Thus, you will use inner join to discard the countries with either value missing.

In `merge()`, the `how` parameter specifies the type of merge to be performed, the default option is `inner`, it could also be `left`, `right`, and `outer`.

These join types give you flexibility in how you want to merge and combine data from different sources.

Let's see how each of these works using our dataframes defined above

### Left Join

- Returns all rows from the left DataFrame and the matching rows from the right DataFrame.
- Missing values (`NaN`) are introduced for non-matching keys from the right DataFrame.

```
df_merge_lft = pd.merge(df_existing, df_external, how='left')
print(df_merge_lft.shape)
df_merge_lft.head()
```

(4, 4)

	FellowshipID	FirstName	Skills	Age
0	1001	Frodo	Hiding	50.0
1	1002	Samwise	Gardening	39.0
2	1003	Gandalf	Spells	NaN
3	1004	Pippin	Fireworks	NaN

**When you may use left join?** You should use left join when the primary variable(s) of interest are present in the one of the datasets, and whose missing values cannot be imputed. The variable(s) in the other dataset may not be as important or it may be possible to reasonably impute their values, if missing corresponding to the observation in the primary dataset.

### Examples:

- 1) Suppose you wish to analyze the association between the covid infection rate and the government effectiveness score (a metric used to determine the effectiveness of the government in implementing policies, upholding law and order etc.) based on the data of all countries. Let us say that one of the datasets contains the covid infection rate, while the other one contains the government effectiveness score for each country. If the infection rate for a country is missing, it might be hard to impute. However, the government effectiveness score may be easier to impute based on GDP per capita, crime rate etc. - information that is easily available online. In such a case, you may wish to use a left join where you keep all the countries for which the infection rate is known.
- 2) Suppose you wish to analyze the association between demographics such as age, income etc. and the amount of credit card spend. Let us say one of the datasets contains the demographic information of each customer, while the other one contains the credit card spend for the customers who made at least one purchase. In such a case, you may want to do a left join as customers not making any purchase might be absent in the card spend data. Their spend can be imputed as zero after merging the datasets.

### Right join

- Returns all rows from the right DataFrame and the matching rows from the left DataFrame.
- Missing values (NaN) are introduced for non-matching keys from the left DataFrame.

```
df_merge_right = pd.merge(df_existing, df_external, how='right')
print(df_merge_right.shape)
df_merge_right.head()
```

(5, 4)

	FellowshipID	FirstName	Skills	Age
0	1001	Frodo	Hiding	50
1	1002	Samwise	Gardening	39
2	1006	Legolas	NaN	2931
3	1007	Elrond	NaN	6520
4	1008	Barromir	NaN	51

**When you may use right join?** You can always use a left join instead of a right join. Their purpose is the same.

#### 14.2.1.1 outer join

- Returns all rows from both DataFrames, with `NaN` values where no match is found.

```
df_merge_outer = pd.merge(df_existing, df_external, how='outer')
print(df_merge_outer.shape)
df_merge_outer
```

(7, 4)

	FellowshipID	FirstName	Skills	Age
0	1001	Frodo	Hiding	50.0
1	1002	Samwise	Gardening	39.0
2	1003	Gandalf	Spells	NaN
3	1004	Pippin	Fireworks	NaN
4	1006	Legolas	NaN	2931.0
5	1007	Elrond	NaN	6520.0
6	1008	Barromir	NaN	51.0

**When you may use outer join?** You should use an outer join when you cannot afford to lose data present in either of the tables. All the other joins may result in loss of data.

**Example:** Suppose I took two course surveys for this course. If I need to analyze student sentiment during the course, I will take an outer join of both the surveys. Assume that each survey is a dataset, where each row corresponds to a unique student. Even if a student has answered one of the two surveys, it will be indicative of the sentiment, and will be useful to keep in the merged dataset.

#### 14.2.2 Combining based on Indices: `join()`

Unlike `merge()`, `join()` in pandas combines data based on the `index`. When using `join()`, you must specify the `suffixes` parameter if there are overlapping column names, as it does not have default values. You can refer to the official function definition here: [pandas.DataFrame.join](#).

Please copy the code below and run it in your notebook:

```

df_enriched_join = df_existing.join(df_external)
print(df_enriched_join.shape)
df_enriched_join.head()

```

Let's set the suffix to fix this issue

```

df_enriched_join = df_existing.join(df_external, lsuffix = '_existing', rsuffix = '_external')
print(df_enriched_join.shape)
df_enriched_join.head()

```

(4, 6)

	FellowshipID_existing	FirstName_existing	Skills	FellowshipID_external	FirstName_external
0	1001	Frodo	Hiding	1001	Frodo
1	1002	Samwise	Gardening	1002	Samwise
2	1003	Gandalf	Spells	1006	Legolas
3	1004	Pippin	Fireworks	1007	Elrond

As can be observed, **join()** merges DataFrames using their index and performs a **left join**.

Now, let's set the "FellowshipID" column as the index in both DataFrames and see what happens.

```

# set the index of the dataframes as the "FellowshipID" column
df_existing.set_index('FellowshipID', inplace=True)
df_external.set_index('FellowshipID', inplace=True)

```

```

# join the two dataframes
df_enriched_join = df_existing.join(df_external, lsuffix = '_existing', rsuffix = '_external')
print(df_enriched_join.shape)
df_enriched_join

```

(4, 4)

FellowshipID	FirstName_existing	Skills	FirstName_external	Age
1001	Frodo	Hiding	Frodo	50.0
1002	Samwise	Gardening	Samwise	39.0
1003	Gandalf	Spells	NaN	NaN
1004	Pippin	Fireworks	NaN	NaN

In this case, `df_enriched_join` will contain all rows from `df_existing` and the corresponding matching rows from `df_external`, with `NaN` values for non-matching entries.

Similar to `merge()`, you can change the type of join in `join()` by explicitly specifying the `how` parameter. However, this explanation will be skipped here, as the different join types have already been covered in the `merge()` section above.

#### 14.2.3 Stacking vertically or horizontally: `concat()`

`concat()` is used in pandas to concatenate DataFrames along a specified axis, either rows or columns. Similar to `concat()` in NumPy, it defaults to concatenating along the rows.

```
# let's concatenate the two dataframes using default setting
df_concat = pd.concat([df_existing, df_external])
print(df_concat.shape)
df_concat
```

(9, 3)

FellowshipID	FirstName	Skills	Age
1001	Frodo	Hiding	NaN
1002	Samwise	Gardening	NaN
1003	Gandalf	Spells	NaN
1004	Pippin	Fireworks	NaN
1001	Frodo	NaN	50.0
1002	Samwise	NaN	39.0
1006	Legolas	NaN	2931.0
1007	Elrond	NaN	6520.0
1008	Barromir	NaN	51.0

```
# Concatenating along columns (horizontal)
df_concat_columns = pd.concat([df_existing, df_external], axis=1)
print(df_concat_columns.shape)
df_concat_columns
```

(7, 4)

FellowshipID	FirstName	Skills	FirstName	Age
1001	Frodo	Hiding	Frodo	50.0
1002	Samwise	Gardening	Samwise	39.0
1003	Gandalf	Spells	NaN	NaN
1004	Pippin	Fireworks	NaN	NaN
1006	NaN	NaN	Legolas	2931.0
1007	NaN	NaN	Elrond	6520.0
1008	NaN	NaN	Barromir	51.0

#### 14.2.4 Missing values after data enriching

Using either of the approaches mentioned above may introduce missing values for unmatched entries. Handling these missing values is crucial for maintaining the quality and integrity of your dataset after merging or joining DataFrames. It is important to choose a method that best fits your data and analysis goals, and to document your decisions regarding missing value handling for transparency in your analysis.

**Question:** Read the documentations of the Pandas DataFrame methods `merge()` and `concat()`, and identify the differences. Mention examples when you can use (i) either, (ii) only `concat()`, (iii) only `merge()`

**Solution:**

- (i) If we need to merge datasets using row indices, we can use either function.
- (ii) If we need to stack datasets one on top of the other, we can only use `concat()`
- (iii) If we need to merge datasets using overlapping columns we can only use `merge()`

For a comprehensive user guide, please refer to the [official documentation](#).

## 14.3 Independent Study

### 14.3.1 Practice exercise 1

You are given the life expectancy data of each continent as a separate \*.csv file.

#### 14.3.1.1

Visualize the change of life expectancy over time for different continents.

#### 14.3.1.2

Appending all the data files, i.e., stacking them on top of each other to form a combined datasets called “data\_all\_continents”

### 14.3.2 Practice exercise 2

#### 14.3.2.1 Preparing GDP per capita data

Read the GDP per capita data from [https://en.wikipedia.org/wiki/List\\_of\\_countries\\_by\\_GDP\\_\(nominal\)\\_per\\_capita](https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_capita)

Drop all the `Year` columns. Use the `drop()` method with the `columns`, `level` and `inplace` arguments. Print the first 2 rows of the updated DataFrame. If the first row of the DataFrame has missing values for all columns, drop that row as well.

#### 14.3.2.2

Drop the inner level of column labels. Use the `droplevel()` method. Print the first 2 rows of the updated DataFrame.

#### 14.3.2.3

Convert the columns consisting of GDP per capita by *IMF*, *World Bank*, and the *United Nations* to numeric. Apply a lambda function on these columns to convert them to numeric. Print the number of missing values in each column of the updated DataFrame.

Note: *Do not apply the function 3 times. Apply it once on a DataFrame consisting of these 3 columns.*

#### 14.3.2.4

Apply the lambda function below on all the column names of the dataset obtained in the previous question to clean the column names.

```
import re
column_name_cleaner = lambda x:re.split(r'\|/', x)[0]
```

*Note: You will need to edit the parameter of the function, i.e., x in the above function to make sure it is applied on column names and not columns.*

Print the first 2 rows of the updated DataFrame.

#### 14.3.2.5

Create a new column `GDP_per_capita` that copies the GDP per capita values of the `United Nations`. If the GDP per capita is missing in the `United Nations` column, then copy it from the `World Bank` column. If the GDP per capita is missing both in the `United Nations` and the `World Bank` columns, then copy it from the `IMF` column.

Print the number of missing values in the `GDP_per_capita` column.

#### 14.3.2.6

Drop all the columns except `Country` and `GDP_per_capita`. Print the first 2 rows of the updated DataFrame.

#### 14.3.2.7

The country names contain some special characters (*characters other than letters*) and need to be cleaned. The following function can help clean country names:

```
import re
country_names_clean_gdp_data = lambda x: re.sub(r'[^w\s]', ' ', x).strip()
```

Apply the above lambda function on the country column to clean country names. Save the cleaned dataset as `gdp_per_capita_data`. Print the first 2 rows of the updated DataFrame.

#### 14.3.2.8 Preparing population data

Read the population data from [https://en.wikipedia.org/wiki/List\\_of\\_countries\\_by\\_population\\_\(United\\_Nations,\\_July\\_2023\)](https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations,_July_2023))

- Drop all columns except Country, UN Continental Region[1], and Population (1 July 2023).
- Drop the first row since it is the total population in the world

#### 14.3.2.9

Apply the lambda function below on all the column names of the dataset obtained in the previous question to clean the column names.

```
import re
column_name_cleaner = lambda x:re.split(r'\|/\|\(| ', x.name)[0]
```

*Note: You will need to edit the parameter of the function, i.e., x in the above function to make sure it is applied on column names and not columns.*

Print the first 2 rows of the updated DataFrame.

#### 14.3.2.10

The country names contain some special characters (*characters other than letters*) and need to be cleaned. The following function can help clean country names:

```
import re
country_names_clean_population_data = lambda x: re.sub("[\(\[].*?\[\)\]]", "", x).strip()
```

Apply the above lambda function on the country column to clean country names. Save the cleaned dataset as `population_data`.

#### 14.3.2.11 Merging GDP per capita and population datasets

Merge `gdp_per_capita_data` with `population_data` to get the population and GDP per capita of countries in a single dataset. Print the first two rows of the merged DataFrame.

Assume that:

1. We want to keep the GDP per capita of all countries in the merged dataset, even if their population is unavailable in the population dataset. For countries whose population is unavailable, their Population column will show NA.

2. We want to discard an observation of a country if its GDP per capita is unavailable.

#### 14.3.2.12

For how many countries in `gdp_per_capita_data` does the population seem to be unavailable in `population_data`? Note that you don't need to clean country names any further than cleaned by the functions provided.

Print the observations of `gdp_per_capita_data` with missing Population.

#### 14.3.3 Merging datasets with *similar* values in the key column

We suspect that population of more countries may be available in `population_data`. However, due to unclean country names, the observations could not merge. For example, the country *Guinea Bissau* is mentioned as *GuineaBissau* in `gdp_per_capita_data` and *Guinea-Bissau* in `population_data`. To resolve this issue, we'll use a different approach to merge dataframes. We'll merge the population of a country to an observation in the GDP per capita dataset, whose name in `population_data` is the most '*similar*' to the name of the country in `gdp_per_capita_data`.

#### 14.3.3.1

Proceed as follows:

1. For each country in `gdp_per_capita_data`, find the country with the most '*similar*' name in `population_data`, based on the similarity score. Use the lambda function provided below to compute the similarity score between two strings (*The higher the score, the more similar are the strings. The similarity score is 1.0 if two strings are exactly the same*).
2. Merge the population of the most '*similar*' country to the country in `gdp_per_capita_data`. The merged dataset must include 5 columns - the country name as it appears in `gdp_per_capita_data`, the GDP per capita, the country name of the most '*similar*' country as it appears in `population_data`, the population of that country, and the similarity score between the country names.
3. After creating the merged dataset, **print** the rows of the dataset that have similarity scores less than 1.

Use the function below to compute the similarity score between the `Country` names of the two datasets:

```
from difflib import SequenceMatcher  
similar = lambda a,b: SequenceMatcher(None, a, b).ratio()
```

**Note:** You may use one for loop only for this particular question. However, it would be perfect if don't use a for loop

**Hint:**

1. Define a function that computes the index of the observation having the most '*similar*' country name in `population_data` for an observation in `gdp_per_capita_data`. The function returns a Series consisting of the most '*similar*' country name, its population, and its similarity score (*This function can be written with only one line in its body, excluding the return statement and the definition statement. However, you may use as many lines as you wish*).
2. Apply the function on the `Country` column of `gdp_per_capita_data`. A DataFrame will be obtained.
3. Concatenate the DataFrame obtained in (2) with `gdp_per_capita_data` with the pandas `concat()` function.

#### 14.3.3.2

In the dataset obtained in the previous question, for all observations where similarity score is less than 0.8, replace the population with `Nan`.

Print the observations of the dataset having missing values of population.

# A Assignment A

Python Iterables and Data Structures

## Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the *Code* cells and your answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
3. Use [Quarto](#) to print the *.ipynb* file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. The assignment is worth 100 points, and is due on **5th October 2025 at 11:59 pm**.
5. **Five points are properly formatting the assignment.** The breakdown is as follows:
  - Must be an HTML file rendered using Quarto (2 pts).
  - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) (1 pt)
  - Final answers of each question are written in Markdown cells (1 pt).
  - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

## A.1 GDP per Capita (35 pts)

### A.1.1 List Comprehension

USA's GDP per capita from 1960 to 2021 is given by the tuple T in the code cell below. The values are arranged in ascending order of the year, i.e., the first value is for 1960, the second value is for 1961, and so on.

```
T = (3007, 3067, 3244, 3375, 3574, 3828, 4146, 4336, 4696, 5032, 5234, 5609, 6094, 6726, 7226, 7801)
```

#### A.1.1.1

Use list comprehension to create a list of the gaps between consecutive entries in T, i.e, the increase in GDP per capita with respect to the previous year. The list with gaps should look like: [60, 177, ...]. Let the name of this list be `GDP_increase`.

(4 points)

#### A.1.1.2

Use `GDP_increase` to find the maximum gap size, i.e, the maximum increase in GDP per capita.

(1 point)

#### A.1.2

Use list comprehension with `GDP_increase` to find the percentage of gaps that have size greater than \$1000.

(3 points)

#### A.1.2.1

Use list comprehension with `GDP_increase` to print the list of years in which the GDP per capita increase was more than \$2000.

**Hint:** The `enumerate()` function may help.

(4 points)

### A.1.2.2

Use list comprehension to:

1. Create a list that consists of the difference between the maximum and minimum GDP per capita values for each of the 5 year-periods starting from 1976, i.e., for the periods 1976-1980, 1981-1985, 1986-1990, ..., 2016-2020.
2. Find the five year period in which the difference (*between the maximum and minimum GDP per capita values*) was the least.

(4 + 2 points)

### A.1.3 Dictionary

The GDP per capita of USA for most years from 1960 to 2021 is given by the dictionary D given in the code cell below.

```
D = {'1960':3007, '1961':3067, '1962':3244, '1963':3375, '1964':3574, '1965':3828, '1966':4146, '1967':4452, '1968':4757, '1969':5062, '1970':5367, '1971':5672, '1972':5977, '1973':6282, '1974':6587, '1975':6892, '1976':7197, '1977':7502, '1978':7807, '1979':8112, '1980':8417, '1981':8722, '1982':9027, '1983':9332, '1984':9637, '1985':9942, '1986':10247, '1987':10552, '1988':10857, '1989':11162, '1990':11467, '1991':11772, '1992':12077, '1993':12382, '1994':12687, '1995':12992, '1996':13297, '1997':13602, '1998':13907, '1999':14212, '2000':14517, '2001':14822, '2002':15127, '2003':15432, '2004':15737, '2005':16042, '2006':16347, '2007':16652, '2008':16957, '2009':17262, '2010':17567, '2011':17872, '2012':18177, '2013':18482, '2014':18787, '2015':19092, '2016':19397, '2017':19602, '2018':19807, '2019':20012, '2020':20217, '2021':20422}
```

Find: ##### The GDP per capita in 2015

(2 points)

#### A.1.3.1

The GDP per capita of 2014 is missing. Update the dictionary to include the GDP per capita of 2014 as the average of the GDP per capita of 2013 and 2015.

(5 points)

#### A.1.3.2

Impute the GDP per capita of other missing years in the same manner as in (2), i.e., as the average GDP per capita of the previous year and the next year. Note that the GDP per capita is not missing for any two consecutive years.

(5 points)

### A.1.3.3

Print the years and the imputed GDP per capita for the years having a missing value of GDP per capita in (3).

(5 points)

## A.2 Student Survey (40 pts)

### A.2.1 Marriage age responses

Below is a list consisting of responses to the question: “At what age do you think you will marry?” from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age=['24','30','28','29','30','27','26','28','30+','26','28','30','30','30','pr
```

Use list comprehension to:

#### A.2.1.1

Remove the elements that are not integers - such as ‘probably never’, ‘30+’, etc. What is the length of the new list?

**Hint:** The built-in python function of the `str` class - `isdigit()` may be useful to check if the string contains only digits.

(3 points)

#### A.2.1.2

Cap the values greater than 80 to 80, in the clean list obtained in (1). What is the mean age when people expect to marry in the new list?

(3 + 4 points)

#### A.2.1.3

Determine the percentage of people who expect to marry at an age of 30 or more.

(5 points)

#### A.2.2 Majors/Minors: Nested lists of student majors.

Below is the list consisting of the majors / minors of students of the course STAT303-1 Fall 2023. This data is a list of lists, where each sub-list (*smaller list within the outer larger list*) consists of the majors / minors of a student. Most of the students have majors / minors in one or more of these four areas:

1. Math / Statistics / Computer Science
  2. Humanities / Communication
  3. Social Sciences / Education
  4. Physical Sciences / Natural Sciences / Engineering

There are some students having majors / minors in other areas as well.

Use list comprehension for all the questions below.

### A.2.2.1

How many students have major / minor in any three of the above mentioned four areas?

(1 point)

## A.2.2.2

How many students have *Math / Statistics / Computer Science* as an area of their major / minor?

**Hint:** Nested list comprehension

(4 points)

### A.2.2.3

How many students have *Math / Statistics / Computer Science* as the **only area** of their major / minor?

**Hint:** Nested list comprehension

(5 points)

### A.2.3

How many students have *Math / Statistics / Computer Science* and *Social Sciences / Education* as a couple of areas of their major / minor?

**Hint:** The in-built function `all()` may be useful.

(6 points)

### A.2.4 Starting salary expectations

Below is a list consisting of responses to the question: “What do you expect your starting salary to be after graduation, to the nearest thousand dollars? (ex: 47000)” from students of the STAT303-1 Fall 2023. class.

```
expected_salary = ['90000', '110000', '100000', '90k', '80000', '47000', '100000', '70000',
```

Clean `expected_salary` using list comprehensions only, and find the mean expected salary.

(5 + 4 points)

## A.3 Starbucks Drinks (20 pts)

The code cell below defines an object having the nutrition information of drinks in starbucks. Assume that the manner in which the information is structured is consistent throughout the object.

```
'value': 1}, {'Starbucks_drinks_nutrition': {value: 'Lime'}}, StarbucksRefDarkerIsABeverageNutritonTypeIn: typearil
```

Use the object above to answer the following questions:

### A.3.1

What is the datatype of the object?

(2 points)

#### A.3.1.1

If the object in (1) is a dictionary, what is the datatype of the values of the dictionary?

(2 points)

### **A.3.1.2**

If the object in (1) is a dictionary, what is the datatype of the elements within the values of the dictionary?

*(2 points)*

### **A.3.1.3**

How many calories are there in Iced Coffee?

*(4 points)*

### **A.3.1.4**

Which drink(s) have the highest amount of protein in them, and what is that protein amount?

### **A.3.1.5**

Which drink(s) have a fat content of more than 10g, and what is their fat content?

*(5 points)*

### **A.3.1.6**

Use dictionary-comprehension to print the name and `carb` content of all drinks that have a `carb` content of more than 50 units.

**Hint:** It will be a nested dictionary comprehension.

*(5 points)*

## B Datasets & Templates

Datasets used in the book, and assignment / project templates can be found [here](#)