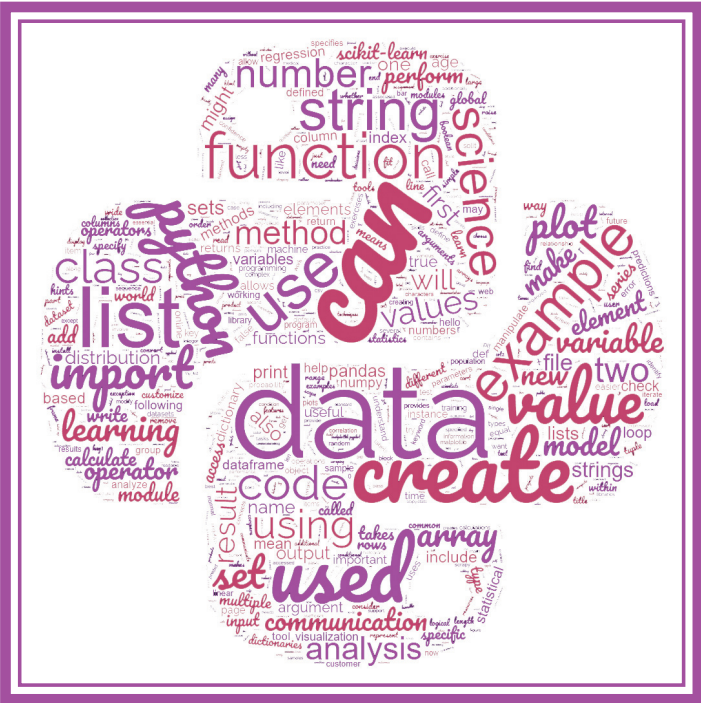


Introduction to DATA SCIENCE

A PYTHON Path for a Non-computer Scientist



Introduction to DATA SCIENCE

**A PYTHON Path for a
Non-computer Scientist**

SEBASTIAN FITZEK

**Introduction to
DATA SCIENCE**

**A PYTHON Path for a
Non-computer Scientist**

comunicare(●)ro

Redactor: ANCA MILU-VAIDSEGAN
Tehnoredactor: CRISTIAN LUPEANU

Toate drepturile asupra prezentei ediții aparțin
Editurii COMUNICARE.RO, 2023.

Editura COMUNICARE.RO este departament în cadrul
Școlii Naționale de Studii Politice și Administrative,
Facultatea de Comunicare și Relații Publice.

Editura COMUNICARE.RO

SNSPA, Facultatea de Comunicare și Relații Publice
Str. Povernei, nr. 6, sector 1
010643, București
România
Tel.: 0372.177.150
www.edituracomunicare.ro
e-mail: editura@comunicare.ro

ISBN 978-973-711-642-0 (ediție electronică)

CONTENTS

<i>Biographical data</i>	/ 11
<i>Acknowledgments</i>	/ 12
<i>Foreword</i>	/ 13
<i>An introduction to Data Science from the perspective of the communication and public relations specialist</i>	/ 17

Part I: PROGRAMMING / 23

Python fundamentals in the introduction to Data Science / 24

Let's get started with Python basics! / 25

How to launch an interactive Python using IDLE	/ 26
Other online shells for Python	/ 27
Introduction to Data Types	/ 28
Let's learn how to code in Python	/ 29
Integers and Floats	/ 30
Basic operators	/ 32
Lesser-Known operators	/ 35
Variables	/ 35
Assignment operators	/ 37
Numbers & variables in the wild	/ 38
★ <i>Magic trick exercise</i>	/ 40
★ <i>Exercises for basic variables in Python</i>	/ 41

Strings Basic / 42

String operators	/ 43
String Indexing	/ 45
String Slices	/ 46
Print() function	/ 47
Escape characters	/ 47
Triple quotes	/ 49
More about strings	/ 51
★ <i>Exercises with strings</i>	/ 52

Introducing functions / 53

Len() function	/ 55
Input	/ 56
Type Casting	/ 58
F Strings	/ 60
★ <i>Age calculator exercise</i>	/ 60
★ <i>Shopping cart exercise</i>	/ 61

The world of methods	/ 63
Introducing methods Upper and Lower	/ 65
Method of navigating documentation in Python	/ 66
Help() & IPython	/ 68
Reading function Signatures + Strip methods	/ 68
Replace()	/ 70
Other very useful string methods for data researchers	/ 71
Method Chaining	/ 71
★ <i>Exercises with string methods</i>	/ 73
Booleans	/ 73
Comparison operators	/ 75
Comparing across types	/ 77
Truthiness & Falseyness	/ 78
The “in” operator	/ 79
Comparing strings	/ 80
★ <i>Exercises with Booleans</i>	/ 83
Conditionals basics	/ 84
Name length codealong	/ 86
A tangent on indentation	/ 87
Nesting conditionals	/ 88
★ <i>Water boiling Codealong</i>	/ 89
★ <i>BMI calculator exercise</i>	/ 89
★ <i>Tweet checker exercise</i>	/ 90
Writing more complex logic	/ 91
Logical AND	/ 92
Logical OR	/ 93
Logical NOT	/ 94
TruthyFalsy testing	/ 95
Logical operator precedence	/ 96
★ <i>Exercises with logical AND, OR, and NOT</i>	/ 97
Loops	/ 98
Avoiding infinite loops	/ 100
The range() function	/ 101
Working with Nested Loops	/ 102
Break and continue keywords	/ 102
★ <i>99 Bottles of Beer Codealong</i>	/ 104
★ <i>Loops problem set</i>	/ 104
★ <i>Snake Eyes Codealong</i>	/ 105
★ <i>Dice Roller Exercise</i>	/ 107
Functions	/ 108
Our very first function!	/ 109
Functions with an Input	/ 110
Functions with multiple arguments	/ 112
Introducing Return!	/ 113

- Using the Return keyword / 113
- Default parameters / 114
- Ordering default parameters / 115
- KeywordNamed argument / 116
- ★ *Function practice set* / 117

Global Scope / 118

- Local Scope / 119
- Scope in loops and conditionals / 120
- Enclosing Scope / 120
- Built-in Scope / 121
- Scope precedence rules / 122
- The ‘Global’ keyword / 123
- ★ *Exercises for understanding the Global Scope* / 124

Lists the basics / 124

- Accessing data in lists / 126
- Updating list elements / 127
- Append() and Extend() / 127
- Insert() / 128
- List Slices / 129
- Deletion methods pop(), popitems(), remove() / 130
- Iterating over lists / 131
- Lists + loops patterns / 133
- Nested lists / 135
- List operators / 136
- Sort(), Reverse(), and Count() / 137
- Lists are mutable / 138
- Comparing lists == vs is / 139
- Join() and Split() / 139
- List unpacking / 140
- Copying lists / 141
- ★ *Exercises with lists* / 143
- ★ *Todo list exercise intro* / 144

Dictionaries / 145

- Creating your Dictionaries / 146
- Accessing data in Dictionaries / 148
- Adding and updating data in Dictionaries / 149
- The Get() method and “in” operator / 150
- Dictionary Pop(), Clear(), and Del() / 151
- Dictionaries are mutable too! / 152
- Iterating Dicts Keys(), Values(), and Items() / 153
- Fancy Dictionary merging / 154
- Lists and Dicts combined / 155
- Fromkeys() / 156
- Update() / 157
- ★ *Peak Dictionary exercise* / 158

Sets and Tuples / 159

Tuple functionality / 161

Sets introduction / 162

Set operators: Intersection, Union, Difference / 163

★ *Exercises with sets* / 165**Back to functions. Introducing args / 165**

Introducing Kwargs / 167

Parameter list ordering / 168

A common gotcha mutable default Args / 169

Unpacking Args / 170

★ *ArgsKwargs Exercises* / 170**Working with Errors / 171**

Common error types / 172

Raising exceptions / 173

When to raise / 174

Try and except / 175

LBYL and EAFP / 176

★ *Exercises with correct error handling* / 177**Modules / 178**

Working with built-in modules / 179

Most popular built-in modules for Data Science / 180

Fancy import syntax / 181

Creating custom modules / 182

3rd party modules Pip & PyPI / 183

Our first Pip package! / 183

★ *Sentiment analysis fun project installation* / 184**Object-Oriented Programming / 185**

Class Syntax / 186

Writing our first class / 187

Instance methods / 188

★ *Practicing Instance methods* / 189

Class Attributes / 190

Class Methods / 191

Inheritance basics / 192

The Super() function / 192

Part II: VISUALIZATION IN DATA SCIENCE**USING THE MOST IMPORTANT PYTHON MODULES / 195****Introduction to Pandas module / 197**

How to install Pandas? / 198

Create a Series in Pandas / 199

Create a DataFrame in Pandas / 200

Read a CSV file with Pandas / 202

Advanced parameters /	203
Selecting rows and columns in Pandas /	204
Data wrangling in Pandas /	206
Arithmetics and statistics in Pandas /	210
Hierarchical indexing in Pandas /	212
Aggregation in Pandas /	214
Data Export in Pandas /	215
Pivot and Pivot Table in Pandas /	216
Visualization in Pandas /	217
★ <i>A few exercises with Pandas</i> /	229
NumPy, a perfect tool for working with Arrays /	230
Matrix Manipulation in Numpy /	237
Array Mathematics in Numpy /	239
Array Manipulation /	244
★ <i>Exercises with NumPy</i> /	251
Let's delve into DataVisualization in Python /	252
DataVisualization with Matplotlib /	253
★ <i>Matplotlib exercises for DataVisualization</i> /	256
Seaborn /	257
★ <i>Some exercises with Seaborn</i> /	265
Web Scraping in Data Science /	266
Scraping websites with Selenium /	272
★ <i>A few exercises with Webscraping</i> /	275
★ <i>Example of Gaussian noise (standard deviation)</i> /	275
Part III: INTRODUCTION TO BUSINESS STATISTICS IN DATA SCIENCE /	279
Big Data, Statistics, and Probability /	280
Business Intelligence (BI) techniques /	281
Big Data and Statistics /	282
Hypothesis testing /	285
Basic probability with Python /	286
Probability in Data Science /	288
Fundamentals of Combinatorics /	289
Bayes' Law /	291
Fundamentals of Probability Distributions /	292
★ <i>A Practical Example of Combinatorics</i> /	294
★ <i>A Practical Example of Bayesian Inference</i> /	295
Descriptive statistics /	297
Statistics with population and sample /	299
Cross Tables and Scatter Plots /	300
Skewness exercise solution /	302
★ <i>Exercises with Histograms in Descriptive Statistics</i> /	303
★ <i>Correlation exercise</i> /	303

Inferential Statistics / 304

Inferential Statistics Confidence Intervals / 306

The Normal Distribution / 308

★ *Exercises with practical examples of Inferential Statistics* / 310**Correlation and Regression / 312**

More about Correlation vs Regression / 314

★ *Exercises with Correlation and Regression* / 315**Time Series Analysis / 316**

Time Series Forecasting / 317

Time Series – Visualization Basics / 320

Time Series – Power Transformation / 321

★ *Exercises with Time Series Analysis* / 322**Part VI: MACHINE LEARNING (optional) / 323****Scikit-learn, a free machine learning for advanced / 323**

Description of the start-up process / 325

Description of the start-up process / 325

Training and Test Data in Scikit-learn / 327

Processing The Data Standardization in Scikit-learn / 329

Normalizer class in Scikit-learn's / 329

Binarization in Scikit-learn / 330

Encoding Categorical Features in Scikit-learn / 331

Imputing Missing Values in Scikit-learn / 333

Generating Polynomial Features in Scikit-learn / 334

Create your model in Scikit-learn / 335

Model Fitting in Scikit-learn / 338

Prediction in Scikit-learn / 339

Evaluate your model's performance in Scikit-learn / 340

Tune your model in Scikit-learn / 341

★ *Exercises with Scikit-learn* / 343★ *How machine learning helps us as Data Scientists* / 344★ *Conclusions and tips for the future Data Science specialist* / 346★ *Where and what online materials**we could read to learn more about Data Science* / 347★ *What kind of jobs can I find in**Communication and Data Science, where, and how?* / 348★ *The impact of Data Science and**Communication on the future of human society* / 350**References / 353**

BIOGRAPHICAL DATA

Dr. Sebastian Fitzek* is a university lecturer at the Faculty of Communication and Public Relations (National School of Political and Administrative Studies) and a scientific researcher at the Institute for Quality-of-Life Research at the Romanian Academy. He specializes in qualitative research methods. He is also the coordinator of the master's Program in Leadership and Political Communication at the Faculty of Communication and Public Relations. Dr. Fitzek's teaching and research focus on data science, leadership and political communication, political science, political anthropology, political sociology, and political psychoanalysis. He has completed postdoctoral research on interethnic imaginary in Bucharest and holds a Ph.D. in Sociology from the West University of Timisoara. Dr. Fitzek is skilled in political communication, political analysis, and political leadership, and has expertise in the use of data science Python tools such as Matplotlib, Panda, Numpy, Sklearn, Seaborn, BS4, Selenium, and Scapy. He also has basic knowledge of R and SQL and is proficient in the use of the Maxqda 2020 analytics program for qualitative methods. From October 2022 to July 2023, he is attending a specialization course in Data Science at the Artificial Intelligence Research Centre "Medical Image Analysis & Artificial Intelligence" in Krems, Austria. In addition to his teaching and research, Dr. Fitzek has good communication skills and is experienced in university teaching, master coordination, and training. His main areas of interest and research are communication, social, and political sciences. Over the past 15 years, he has coordinated several projects and research work in political leadership, public

* Faculty of Communication and Public Relations, National University of Political Studies and Public Administration, Bucharest, Romania.

Medical Image Analysis & Artificial Intelligence, Faculty of Medicine and Dentistry, Danube Private University, 3500 Krems, Austria.

Research Institute for Quality of Life, National Institute for Economic Research "Costin C. Kirilăscu", Romanian Academy Bucharest, Romania.

image, political anthropology, sociology, social work, and public health. Sebastian Fitzek has participated in over 50 conferences and has authored and co-authored nationally and internationally scientifically indexed articles and chapters.

ACKNOWLEDGMENTS

Writing a book is harder than I thought and more rewarding than I could have ever imagined. None of this would have been possible without the encouragement, help, and support of Professor Dr. Alina Bârgăoanu, the Dean of the Faculty of Communication and Public Relations, NUPSPA, to whom I am deeply grateful.

Professor Dr. Anna Choi, Head of Health Services Research Group/Rehabilitation Scientific Group at MIAAI has my profound gratitude for paving the way for my pursuit of Medical Image Analysis & Artificial Intelligence research within the wonderful team of the MIAAI Centre of Excellence in Krems, Austria.

As a Ph.D. supervisor, but also as a mentor throughout my scientific life, I am deeply grateful to Prof. Dr. Elena Zamfir for the constant support, optimism, and confidence she showed me from the day we met on the steps of the Faculty of Sociology and Social Work, University of Bucharest.

The world is better thanks to people who want to develop and lead others. What makes the world even more beautiful is the gift of these inspiring individuals to give their time to mentoring.

FOREWORD

Asking yourself the most important questions is a good way to introduce a book dedicated to Data Science because it helps to set the stage for the material that will be covered in the book. By asking questions, you can get a sense of what the book will be about and what you can expect to learn from it. Additionally, asking questions can help to engage the reader and get them thinking about the topic at hand, which can make the material more interesting and relevant to their own experiences. So, the first legitimate question is why is Data Science today a discipline of great importance for the present and future of master students? For sure, Data Science is a discipline of great importance today because it allows organizations to make better decisions by leveraging the vast amounts of data that are generated in today's world. With the help of Data Science, organizations can gain insights into their operations and customers, and use that information to improve their products, services, and overall business strategies. Additionally, Data Science is a rapidly growing field, with many job opportunities for individuals with the right skills and training. As a result, pursuing a master's degree in communication and Data Science can open a wide range of career possibilities for students.

Another key question is why should communication students be the ones to start and deepen Data Science or what is the connection between communication and public relations students and Data Science? Communication and public relations students should be interested in studying Data Science because it can help them better understand and analyze the vast amounts of data that are generated in today's world. Data Science can provide communication and public relations students with the skills to glean meaningful insights from data, which can be utilized to elevate and refine their practices in these areas. Learning Data Science, students can also gain the ability to extract valuable insights from data that can be applied to enhance their work in communication and public relations. Additionally, Data Sci-

ence can help communication and public relations students to better understand the needs and preferences of their target audience, and to create more effective communication strategies and campaigns. Finally, the skills and knowledge gained from studying Data Science can be highly valuable in the job market and can help communication and public relations students stand out from their peers and advance their careers.

A thirty-concrete key question would be how can it open a career and what are the shortcuts to success for any student wishing to specialize in this field? Studying Data Science can open a wide range of career possibilities for students. Data Science professionals may find employment as data scientists, data analysts, machine learning engineers, or business intelligence analysts. Alternatively, those with a strong background in Data Science may choose to pursue careers as data scientists, data analysts, machine learning engineers, or business intelligence analysts. In these roles, individuals can work in a variety of industries, including technology, finance, healthcare, and government, to help organizations make better decisions using data. To succeed in this field, students should be prepared to learn a wide range of technical skills, such as programming, statistics, and machine learning, as well as soft skills, such as problem-solving and communication. Additionally, students can gain a competitive advantage by participating in internships or other hands-on learning experiences, as well as by staying up to date with the latest developments in the field.

However, the field is very broad and then we should ask ourselves which parts or structures of Data Science are worth learning in the early stages. In this sense, what is worth studying in a master's program of only 2 years, having this limited time, and what is worth didactically deepened in the early stages of initiation? In a master's program with a limited time frame, students need to prioritize the key concepts and skills that are most essential for success in Data Science. Some of the most important areas to focus on in the early stages of a Data Science program include:

- In **Part I** of this book, we will learn **Fundamental Programming in Python: Data Science** involves working with large and complex

datasets, and students will need to be proficient in **Python**, to manipulate and analyze that data.

- In **Part II** of this book, we will learn about **Data Visualization** and other key procedures for a communication and data science specialist. Data visualization is an important tool for communicating the results of data analyses, and students will need to learn how to create clear and effective visualizations to effectively communicate their findings. **Web scraping** is an important tool for **data scientists** because it allows them to **extract data from websites** and turn it into structured, usable data that can be **analyzed** and **visualized**.
- In **Part III: Introduction to business statistics in Data Science**. This part aims to provide a fundamental understanding of statistics and its role in data science and business, as well as equip students with the tools and skills to apply statistical analysis to real-world situations.
- **Part IV** of this book is optional and is intended for readers who wish to further their knowledge in communication and data science. It covers the topic of **Machine learning** and is geared towards those who want to rapidly progress in this field and delve into advanced areas of data science. For the basics of this discipline, I recommend delving into just the first two parts, and optionally part three.

By focusing on these key areas, students can gain a solid foundation in Data Science, which they can then build upon in more advanced coursework and real-world experiences. The four stages of study broadly make up the overall structure of this book. By learning programming, statistics, machine learning, and data visualization, students can gain a solid foundation in Data Science, which they can then build upon in more advanced coursework and real-world experiences. These four areas are essential for success in Data Science because they provide the tools and techniques needed to manipulate, analyze, and interpret data, as well as to communicate the results of those analyses to others.

As the author of this book, I agree that organizing information into theoretical, example and practical components is an effective way to help students assimilate key information quickly and efficiently. By providing a mix of conceptual and practical material, students can

gain a deep understanding of the subject matter and apply this knowledge to real-world problems and scenarios. In addition, by providing examples and exercises, students can see how the concepts they are learning apply in practice, which can help to reinforce their understanding and facilitate the transfer of knowledge to new situations. Overall, this approach can help to engage and motivate learners and help them develop the skills and knowledge they need to succeed as communication and data scientists.

I hereby wish my students and all my readers to use this book to the fullest and learn to love Data Science professionally and then teach others as I have lovingly taught them. The best way to learn about any subject is to approach it with an open mind and a willingness to try new things. To maximize the benefit of this book, I would encourage my students to engage with the material actively, asking questions, trying out the examples and exercises, and seeking out additional resources and opportunities to learn more. Additionally, I encourage them to connect with other Data Science enthusiasts and professionals, either through online communities or in-person events, to learn from each other and stay up to date with the latest developments in the field. By adopting this approach, my students can develop a deep understanding of Data Science and they can become skilled and knowledgeable professionals who are well-equipped to teach others.

AN INTRODUCTION TO DATA SCIENCE FROM THE PERSPECTIVE OF THE COMMUNICATION AND PUBLIC RELATIONS SPECIALIST

Data Science is a rapidly growing field that is revolutionizing many aspects of our lives, from how we shop and consume media, to how we manage our health and make important decisions. At its core, Data Science is about using data to gain insight and make better decisions. This involves collecting and storing large amounts of data, applying statistical and computational techniques to analyze that data, and using the insights generated from those analyses to inform and improve business, policy, and other decision-making processes. A discipline such as this has its roots in several fields, including statistics, computer science, and domain-specific disciplines such as biology or economics. As a result, data scientists come from a wide range of backgrounds and may have expertise in fields such as mathematics, programming, or a specific industry. However, what all data scientists have in common is a passion for working with data, how to communicate, and a desire to use that data to improve the world around them. Data science can also be useful for PR specialists in several ways. **Data analysis:** PR specialists often need to analyze data to understand the public's perception of a company or product, and to identify trends and patterns that can inform their PR strategy. Data science tools and techniques can help them more effectively analyze and interpret data from various sources, such as social media, surveys, and media coverage. **Targeted outreach:** Data science can help PR specialists identify and target specific groups of people or influencers who are likely to be interested in a company or product. By using data to understand the characteristics and behaviors of these target audiences, PR specialists can more effectively craft messaging and outreach efforts. **Personalization:** Data science can also be used to personalize PR efforts, for example by using machine learning to tailor messaging to individual audience members based on their interests

and needs. This can make PR efforts more effective and help build stronger relationships with target audiences. **ROI analysis:** Finally, data science can help PR specialists measure the return on investment (**ROI**) of their PR efforts. By analyzing data such as website traffic, social media engagement, and media coverage, PR specialists can understand the impact of their work and adjust their strategy as needed.

On the other hand, in recent years, huge changes in technology have brought IT much closer to Communication Science, demanding a new approach to the future of the job market. **Communication Science** and **Data Science** are now a common core of success for anyone who wants to secure a job in any corner of the world and any field. The two fields intertwine in a perfect formula, concerned with intelligent analysis of small and big data applicable to a wide variety of fields. The intelligent use of data is aimed at achieving valuable and successful results for any company in the labor market. Tasks in communication and Data Science include gaining a thorough understanding of the problem domain (Business Understanding), processing and merging key data from different sources (Data Gathering), statistical analysis and modeling of data (Data Analytics), as well as communication data via interactive visualization (Visual Analytics) and use of results (Decision Support and Deployment). This book will teach you how to gather and assess datasets, and how to extract results from them. You will also learn how to identify data patterns in various areas, such as stock analysis, data automation, and key data selection, and how to visually represent these patterns in both qualitative and quantitative research in any field of work. With this know-how, you qualify for top positions in the **IT** communications sector.

Typically, **data scientists** require a diverse set of technical skills such as programming, statistics, and machine learning to be effective in their roles. In addition to these technical skills, data scientists should also possess strong problem-solving and communication abilities, as well as the ability to work well in a team and manage complex projects. With the right skills and experience, data scientists can find exciting and rewarding career opportunities in a wide range of industries, from technology and finance to healthcare and government. In conclusion, Data Science is a fascinating and rapidly growing field that offers many exciting career opportunities for individuals with the

right skills and experience. Whether you are a beginner just starting to explore Data Science, or an experienced professional looking to deepen your expertise, there is a wealth of resources and opportunities available to help you succeed in this exciting field.

It is important for future communication and Data Science specialists to think differently because this field involves approaching problems and making decisions in a novel and innovative manner. This is a crucial old dilemma that we need to clarify. Why do we need to think differently from the typical way of perceiving things, especially when it comes to decisions? Data Science requires a different perspective on problem-solving and decision-making. Unlike traditional methods of analysis and decision-making, which often rely on intuition and subjective experiences, Data Science uses objective, quantifiable data to inform decisions. This approach allows for more unbiased and accurate analyses and can help to identify new insights and opportunities that may not be immediately apparent using more traditional methods. Additionally, thinking differently is important in Data Science because it involves considering a wider range of factors and perspectives. By incorporating a diverse range of data sources and viewpoints, data scientists can generate more comprehensive and accurate analyses and can identify solutions that may not have been considered using more narrow approaches. In conclusion, thinking differently is essential for future communication and Data Science specialists because it allows them to approach problems and decision-making more objectively and comprehensively, which can lead to better outcomes and more accurate insights. By adopting a data-driven approach, communication and Data Science specialists can help organizations to make more informed and effective decisions and can use their expertise to drive positive change in the world.

Another thing we need to clarify is to eliminate the fear that mathematics or mathematical thinking is necessarily a prerequisite for being a good future communication and Data Science specialist. Yes, it is not necessary to have a strong background in mathematics. While Data Science does involve some mathematical concepts and techniques, these are often implemented using specialized software and tools, which can make them accessible to individuals with a wide range of backgrounds and expertise. Additionally, while it is true that

some Data Science roles may require a strong foundation in mathematics, many other Data Science roles do not. For example, many organizations require data-savvy communication and public relations specialists who can help to interpret and communicate the results of data analyses to a broader audience. In these roles, strong communication and storytelling skills are often more important than advanced mathematical knowledge. While a basic background in mathematics can be helpful for some Data Science roles, it is not a prerequisite for success in this field. Individuals with diverse backgrounds can thrive as communication and Data Science specialists by prioritizing the development of strong problem-solving and communication abilities, as well as a foundational understanding of Data Science concepts and techniques.

Furthermore, here are some other good tips that could help you assimilate and learn knowledge faster:

1. **Start by gaining a solid foundation in the basics.** To become a good Data Science specialist, it is important to start by gaining a strong foundation in the core concepts and skills that are essential for success in this field. This may include topics such as programming, statistics, and machine learning, as well as more specialized areas such as natural language processing or computer vision. To build this foundation, consider taking online courses, attending workshops, or reading books and other resources to learn the basics.
2. **Get hands-on experience.** In addition to learning the theoretical concepts of Data Science, it is also important to gain practical experience by working on real-world projects and datasets. This can help you to develop the skills and knowledge that are needed to apply Data Science in a professional setting and can also provide valuable portfolio pieces to showcase your abilities to potential employers. Consider participating in online competitions, working on side projects, or interning at a company to gain hands-on experience.
3. **Connect with others in the field.** Data Science is a rapidly growing and evolving field, and it is important to stay up to date with the latest developments and trends. One way to do this is to connect with other Data Science professionals and enthusiasts, either through online communities or in-person events. By networking with others in

the field, you can learn from their experiences, share your insights, and stay informed about the latest trends and opportunities in Data Science.

4. **Keep learning and growing.** Finally, remember that becoming a good Data Science specialist is an ongoing process and that there is always more to learn and discover in this field. To continue to grow and improve as a data scientist, it is important to be open to new ideas and perspectives and to be willing to try new things. This may involve attending conferences, taking additional courses, or pursuing advanced certifications to deepen your expertise. By adopting a growth mindset and a commitment to lifelong learning, you can continue to develop and improve as a Data Science specialist.

When learning to code in different programming languages, several strategies can help you to remember important codes and avoid forgetting them. Some of these strategies include:

- **Practice regularly:** One of the best ways to remember something is to use it frequently. By regularly practicing and using the codes you are learning, you can help to reinforce them in your memory and make them more familiar and easier to recall.
- **Use mnemonic devices:** Mnemonic devices, such as acronyms or rhymes, can help remember complex codes or sequences of information. By creating a simple, memorable phrase or sentence that incorporates the codes you are trying to remember, you can make them easier to recall.
- **Break the codes down into smaller chunks:** If a code or sequence of information is particularly long or complex, it can be helpful to break it down into smaller chunks or sub-sequences, and to focus on memorizing each chunk individually. This can make the information more manageable and easier to remember.
- **Create flashcards:** Flashcards are a simple, but effective, tool for memorizing information. By creating flashcards that include the codes you are trying to remember, along with brief explanations or examples, you can test yourself and reinforce your knowledge of the codes.
- **Use visualization:** Visualization techniques, such as creating mind maps or diagrams, help remember complex codes or sequences of information. By creating a visual representation of the informa-

tion, you are trying to remember, you can make it more concrete and easier to recall.

The key to remembering important codes when learning to code is to use a variety of different techniques and strategies and to practice regularly to reinforce your knowledge and improve your memory. By adopting these strategies, you can improve your ability to remember important codes and avoid forgetting them.

PART I

Programming

Programming is essential for manipulating and analyzing data. One of the key tasks of a data scientist is to work with large and complex datasets and extract valuable insights from those datasets. This involves using a variety of tools and techniques to manipulate, clean, and transform the data, as well as to apply statistical and machine learning algorithms to analyze the data. To do this effectively, data scientists need to be proficient in at least one programming language, such as **Python** or **R**, which are commonly used in Data Science. Programming skills are in high demand in the job market. In today's job market, there is a high demand for individuals with expertise in Data Science and having strong programming skills can make you a more competitive candidate for these roles. Many employers are looking for data scientists who have a strong foundation in programming, and who can use those skills to work with large and complex datasets. By learning basic programming skills, you can improve your chances of landing a job in Data Science and can increase your earning potential in the field. Also, programming skills can help you to stand out from your peers. In addition to being essential for working in Data Science, programming skills can also be valuable in other fields, such as finance, healthcare, or government. By learning basic programming skills, you can differentiate yourself from other candidates and can open a wider range of career opportunities. Additionally, programming skills can be useful for personal projects and side hustles and can help you to develop

Python fundamentals in the introduction to Data Science

Python is a widely used programming language in Data Science. One of the reasons to start learning Python to get into Data Science is that it is currently the most widely used language in the field. Many data scientists use Python as their primary programming language, and a large and growing number of Data Science libraries and frameworks are written in Python. This means that learning Python can provide you with valuable skills that are in high demand in the job market and can help you to work effectively with the tools and technologies that are commonly used in Data Science. Another reason why learning Python is important for Data Science is that Python has a large and active community of users and developers. This means that there is a wealth of resources available for learning Python, including books, online courses, and tutorials, as well as forums and communities where you can ask questions and get help from others. Additionally, the large and active community means that new libraries, frameworks, and tools are constantly being developed and released, which can help to keep your skills up to date and relevant. Python is not only a popular language in the realm of Data Science, but it is also a versatile, general-purpose programming language that can be utilized for various purposes such as web development, scientific computing, and automation. This versatility means that gaining proficiency in Python can equip you with a valuable skill that can be utilized in numerous settings and can enhance your flexibility and adaptability in your professional journey.

Python has a simple and intuitive syntax. Another reason why learning Python is important for Data Science is that Python has a simple and intuitive syntax, which makes it easy to learn and use. Unlike many other programming languages, which can have steep learning curves and a complex syntax. Its syntax is designed to be readable and easy to understand, which means that you can quickly start writing code and experimenting with different ideas. In addition to its simplicity, Python also has a wide range of powerful libraries and frameworks that can be used for Data Science and machine learning. This means that you can easily perform complex data analysis and build so-

phisticated machine-learning models using Python. Overall, Python's simple syntax and powerful Data Science tools make it an ideal language for anyone interested in pursuing a career in Data Science.

[hints!] Before proceeding, we must address the most challenging question: how can we conquer the fear of learning to program in Python for data science if we have a degree in communication or a field unrelated to the natural sciences? Learning Python programming for data science can seem intimidating, especially if you have a non-technical or non-science background. However, with the right mindset and strategies, you can conquer your fears and become proficient in Python and data science. Here are a few ideas to keep in mind:

- **Start small:** Don't try to tackle everything at once. Start with basic concepts and gradually build up your knowledge.
- **Continuous practice:** The more you practice coding, the more comfortable you will become. Consider working through online tutorials or exercises or building small projects on your own.
- **Get support:** Consider joining a study group or finding a mentor who can provide guidance and support as you learn.
- **Don't be afraid to ask for help:** If you get stuck or don't understand something, don't be afraid to ask for help. There are many online communities and forums where you can ask questions and get support from experienced programmers.
- **Take breaks:** Learning to code can be mentally exhausting, so be sure to take breaks and give yourself time to rest and recharge.

Learning to program in Python and specializing in data science is a challenging but rewarding process. With dedication, persistence, and the right resources, you can overcome your fear and become proficient in these valuable skills.

Let's get started with Python basics!

To get started with Python, you will need to have a **Python interpreter** installed on your computer. The **Python interpreter** is a program that allows you to run Python code on your computer. There are

many different versions of the Python interpreter available, but the most used one is called **CPython**. **CPython** is the reference implementation of the Python language, and it is developed and maintained by the **Python Software Foundation**.

You can download the latest version of **CPython** from the official Python website at <https://www.python.org/>. Just follow the instructions on the website to download and install the Python interpreter on your computer. Once you have installed the **Python interpreter**, you can start writing and running Python code. There are a few different ways to do this, depending on your preferences and needs. There are a few options available for writing and running Python code. One option is to use a text editor and then run the code using the Python interpreter, which can be useful for writing and executing small programs or scripts, or for testing out different concepts. One possibility is to use an integrated development environment (**IDE**) to write and execute Python code. An **IDE** is a specialized program that provides a more advanced environment for writing and running code. It typically includes features such as code completion, debugging tools, and support for working with large projects.

Once you have installed the **Python interpreter** and chosen a development environment, you can start learning the Python language. Getting started with Python is relatively easy, and there are many resources available to help you learn the language and start writing your programs.

How to launch an interactive Python using IDLE

To launch the interactive **Python interpreter** using **IDLE**, follow these steps:

1. Open the Start menu and search for “**IDLE**”
2. Click on the **IDLE** icon to launch the program
3. In the **IDLE** window that opens, click on the “**Python shell**” option

This will open the interactive **Python interpreter**, which you can use to write and run Python code. You can type Python commands directly into the interpreter, and they will be executed as soon as you press the **Enter key**.

For instance, you can utilize the **print()** function to show output on the screen:

```
print("Hello, World!")  
Hello, World!
```

It is also possible to use the interpreter to carry out basic calculations and operations:

```
12 + 22  
34
```

```
"Hello" + "World"  
"HelloWorld"
```

The interactive **Python interpreter** is a great way to experiment with Python code and try out different ideas. You can use it to test your understanding of the language and learn how to write Python programs. Launching the interactive **Python interpreter** using **IDLE** is a simple process, and it can be a valuable tool for learning and exploring the Python language.

Other online shells for Python

In addition to **IDLE**, numerous other online tools and platforms allow you to run Python code. Some of the most well-known options include:

1. **PythonAnywhere**: This is a cloud-based platform that allows you to write and run Python code from any web browser. It includes a range of features and tools for Python development, including a web-based editor and a range of libraries and frameworks.
2. **repl.it**: This is an online platform that provides a range of programming languages, including Python. It includes a web-based editor and an interactive shell, as well as tools for collaboration and sharing.
3. **Jupyter Notebook**: This is an open-source web-based platform that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. It is frequently used for Data Science and machine learning and supports various programming languages, including Python.

These are just a few examples of the online tools and platforms available for running Python code. There are many others available, and you can choose the one that best fits your needs and preferences. Using an online tool or platform to run Python code can be a convenient and efficient way to write and test your Python programs.

[hints!] Ultimately, the best online tool or platform for a beginner in data science will depend on their specific needs and goals. IDLE and repl.it may be good starting points for those who are new to programming and want a simple and easy-to-use tool, while PythonAnywhere and Jupyter Notebook may be better options for those who are more advanced or looking for more advanced features and tools.

Introduction to Data Types

Python uses **data types** to categorize different types of data and determine how they can be stored and manipulated (VanderPlas, J. 2016). A few examples of the standard data types in Python are:

1. **Numbers:** In Python, there are two main categories of numbers: **integers** (whole numbers) and **floating-point** numbers (numbers with decimal points). For example, **10** is an **integer**, while **3.14** is a **floating-point** number.
2. **Strings:** A string is a **sequence of characters** that represent **text**. In Python, **strings** are enclosed in quotation marks (either single or double). For example, "Hello, World!" is a **string**.
3. **Booleans:** A Boolean value is a data type that represents a logical state with either a **True** or **False** value. They are frequently used in conditional statements to verify whether a specific condition has been satisfied.
4. **Lists:** A list is a group of items that can be of various **data types**. In Python, lists are written in square brackets (**[]**) and the items within a list are separated by commas. For instance, **[1, 2, 3]** represents a list of numbers, and **["red", "green", "blue"]** represents a list of strings.
5. **Tuples:** A tuple is similar to a list, but it is immutable (meaning that it cannot be changed once it has been created). Python uses parentheses (**()**) to denote **tuples**, with items within a tuple separated by commas. For example, the **tuple** **(1, 2, 3)** represents a group of numbers.

6. **Dictionaries:** In Python, a **dictionary** is a **data type** that contains a collection of **key-value pairs**. Dictionaries are represented with curly braces (`{}`) and keys and values are separated by colons (`:`). For example, `{"name": "Ion", "age": 34}` is a **dictionary** that has two keys (`"name"` and `"age"`) and two values (`"Ion"` and `34`).

As you continue learning Python, you'll discover more data types and how they can be used. It is important to understand "data types in Python because they play a central role in how various types of data can be stored, accessed, and manipulated" (Harris, C. R. & all, 2020: 357).

[**hints!**] To develop a strong understanding of data types in Python, it is important to practice regularly and seek out resources and help when needed. With dedication and determination, you can become proficient in these essential concepts.

Let's learn how to code in Python

Here is a straightforward example of how to create a Python program:

```
# This is a comment in Python
# Comments are used to add notes or explanations
to your code

# The print() function is used to display output
on the screen
print("Hello, World!")
```

To run this program, you can **save** it in a file with a **.py** extension and run it using the **Python interpreter**.

Here is a description of the code:

- The **#** symbol is used to indicate a comment in Python. Anything that follows a **#** on a line is **ignored by the Python interpreter**.
- The **print()** function is used **to display output on the screen**. In this case, we are using it **to print the string** `"Hello, World!"`.
- The `"` and `'` characters are used to define string values in Python. A string is a sequence of characters that represent text.

This is just a very simple example, but it shows some of the basic concepts of Python.

[hints!] One tip for fast coding in Python is to use a code editor or Integrated Development Environment (IDE) that provides features such as code completion and auto-indentation. These features can help you write code more quickly by automatically inserting commonly used code snippets and formatting your code correctly. Some popular IDEs for Python include PyCharm, Eclipse with the PyDev plugin, and Visual Studio Code. Another tip is to use libraries and frameworks that provide pre-built functions and features that you can use in your code. This can save you time and effort by allowing you to focus on the core logic of your program rather than having to write low-level code from scratch. Finally, it can be helpful to develop a consistent coding style and adhere to best practices when writing Python code. This can make your code easier to read and understand, and it can help you write code more efficiently by following established conventions.

Integers and Floats

In Python, there are two basic types of numbers: **integers** and **floating-point** numbers. An **integer** is a numerical value that does not contain a decimal point. Examples of integers include **10**, **-5**, and **0**. In Python, you can define an **integer** by using the `int` keyword or by writing a whole number without any decimal point. For instance:

```
# Defining an integer using the int keyword
x = int(10)

# Defining an integer by writing a whole number
y = 5
```

An **integer** is a numerical value that does not include a decimal point. Some examples of integers are **10**, **-5**, and **0**. On the other hand, “a **floating-point** number is a number with a decimal, such as **3.14**, **-0.1**, and **0.0**. In Python, you can create a floating-point number by writing a number with a decimal point” (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620: 16). As an example:

```
# Defining a floating-point number
x = 3.14
y = -0.1
```

In Python, you can perform various operations on **integers** and **floating-point** numbers, such as **addition**, **subtraction**, **multiplication**, and **division**. For example:

```
# Dividing two integers
a = 10 / 3 # 3.3333333333333335

# Adding two floating-point numbers
x = 3.14 + 5.25 # 8.39

# Subtracting two floating-point numbers
y = 10.5 - 5.3 # 5.2

# Multiplying two floating-point numbers
z = 5.5 * 10.1 # 55.55

# Dividing two floating-point numbers
a = 10.5 / 3.2 # 3.28125
```

As you can see, you can use **integers** and **floating-point** numbers in various ways in your Python programs, and you can perform different operations on them to manipulate and analyze your data.

[hints!] One tip to quickly learn about integers and floating-point numbers (also known as “floats”) in Python is to practice working with them in code. This could involve writing code to manipulate integers and floats or working through exercises that involve these data types. To understand the difference between integers and floats, it can be helpful to consider the following:

```
x = 1 # x is an integer
y = 1.0 # y is a float

z = float(x) # z is now a float with the value 1.0
w = int(y) # w is now an integer with the value 1
```

Integers are whole numbers that do not have a decimal point, such as 1, 2, 3, etc. They are often used to represent counts or quantities that cannot be fractional. Floats refer to numbers that contain a decimal point, such as 1.0, 2.5, 3.14, and so on. They are used to represent decimal values or numbers that are not necessarily whole. In Python, you can use the `int` and `float` functions to convert between integers and floats, respectively. By practicing and testing with integers and floats in Python, you can improve your understanding of how these data types operate and how to effectively utilize them in your code.

Basic operators

Operators are special symbols used to perform operations on values and variables. Some of the most frequently used operators in Python include:

- **Arithmetic operators:** These operators are used to perform basic arithmetic operations, such as **addition**, **subtraction**, **multiplication**, and **division**. For example:

```
# Addition
x = 5 + 22 # 27

# Subtraction
y = 10 - 4 # 6

# Multiplication
z = 4 * 8 # 32

# Division
a = 100 / 10 # 10
```


- **Assignment operators:** These operators are used to assigning values to variables. For example:

```
# Assignment operator
x = 5

# Assignment operator with addition
x += 2 # x is now equal to 7

# Assignment operator with subtraction
y -= 3 # y is now equal to 7

# Assignment operator with multiplication
z *= 4 # z is now equal to 20

# Assignment operator with division
a /= 2 # a is now equal to 5
```

- **Comparison operators:** These operators are used to compare two values and determine their relationship. For example:

```
# Equal to
x = 5
y = 2
x == y # False

# Not equal to
x != y # True

# Greater than
x > y # True

# Less than
x < y # False

# Greater than or equal to
x >= y # True

# Less than or equal to
x <= y # False
```

These are just a few examples of operators in Python. There are many others available, and you can learn more about them as you continue learning the language. **Operators** are an important part of Python, and they are used to perform various operations on values and variables. By using the right operators, you can write concise and efficient code that can manipulate and analyze your data.

[hints!] One quick tip for learning basic operators in Python is to practice using them in code. This could involve writing simple programs that use operators to perform tasks such as arithmetic calculations, assignments, and comparisons. Here are some examples of basic operators in Python:

Arithmetic operators: +, -, *, /, % (modulus), ** (exponentiation)

Assignment operators: =, +=, -=, *=, /=, %=

Comparison operators: == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to)

For example, to use the + operator to add two numbers together, you can write code like this:

```
x = 1
y = 2
z = x + y  # z is now 3
```

Operators can also be used to perform more complex tasks, such as assigning the outcome of an arithmetic expression to a variable (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620):

```
x = 1
y = 2
z = x + y * 3  # z is now 7
```

By practicing and experimenting with different operators in Python, you can gain a better understanding of how they work and how to use them effectively in your code.

Lesser-Known operators

In addition to the basic operators mentioned above, Python also includes several **lesser-known operators** that you can use in your programs. Some examples include:

- **Bitwise operators:** These operators are used to execute bitwise operations on integer values. For instance, the `'&'` operator performs a bitwise **AND** operation, and the `'|'` operator performs a bitwise **OR** operation.
- **Logical operators:** These operators are used to combine and manipulate **Boolean** values. For example, the `'and'` operator returns `'True'` if both operands are `'True'`, and the `'or'` operator returns `'True'` if at least one of the operands is `'True'`.
- **Identity operators:** These operators are used to compare the identities of two objects. For instance, the `'is'` operator returns `'True'` if the two operands refer to the same object, and the `'is not'` operator returns `'True'` if the two operands do not refer to the same object (Lubanovic, B., 2014).
- **Membership operators:** These operators are used to determine whether a value “is a member of a sequence (such as a string, list, or tuple). For instance, the `'in'` operator returns `'True'` if the value is a member of the sequence, and the `'not in'` operator returns `'True'` if the value is not a member of the sequence” (Lubanovic, B., 2014: 30).

These are just a few examples of **lesser-known operators** in Python. There are many others available, and you can learn more about them as you continue learning the language. These operators can be useful in certain situations, but they are not used as frequently as the basic operators discussed above. However, it is still important to be aware of them, as they can come in handy when working with more complex data and algorithms.

Variables

In Python, a **variable** is a named location in memory where a value can be stored and accessed. **Variables** are used to store data and information in your programs, and they allow you to manipulate and analyze

that data in various ways. To create a variable in Python, you simply need to give it a name and assign a value to it using the assignment operator (=). For example:

```
# Creating a variable and assigning a value to it  
x = 5  
  
# The variable x now holds the value 5  
print(x) # Output: 5
```

Once you have created a variable, you can use it in your program in various ways. For example, you can use it in expressions and calculations, or you can use it to control the flow of your program (Lubanovic, B., 2014).

```
# Using a variable in an expression  
x = 5  
y = 2  
z = x + y # z is equal to 7  
  
# Using a variable to control the flow of a program  
x = 5  
if x > 10:  
    print("x is greater than 10")  
else:  
    print("x is not greater than 10") # Output:  
"x is not greater than 10"
```

In Python, you can use any combination of letters, numbers, and underscores (_) to name your variables, but the name must start with a letter or underscore. It is also important to choose descriptive and meaningful names for your variables, as this will make your code easier to understand and maintain. **Variables** are an essential part of Python, and they play a crucial role in storing and manipulating data in your programs.

[hints!] One tip for learning variables in Python more easily is to practice using them in code. This could involve writing simple programs that use variables to store and manipulate data or working through exercises that involve variables. To create

a variable in Python, you simply need to give it a name and assign it a value using the = operator. For example:

```
x = 10 # x is now a variable with the value 10
y = "hello" # y is now a variable with the value
"hello"
```

You can then use variables in your code just like you would use their values directly. For example:

```
x = 10
y = 20
z = x + y # z is now 30
```

To learn more about variables in Python, you can also refer to online resources such as tutorials and documentation. These resources can provide additional information and examples to help you understand how variables work and how to use them effectively in your code.

Assignment operators

In Python, assignment operators are used to assign values to variables (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620). The basic assignment operator is the equal sign (=), which is used to assign a value to a variable. For example:

```
# Assigning a value to a variable using the
assignment operator
x = 5
In this example, the variable x is assigned the
value 5.
```

In addition to the basic assignment operator, Python also includes several compound assignment operators, which are used to perform an operation and assign the result to a variable in one step. For example:

The += operator is used to add a value to a variable and assign the result to the variable. For example:

```
# Adding a value to a variable using the += opera-
tor
x = 5
x += 2 # x is now equal to 7
```

The `-=` operator is used to subtract a value from a variable and assign the result to the variable. For example:

```
# Subtracting a value from a variable using the -=  
operator  
x = 5  
x -= 2 # x is now equal to 3
```

The `*=` operator is used to multiply a value by a variable and assign the result to the variable. For example:

```
# Multiplying a value by a variable using the *=  
operator  
x = 5  
x *= 2 # x is now equal to 10
```

The `/=` operator is used to divide a value by a variable and assign the result to the variable. For example:

```
# Dividing a value by a variable using the /=  
operator  
x = 5  
x /= 2 # x is now equal to 2.5
```

These are just a few examples of compound assignment operators in Python. There are many others available, and you can learn more about them as you continue learning the language. Assignment operators are an important part of Python, and they are used to assign values to variables.

Numbers & variables in the wild

In Python, **numbers** and variables can be used in a wide range of applications and contexts. For example, you can use them to perform calculations, analyze data, and create algorithms. One common use of numbers and variables in Python is in scientific and mathematical applications (Meurer, A. & all, 2017). For example, you can use them to perform complex calculations and simulations, such as analyzing the motion of objects in space or modeling the spread of disease.

```
# Calculating the distance between two points in
2D space
x1 = 5
y1 = 2
x2 = 10
y2 = 7
distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
print(distance) # Output: 7.0710678118654755
```

Another common use of numbers and variables in Python is in **data analysis** and **machine learning**. For example, you can use them to manipulate and analyze large datasets and to create and train predictive models.

```
# Analyzing a dataset of customer data
customer_data = [
    {"name": "Ion", "age": 34, "income": 56000},
    {"name": "Jane", "age": 28, "income": 48000},
    {"name": "Bob", "age": 40, "income": 63000},
    {"name": "Alice", "age": 29, "income": 50000},
]

# Calculating the average age and income of the
customers
total_age = 0
total_income = 0
for customer in customer_data:
    total_age += customer["age"]
    total_income += customer["income"]
average_age = total_age / len(customer_data)
average_income = total_income / len(customer_data)
print("Average age:", average_age) # Output: 33.25
print("Average income:", average_income) # Output:
53800.0
```

These are just a few examples of the ways you can use numbers and variables in Python. There are numerous other possibilities, and you can apply them to solve a wide range of problems and challenges across different fields and industries.

★ *Magic trick exercise*

To perform a magic trick using Python, you could write a program that prompts the user to think of a number and then uses a series of calculations and manipulations to “**guess**” the number. Here is an example of how this could be done:

```
# Prompt the user to think of a number
print("Think of a number between 1 and 10")

# Ask the user to double the number
print("Now, double the number you are thinking of")

# Ask the user to add 5 to the number
print("Now, add 5 to the number")

# Ask the user to divide the number by 2
print("Now, divide the number by 2")

# Ask the user to subtract the original number
from the result
print("Now, subtract the original number you were
thinking of from the result")

# Print the result (which should be the number 2)
print("The number you are thinking of is 2!")
```

When the user runs this program, they will be prompted to think of a number and perform a series of calculations on it. At the end of the program, the result of the calculations will be the number **2**, which will appear to be a “**guess**” of the original number the user was thinking of. Of course, this is just one example of how you could use Python to perform a magic trick. There are many other possibilities, and you can use your creativity and imagination to come up with your tricks and illusions.

★ *Exercises for basic variables in Python*

Ex.1. Create a Python program that prompts the user to enter their name, age, and favorite color. The program should then store these values in separate variables and print them out in the following format:

```
Hi! My name is [name] and I am [age] years old. My
favorite color is [color].
```

Ex.2. Write a Python program that declares two variables, **x** and **y**, and assigns them the values of **5** and **10**, respectively. The program should then perform the following operations with these variables:

- Calculate the **sum** of **x** and **y** and assign the result to a new variable called **z**
- Calculate the **difference** between **y** and **x** and assign the result to a new variable called **a**
- Calculate the **product** of **x** and **y** and assign the result to a new variable called **b**
- Calculate the **quotient** of **y** and **x** and assign the result to a new variable called **c**

After performing these operations, the program should print out the values of the variables **z**, **a**, **b**, and **c**.

Ex.3. “Write a Python program that creates a list called **numbers** that contains the values **1**, **2**, **3**, **4**, and **5**. The program should then perform a Python program that creates a list called **numbers** that contains the values **1**, **2**, **3**, **4**, and **5**. The program should then perform the following operations on the list:

- Use the **sum()** function to calculate the sum of all the numbers in the list and store the result in a variable called **total**
- Use the **min()** function to calculate the minimum value in the list and store the result in a variable called **min_value**
- Use the **max()** function to calculate the maximum value in the list and store the result in a variable called **max_value**
- Use the **len()** function to calculate the length of the list and store the result in a variable called **list_length**” (Gad, A. F., 2021).

After performing these operations, the program should **print** the values of the variables `total`, `min_value`, `max_value`, and `list_length`. Here is an example of how the program could be implemented:

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]

# Calculate the sum of the numbers in the list
total = sum(numbers)

# Calculate the minimum value in the list
min_value = min(numbers)

# Calculate the maximum value in the list
max_value = max(numbers)

# Calculate the length of the list
list_length = len(numbers)

# Print the results
print("Total:", total)
print("Minimum value:", min_value)
print("Maximum value:", max_value)
print("List length:", list_length)
```

This program creates a list of numbers and then uses various **built-in functions** to calculate and print out some summary statistics about the list. Using variables and built-in functions in Python allows you to easily perform operations and analyze your data.

If you've made it this far: Congratulations! You're already a good Python programmer!

Strings Basic

In Python, a **string** is a series of characters enclosed in quotation marks. You can create a string by enclosing characters in either single quotes ('') or double quotes (""). Some examples of strings in Python are:

```
my_string = 'Hello World!'
my_string = "Hello World!"
```

You can access individual characters in a string using the indexing operator []. The index of the first character in a string is always **0**. For example, you can access the first character in the string above using the following code:

```
first_char = my_string[0]
```

You can also use **negative indexing** to access characters in a **string**. For example, the code below will access the last character in the string:

```
last_char = my_string[-1]
```

In Python, **strings are immutable**, which means that you cannot modify a string once it has been created. However, you can create new strings by concatenating (**joining**) two or more strings together. Here's an example:

```
my_string = 'Hello'
new_string = my_string + ' World!'
```

The new string would be “Hello World!”

String Operators

Python offers a range of **operators** that can be used with **strings**, such as the **concatenation operator** (+), which merges two strings into a new string, and the **repetition operator** (*), which creates a new string by **repeating** another string a specified number of times.

Here are some examples of using these operators with strings in Python:

```
# Concatenation operator
my_string = 'Hello'
new_string = my_string + ' World!'

# Output: "Hello World!"
print(new_string)

# Repetition operator
my_string = 'Hello '
new_string = my_string * 3

# Output: "Hello Hello Hello "
print(new_string)
```

Additionally, several built-in methods in Python can be used to manipulate strings, such as the **upper()** method, which converts a string to uppercase, and the **lower()** method, which converts a string to lowercase. Here are some examples of using these methods:

```
my_string = 'Hello World!'

# Convert to uppercase
upper_string = my_string.upper()

# Output: "HELLO WORLD!"
print(upper_string)

# Convert to lowercase
lower_string = my_string.lower()

# Output: "hello world!"
print(lower_string)
```

[hints!] One tip for quickly learning to work with strings in Python is to practice using them in code. This could involve writing simple programs that use strings to store and manipulate data or working through exercises that involve strings. To create a string in Python, you can use either single or double quotes. For example:

```
x = "hello" # x is now a string with the value
             "hello"
y = 'world' # y is now a string with the value
             "world"
```

You can use various methods and operators to work with strings in Python. For example, you can use the **+** operator to concatenate (combine) two strings:

```
x = "hello"
y = "world"
z = x + y # z is now "helloworld"
```

You can also use the `len()` function to get the length of a string:

```
x = "hello"
length = len(x)  # length is now 5
```

To learn more about strings in Python, you can also refer to online resources such as tutorials and documentation. These resources can provide additional information and examples to help you understand how strings work and how to use them effectively in your code.

String Indexing

Strings are sequences of characters, and you can access individual characters in a string using the indexing operator `[]`. The index of the first character in a string is always **0**. For example, you can access the first character in the string `my_string` using the following code:

```
my_string = 'Hello World!'
first_char = my_string[0]

# Output: "H"
print(first_char)
```

You can also use **negative indexing** to access characters in a string. For example, the code below will access the last character in the string:

```
my_string = 'Hello World!'
last_char = my_string[-1]

# Output: "!"
print(last_char)
```

You can access a range of characters in a string using the slicing operator `(:)`. This operator allows you to specify the start and end indices of the characters you want to access, separated by a colon `(:)`. For example, the code below will access the characters from the 5th index to the 10th index (not including the 10th index) in the string `my_string`:

```
my_string = 'Hello World!'
substring = my_string[5:10]

# Output: "World"
print(substring)
```

String Slices

You can use the **slicing** operator (`:`) to access a range of characters in a string. This operator allows you to specify the start and end indices of the characters you want to access, separated by a colon (`:`). For instance, the code below will retrieve the characters from the 5th position to the 10th position (not including the 10th index) in the string `my_string`:

```
my_string = 'Hello World!'
substring = my_string[5:10]

# Output: "World"
print(substring)
```

You can also use the **slicing** operator to access characters from the beginning or end of a string by leaving the start or end index empty. For example, the code below will access all characters from the 5th index to the end of the string:

```
my_string = 'Hello World!'
substring = my_string[5:]

# Output: "World!"
print(substring)
```

The code below will “retrieve all characters from the beginning of the string up to the 10th position (not including the 10th index)” (Sheppard, C., 2017: 8):

```
my_string = 'Hello World!'
substring = my_string[:10]
```

Output: “Hello Worl”

```
print(substring)
```

Print() function

The **print()** function in Python is used to display output on the screen. It can be used to print text, strings, numbers, or any other type of data. Here are some examples of using the **print()** function in Python:

```
# Print a string
print('Hello World!')

# Print a number
print(42)

# Print a boolean value
print(True)

# Print multiple values
print('Hello', 'World!', 42, True)
```

By default, the **print()** function separates each value with a space and prints each value on a new line. You can change this behavior by using the **sep** and **end** keyword arguments. The **sep** argument allows you to specify a different separator between values, and the **end** argument allows you to specify a different string to be printed at the end of the output (Sheppard, C., 2017). Here's an example:

```
# Print multiple values with a custom separator
and end string
print('Hello', 'World!', 42, True, sep=', ',
end='\n\n')
```

This would print the following output:

```
Hello, World!, 42, True
```

Escape characters

Escape characters are characters that are preceded by a backslash (`\`) and have a special meaning. These characters are used to represent certain special characters within a string.

Here are some examples of **escape** characters in Python:

`\n` - Represents a newline character

`\t` - Represents a tab character

`\\` - Represents a backslash character

`\"` - Represents a double quote character

`\'` - Represents a single quote character

Here's an example of using **escape** characters in a **string**:

```
my_string = 'Hello\nWorld!\tHow are you today?\\'

# Output:
# Hello
# World!      How are you today?\
print(my_string)
```

In the example above, the `\n` character is used to insert a new line in the string, the `\t` character is used to insert a tab, and the `\\` character is used to insert a backslash.

[hints!] One tip for memorizing escape characters in Python is to practice using them in code. This could involve writing simple programs that use “escape” characters to include special characters in strings or working through exercises that involve “escape” characters. Escape characters in Python are used to represent special characters within strings that may have a special meaning in Python, such as newline, tab, or quotation marks. To use an escape character, you need to prefix it with a backslash (`\`).

Here are some examples of escape characters in Python:

`\n`: Newline

`\t`: Tab

`\\`: Backslash

`\'`: Single quote

`\"`: Double quote

For example, to create a string that includes a newline, you can use the `\n` escape character:

```
x = "Hello,\nworld!"  
print(x)  # prints "Hello,\nworld!"
```

To learn more about escape characters in Python, you can also refer to online resources such as tutorials and documentation. These resources can provide additional information and examples to help you understand how escape characters work and how to use them effectively in your code.

Triple quotes

Triple quotes (`'''` or `"""`) are used to create a string that spans multiple lines. This is useful when you want to create a string that contains a lot of text or includes line breaks (Campesato, O., 2022).

Here's an example of using triple quotes to create a multi-line string:

```
my_string = '''Hello World!  
This is a multi-line string.  
You can use triple quotes to create strings that  
span multiple lines.'''  
  
# Output:  
# Hello World!  
# This is a multi-line string.  
# You can use triple quotes to create strings that  
span multiple lines.  
print(my_string)
```

Note that when using triple quotes, you can use single quotes (`'''`) or triple double quotes (`"""`) to create the string. You just need to make sure that the opening and closing quotes match.

Additionally, you can use escape characters within a triple-quoted string, just like you would with a regular string. For example:

```
my_string = '''Hello\nWorld!\tThis is a multi-line
string with escape characters.'''

# Output:
# Hello
# World!      This is a multi-line string with es-
# cape characters.
print(my_string)
```

[**hints!**] Triple quotes (“” or “”) “are used to create multi-line strings in Python. These can be useful in a variety of contexts, including in data science projects. Some possible uses for triple quotes in data science include” (Campesato, O., 2022: 15):

- Creating long strings: Triple quotes can be used to create strings that span multiple lines, which can be useful for creating long blocks of text, such as descriptions or annotations (Campesato, O., 2022).
- Formatting string output: You can use triple quotes to create a string that includes multiple lines and line breaks, which can make it easier to format the output of your code.
- Creating templates: You can use triple quotes to create templates that include placeholders for variables. This can be useful for creating report templates or other documents that require specific formatting. (bin Uzayr, S., 2021)
- Commenting: In some cases, you may want to include long comments in your code that span multiple lines. Triple quotes can be used to create multi-line comments that are easier to read and understand.

While triple quotes are not strictly necessary for all data science projects, they can be a useful tool to have in your toolkit, particularly when working with long strings or when formatting the output of your code.

More about strings

A string is a sequence of characters enclosed in quotation marks. You can use either single or double quotes to create a string, if you start and end the string with the same type of quotation mark (bin Uzayr, S., 2021). For example, the following are all valid strings in Python:

```
# Using single quotes
string1 = 'This is a string'

# Using double quotes
string2 = "This is also a string"

# Using triple quotes
string3 = """This is a multi-line string. You can
use it to create strings that span multiple lines.
Just make sure to start and end the string with
triple quotes."""
```

In Python, you can use the + operator to concatenate (**join**) two or more strings together, like this:

```
# Concatenating two strings
string1 = 'Hello'
string2 = 'world'
print(string1 + ' ' + string2)

# Output: Hello world
```

You can also use the * operator to repeat a string multiple time, like this:

```
# Repeating a string
string1 = 'Hello '
print(string1 * 5)

# Output: Hello Hello Hello Hello Hello
```

In Python, strings are immutable, which means that you can't change an existing string. However, you can create a new string by combining (concatenating) multiple strings together. For example, you can use the **replace()** method to replace a certain substring with another string, like this:

```
# Replacing a substring in a string
string1 = 'Hello world'
string2 = string1.replace('world', 'Python')
print(string2)

# Output: Hello Python
```

There are many other string methods in Python that you can use to manipulate strings, such as **upper()** to convert a string to uppercase, **lower()** to convert it to lowercase, **split()** to split a string into a list of substrings, and so on. You can find more information about these methods in the official Python documentation or by searching online.

★ *Exercises with strings*

Here are some exercises you can try to practice working with strings in Python:

- a) Create a string that contains your name and **print** it on the screen.
- b) Create a string that contains a quote or saying that you like and **print** it on the screen.
- c) Use the **upper()** method to convert the string from step 2 to **uppercase** and **print** the result to the screen.
- d) Use the **lower()** method to convert the string from step 2 to **lowercase** and **print** the result to the screen.
- e) Use the **replace()** method to replace a word in the string from step 2 with another word, and **print** the result to the screen.
- f) Use the **split()** method to split the string from step 2 into a list of substrings, and **print** the result to the screen.
- f) Use the **+** operator to concatenate (**join**) two or more strings together and **print** the result to the screen.
- h) Use the ***** operator to repeat a string multiple times and **print** the result to the screen.

Remember to try these exercises in a Python interpreter or a Python script, and feel free to experiment and modify the exercises to suit your own needs. Have fun!

! Have you reached this stage? Wow, you're already a serious candidate to understand data science! Congratulations!

Introducing functions

A **function** is a block of code that performs a specific task and returns a result. In Python, you can define **your functions** to reuse and organize your code. To define a **function**, you use the **def keyword** followed by the function name and a set of parentheses that may include arguments (also called parameters) separated by commas. The code block of the function, which contains the statements that the function will execute, is indented, and starts on the next line. Here is an example of a simple function in Python:

```
def greet():  
    print('Hello world!')
```

To call a **function**, you use the function name followed by a set of parentheses. When you call a function, the code inside the function is executed. Here is an example of calling a function in Python:

```
# Define the function  
def greet():  
    print('Hello world!')  
  
# Call the function  
greet()  
  
# Output: Hello world!
```

Functions can also take arguments, which are values that you can pass to the function when you call it. The **function** can then use these arguments to perform its task. For example, you can **define a function** that takes a name as an argument and greets the person with their name, like this:

```
def greet(name):  
    print('Hello ' + name + '!')  
  
greet('Ion')  
# Output: Hello Ion!  
  
greet('Jane')  
# Output: Hello Jane!
```

In Python, **functions** can also return a result using the **return key-word**. When a **function returns a result**, you can save the result in a variable or use it in an expression. Here is an example of a function that returns a result:

```
def square(number):  
    return number * number  
  
result = square(5)  
print(result)  
# Output: 25
```

You can also use the **returned** value of a **function** directly in an expression, like this:

```
print(square(2) + square(3))  
# Output: 13
```

These are just some of the basics of working with **functions** in Python. You can find more information about functions in the official Python documentation or by searching online.

[hints!] One tip for quickly learning and memorizing functions in Python is to practice using them in code. This could involve writing simple programs that use functions to perform tasks or working through exercises that involve functions. To create a function in Python, you need to use the **def** keyword, fol-

lowed by the name of the function, a set of parentheses, and a colon. For example:

```
def greet(name):  
    print("Hello, " + name + "!")
```

To call a function, you simply need to use its name followed by a set of parentheses. For example:

```
greet("Alice") # prints "Hello, Alice!"
```

You can also pass arguments to a function by including them within the parentheses when you call the function. For example:

```
def greet(name, age):  
    print("Hello, " + name + "! You are " +  
    str(age) + " years old.")  
  
greet("Bob", 30) # prints "Hello, Bob! You are 30  
years old."
```

Well Done!

Len() function

The **len()** function is a built-in function in Python that returns the length of an object. The object can be a string, a list, a tuple, or other types of objects that support the **len()** function (Lott, S. F., 2018). For example, you can use the **len()** function to get the **length of a string**, like this:

```
string = 'Hello world'  
print(len(string))  
# Output: 11
```

The **len()** function counts each character in the string, including whitespace and punctuation, and returns the total number of characters.

You can also use the **len()** function to get the length of a list, like this:

```
numbers = [1, 2, 3, 4, 5]  
print(len(numbers))  
# Output: 5
```

In this case, the **len()** function counts the number of elements in the list and returns the total. The **len()** function is often used in loops or other control structures to iterate over the elements of a sequence, such as a list or a string. For example, you can use the **len()** function with a for loop to iterate over the characters in a string, like this:

```
string = 'Hello world'
for i in range(len(string)):
    print(string[i])

# Output:
# H
# e
# l
# l
# o
#
# w
# o
# r
# l
# d
```

In this example, the **len()** function is used to get the length of the string, and the for **loop** uses the **length** to determine how many times to **iterate** over the string. These are just some of the ways you can use the **len()** function in Python.

Input

The **input()** function in Python is used to get input from the user. “By default, all processors will be loaded and run over the **input text**; however, users can also specify the processors to load and run with a list of processor names as an argument.” (Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D., 2020: 3). When you call the **input()** function, the program will pause and wait for the user to enter some text on the keyboard and press the enter key. The text that the user enters is then returned as a string by the **input()** function.

Here is an example of using the **input()** function in Python:

```
name = input('Enter your name: ')
print('Hello ' + name + '!')
```

When you run this code, the **input()** function will display the prompt Enter your name: on the screen, and wait for the user to enter their name. The user can then type their name and press the enter key, and the **input()** function will return their name as a string. The code then prints a greeting using the user's name. The **input()** function is often used in Python programs to get user input and perform some action or calculation. For example, you can use the **input()** function to get the user's age and calculate how many years they have left until retirement, like this:

```
age = int(input('Enter your age: '))
retirement_age = 65
years_left = retirement_age - age
print('You have ' + str(years_left) + ' years left
until retirement.')
```

In this code, the **input()** function is used to get the user's age as a string. The code then converts the string to an integer using the **int()** function and calculates the number of years until retirement using the user's age. Finally, the code converts the result to a string using the **str()** function and prints the result to the screen.

[hints!] To learn how to use the **input()** function in Python quickly and easily, you can follow these steps:

- First, familiarize yourself with the basic syntax of the **input()** function. The **input()** function takes a string as an argument, which is displayed as a prompt to the user. When the user enters some input and hits the Enter key, the **input()** function returns the input as a string. For example:

```
name = input("What is your name? ")
print("Hello, {}".format(name))
```

- Practice using the `input()` function by writing simple programs that ask the user for input and then process or output the input in some way. For example:

```
# Asks the user for their age and prints a message  
depending on their age  
age = int(input("How old are you? "))  
if age < 18:  
    print("You are a minor.")  
else:  
    print("You are an adult.")
```

- Experiment with different types of input, such as numbers, strings, and booleans, and learn how to convert the input from a string to the desired data type using type casting. For example:

```
# Asks the user for a number and prints the square  
of the number  
num = int(input("Enter a number: "))  
print("The square of the number is:", num**2)
```

Read through the documentation for the `input()` function and learn about its advanced features, such as customizing the prompt and handling errors.

Type Casting

Type casting is the process of converting a value from one data type to another data type. **Type casting** in Python allows you to change a value from one data type to another, such as converting an integer to a string or a string to a list. You can utilize built-in functions like `int()`, `float()`, `str()`, and `list()` to perform type conversions. These functions accept the value to be converted as an input and return the converted value. For example, you can use `int()` to convert a floating-point number to an integer or `str()` to convert a list to a string. **Type casting** is a useful technique for ensuring that values are the correct data type for a particular operation or for adapting values to meet the requirements of another system or software. For example, you can use the `int()` function to convert a string to an integer, like this:

```
string = '123'
number = int(string)
print(number)
# Output: 123
```

In this code, the string variable contains the string `'123'`. The `int()` function is then used to convert the string to an integer, and the result is saved in the `number` variable. You can also use the `float()` function to convert a string to a floating-point number, like this:

```
string = '3.14159'
number = float(string)
print(number)
# Output: 3.14159
```

In this code, the string variable contains the string `'3.14159'`. The `float()` function is then used to convert the string to a floating-point number, and the result is saved in the `number` variable. Here is another example of type distribution in Python:

```
# convert a string to an integer
num_str = "42"
num = int(num_str)

print(num + 1) # this will print 43
```

In this example, we convert the string `"42"` to an integer using the `int()` function, and then we add `1` to it. This shows how type casting can be useful when you need to perform arithmetic on a value that's stored as a string. **Type casting** in Python allows you to change values to other data types, such as float, complex, or bool. For instance, you can use the `float()` function to convert an integer to a floating-point number or the `complex()` function to convert a string to a complex number. However, it's important to keep in mind that type casting only works when the value can be represented in the target data type. For example, attempting to use type casting to convert a string with letters to an integer will result in an error because letters cannot be represented as integers. It's essential to be mindful of these limitations when using typecasting in your code.

F Strings

F-strings, or formatted string literals, are a new way to format strings in Python that was introduced in **Python 3.6**. They are called “**F-strings**” because they start with the letter “**F**” followed by a string enclosed in curly braces. The expression inside the curly braces is evaluated and the result is inserted in the string in place of the **F-string**. Here’s an example:

```
name = "Alice"
age = 30

# Using an F-string to insert the values of variables into a string
greeting = f"Hello, my name is {name} and I am {age} years old."

print(greeting) # Output: "Hello, my name is Alice and I am 30 years old."
```

F-strings provide a convenient and powerful way to include the values of variables in a string. They are faster and more readable than the old **%-formatting** syntax and they are easier to use than the **str.format()** method. If you’re using **Python 3.6** or later, you should use **F-strings** whenever you need to insert the values of variables into a string.

Unbelievably, you are the most persistent Data Science champion candidate!!!

★ *Age calculator exercise*

Here is an exercise you can try to practice working with “input” and “type casting” in Python:

- Create a program that asks the user to enter their age in years.
- Convert the user’s age to an integer using the **int()** function.
- Calculate the user’s age in months, days, hours, minutes, and seconds by multiplying the user’s age in years by the corresponding conversion factors:
 - 1 year = 12 months
 - 1 year = 365 days
 - 1 day = 24 hours

- 1 hour = 60 minutes
- 1 minute = 60 seconds

Print the results to the screen.

Here is an example of how the program could work:

```
Enter your age in years: 20
You are 240 months old.
You are 73,000 days old.
You are 1,752,000 hours old.
You are 105,120,000 minutes old.
You are 6,307,200,000 seconds old.
```

Remember to use the **input()** function to get the user's age, and use the **int()** function to convert the age to an integer. You can use the **print()** function to print the results to the screen. You can also use variables to store intermediate results, such as the user's age in months or days, to make the code easier to read and understand. Have fun with this exercise and feel free to modify it to suit your own needs!

★ *Shopping cart exercise*

Here's one possible way to implement a shopping cart in Python:

```
# Define a class to represent a shopping cart
class ShoppingCart:
    def __init__(self):
        # The shopping cart is initially empty
        self.items = []

    def add_item(self, item):
        # Add an item to the cart
        self.items.append(item)

    def remove_item(self, item):
        # Remove an item from the cart if it exists
        self.items.remove(item)

    def get_total(self):
        # Return the total cost of all items in the
        # cart
        total = 0
```

```
        for item in self.items:
            total += item.price
        return total

# Define a class to represent an item that can be
added to a shopping cart
class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

# Create a shopping cart
cart = ShoppingCart()

# Create some items
item1 = Item("apple", 0.5)
item2 = Item("banana", 0.25)
item3 = Item("orange", 0.75)

# Add the items to the cart
cart.add_item(item1)
cart.add_item(item2)
cart.add_item(item3)

# Print the total cost of the items in the cart
print(cart.get_total()) # should print 1.50

# Remove an item from the cart
cart.remove_item(item2)

# Print the total cost of the items in the cart
again
print(cart.get_total()) # should print 1.25
```

This implementation uses two classes: **ShoppingCart** and **Item**. The **ShoppingCart** class maintains a list of **Item** objects and provides methods for adding and removing items, as well as calculating the total cost of all items in the cart. The **Item** class represents a single item and has a name and price. To use this implementation, you first create an instance of the **ShoppingCart** class, which represents an empty

shopping cart. To use this implementation of a shopping cart, you would first need to create an instance of the **ShoppingCart** class. This would create an empty shopping cart where you could add items. To add items to the cart, you would need to create instances of the **Item** class, which represents a single item that can be added to a shopping cart. Each **Item** instance should have a name, a price, and a quantity. Once you have created instances of the **Item** class, you can add them to the shopping cart using the **add_item()** method of the **ShoppingCart** class. This method takes an **Item** instance as an argument and adds it to the shopping cart. You can also remove items from the shopping cart using the **remove_item()** method, which takes an **Item** instance as an argument and removes it from the shopping cart. The **ShoppingCart** class also includes methods for calculating the total price of the items in the cart, as well as the total number of items in the cart. These methods can be useful for calculating the total cost of the items in the cart before checkout. The **ShoppingCart** and **Item** classes provide a simple way to manage a shopping cart and the items in it.

The world of methods

In Python, a **method is a function** that is associated with an object. In other words, a method is a function that belongs to an object and can be used to manipulate that object or perform operations on it. Methods are defined in classes and can be called on instances of that class. For example, if you have a class **Person** that represents a person, you could define a method **say_hello()** that allows a person to greet someone else. You could then call this method on an instance of the **Person** class to have that person greet someone. Methods are defined using the **def** keyword, followed by the method name and a set of parentheses that may include arguments. Here is an example of a method definition:

```
class Person:
    def say_hello(self, name):
        print("Hello, " + name + "!")
```

The **say_hello()** method in this example takes a single argument, called **name**, which represents the name of the person being greeted. The **self**-parameter is a special argument that refers to the instance of

the class on which the method is being called. After defining a method in a class, it can be called on instances of that class using the dot notation. For example, if you have a class called **Greeting** with a **say_hello()** method, you can call the method on an instance of the **Greeting** class like this: **greeting_instance.say_hello("Ion")**. This will execute the **say_hello()** method and pass the string "Ion" as the value for the name argument. For example, if you have an instance of the **Person** class named **p**, you could call the **say_hello()** method like this:

```
p = Person()
p.say_hello("Ion")
```

This would print "Hello, Ion!" on the screen. Methods in Python provide a way to associate functions with objects, allowing you to perform operations on those objects and manipulate them in useful ways. **[hints!]** Methods are defined inside the class definition and are used to perform operations on the object or manipulate its attributes. For example, consider the following class definition:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print("Woof!")
```

In this example, the **bark()** method is defined inside the **Dog** class and can be called on any **Dog** object. To call a method on an object, you use the dot notation, like this:

```
dog = Dog("Fido", "Labrador")
dog.bark() # Outputs "Woof!"
```

Methods are a powerful feature of object-oriented programming in Python, and they allow you to write clean, modular code that is easy to understand and maintain.

[hints!] Methods are functions that are associated with a specific object or data type in Python. They allow you to perform operations on or retrieve information about an object and can be

an important part of working with data in Python. In data science, methods are often used to perform operations on data structures, such as lists, dictionaries, and pandas DataFrames. For example, you might use the `sort` method to sort a list, or the `mean` method to calculate the mean of a column in a DataFrame. Methods are also useful in the context of communication because they can help you create more readable and maintainable code. By using methods, you can encapsulate complex logic within a single function, rather than spreading it across multiple lines of code. This can make your code easier to understand and modify, which can be helpful when working on a team or sharing your code with others. The use of methods is an important aspect of programming in Python and is particularly useful for data science and communication tasks. By using these methods, you can more effectively manipulate and analyze data, and create code that is more readable and maintainable.

Introducing methods `Upper` and `Lower`

The `str.upper()` method is a built-in method in Python that can be used to convert all the characters in a string to uppercase. This method takes no arguments and returns a new string with all the characters in the original string converted to uppercase. Here is an example of how to use the `str.upper()` method:

```
my_string = "hello world"

# Convert the string to uppercase
my_string = my_string.upper()

# Print the result
print(my_string)
```

This code would print “HELLO WORLD” on the screen. The **str.lower()** method is similar to **str.upper()**, but it converts all the characters in a string to lowercase instead of uppercase. This method also takes no arguments and returns a new string with all the characters in the original string converted to lowercase. Here is an example of how to use the **str.lower()** method:

```
my_string = "HELLO WORLD"

# Convert the string to lowercase
my_string = my_string.lower()

# Print the result
print(my_string)
```

This code would print “hello world” to the screen. The **str.upper()** and **str.lower()** methods provide a simple and convenient way to convert strings to uppercase or lowercase, respectively. These methods can be useful in a variety of situations, such as when you need to compare two strings or when you want to standardize the case of a string.

Method of navigating documentation in Python

Python documentation is a comprehensive resource that provides detailed information about the Python language, its standard library, and its third-party packages. The documentation is available online at <https://docs.python.org/> and can be accessed at any time. To navigate the documentation, you can use the search bar at the top of the page to search for a specific term or concept. This will show you a list of relevant pages, along with a brief description of each page. You can click on any page in the list to view its contents. Alternatively, you can use the navigation menu on the left-hand side of the page to browse the documentation by topic. The menu is organized into several sections, each of which covers a different aspect of Python. For example, the “Language Reference” section contains detailed information about the Python language, including its syntax, data types, and built-in functions. The “Library Reference” section contains information about the modules and functions in the Python standard library, while the “Tutorials” section contains a series of tutorials that cover

various topics in Python. Once you locate a page with the desired information, you can read through it to learn more about the topic. The pages in the documentation are well-written and provide code examples and in-depth explanations of the covered concepts. Python documentation is an incredibly valuable resource for those looking to learn more about Python or build applications using the language. It is an essential tool for anyone interested in furthering their knowledge and skills in this popular programming language. It is well-organized, comprehensive, and easy to navigate, making it a great resource for anyone working with Python.

[hints!] If you are using an Integrated Development Environment (IDE) such as PyCharm, you can usually view the documentation for a specific object by hovering your mouse over it or by pressing a keyboard shortcut (e.g., F1 in PyCharm). It offers many features that can be helpful for beginners in Python and data science, including:

- **Code completion and auto-formatting:** PyCharm can help you write code more efficiently by suggesting completions for your code and automatically formatting it according to style guidelines.
- **Syntax highlighting:** PyCharm highlights different elements of your code, such as keywords and variables, to make it easier to read and understand.
- **Debugging:** PyCharm has a built-in debugger that allows you to pause your code, inspect variables, and step through your code line by line to identify and fix errors.
- **Testing:** PyCharm has tools to help you create and run tests for your code, which is important for ensuring that your code is reliable and correct.
- **Integrated development environment (IDE):** PyCharm is an all-in-one environment for writing, testing, and debugging code, which can be especially helpful for beginners who may not be familiar with the various tools and command-line interfaces that are often used for these tasks.

PyCharm can be a great tool for beginners in Python and data science because it provides a user-friendly interface and a wide range of helpful features that can make it easier to write, test, and debug code.

Help() & IPython

The **help()** function is a built-in function in Python that can be used to get information about a specific Python object, such as a module, function, or class (VanderPlas, J. 2016). When you call the **help()** function and pass it the name of an object, it will display detailed information about that object, including its definition, its methods and attributes, and any available documentation. For example, if you want to get information about the `str` class, which is used to represent strings in Python, you could call the **help()** function like this:

```
help(str)
```

This would display detailed information about the `str` class, including its methods, attributes, and documentation. In addition to using the **help()** function in the Python interpreter, you can also use it in the IPython interpreter, which is an enhanced version of the Python interpreter that provides additional features and functionality. To use the **help()** function in IPython, you can simply type **help()** followed by the name of the object you want to get information about (VanderPlas, J. 2016). For example:

```
In [1]: help(str)
```

This would display the same information about the `str` class as in the previous example. The **help()** function is a useful tool for getting detailed information about a specific Python object. It can be used in the Python interpreter or the IPython interpreter and can be a helpful resource when you are working with Python and need more information about a particular object.

Reading function Signatures + Strip methods

A **function signature** is a summary of a function's purpose, including its name, the types of its arguments, and the type of its return value. Function signatures are typically displayed in the documentation for a function and can provide useful information about how the function works and how to use it. Here is an example of a function signature:

```
def greet(name: str) -> str:
```

This function signature tells us that the function is named **greet**, it takes one argument named **name** which should be a string, and it returns a string. The **str.strip()** method is a built-in method in Python that can be used to remove leading and trailing whitespace from a string. This method takes no arguments and returns a new string with the leading and trailing whitespace removed. Here is an example of how to use the **str.strip()** method:

```
my_string = "    hello world    "

# Remove leading and trailing whitespace
my_string = my_string.strip()

# Print the result
print(my_string)
```

This code would print “hello world” to the screen, with the leading and trailing whitespace removed. The **str.lstrip()** and **str.rstrip()** methods are similar to **str.strip()**, but they only remove leading and trailing whitespace, respectively. These methods can be useful if you only need to remove leading or trailing whitespace, rather than both. Overall, function signatures and string strip methods can provide useful information and functionality when working with Python. Function signatures can help you understand how a function works, while string strip methods can be used to clean up and manipulate strings.

[**hints!**] To better learn about reading function signatures and strip methods in Python, you can follow these tips:

- Start by reading through the documentation for the built-in **strip()** method in Python. This method is used to remove leading and trailing whitespace from a string. You can learn about its syntax, arguments, and return value by reading the documentation. Practice using the **strip()** method by writing simple programs that manipulate strings. For example:

```
text = "    Hello, world!    "
print(text.strip()) # Outputs "Hello, world!"
```

- Learn about other string manipulation methods, such as **lstrip()** and **rstrip()**, which remove leading and trailing whitespace, respectively.

- Pay attention to the function signatures when you are reading through the documentation or using a function in your code. The function signature specifies the name of the function, the parameters it takes, and the return value. It is an important part of the documentation and helps you understand how to use the function correctly. For example, the function signature for the `strip()` method is:

```
str.strip(self, chars=None) -> str
```

- This tells us that the `strip()` method is a method of the `str` class and that it takes an optional `chars` parameter (which specifies the characters to remove from the string) and returns a new string.

Replace()

The **`str.replace()`** method is a built-in method in Python that can be used to replace a substring in a string with another string. This method takes two arguments: the substring to be replaced, and the replacement string. It returns a new string with the specified substring replaced by the replacement string.

Here is an example of how to use the **`str.replace()`** method:

```
my_string = "hello world"

# Replace "hello" with "goodbye"
my_string = my_string.replace("hello", "goodbye")

# Print the result
print(my_string)
```

This code would print “goodbye world” to the screen. Note that the **`str.replace()`** method only replaces the first occurrence of the substring in the string. If you want to replace all occurrences of the substring, you can use the **`re.sub()`** function from the **`re`** module, which is part of the Python standard library. The **`str.replace()`** method is a simple and convenient way to replace substrings in strings. It can be useful in a variety of situations, such as when you need to clean up or manipulate strings.

Other very useful string methods for data researchers

In addition to the **str.replace()** method, Python's **str** class also includes several other methods that can be used to manipulate strings. Here are some other useful string methods:

- The **str.split()** method can be used to split a string into a list of substrings based on a specified separator. For example, you could use this method to split a string into spaces to create a list of words in the string.
- The **str.join()** method can be used to join a list of strings into a single string, using a specified separator. This is the opposite of the **str.split()** method, and can be useful for combining multiple strings into a single string.
- The **str.format()** method can be used to insert values into a string, using placeholders. This is a convenient way to create strings with dynamic values and can be useful for generating strings with variable content.
- The **str.find()** method can be used to find the index of a substring within a string. This can be useful for determining the position of a substring within a string, or for checking whether a string contains a specific substring (Lubanovic, B., 2014).

These are just a few examples of the many methods available in the **str** class.

Method Chaining

Method chaining is a technique in which multiple methods are called on the same object, with each method being called on the result of the previous method. This allows you to perform multiple operations on an object in a single expression, without the need to store intermediate results in temporary variables. Here is an example of method chaining:

```
my_string = "  hello world  "

# Remove leading and trailing whitespace, and convert to uppercase
my_string = my_string.strip().upper()

# Print the result
print(my_string)
```

In this code, the `strip()` and `upper()` methods are both applied to the `my_string` object in a single expression. The `strip()` method is executed first, and its result (a new string with leading and trailing whitespace removed) is passed as an argument to the `upper()` method. This enables us to remove the leading and trailing whitespace and convert the string to uppercase in a single line, without the need to create a temporary variable to hold the intermediate result of the `strip()` method. Method chaining is a convenient way to perform multiple operations on an object in a single expression. It can help to simplify your code and make it more concise, by allowing you to avoid using temporary variables to store intermediate results. However, it's worth noting that method chaining can make your code less readable, as it can be challenging to understand the sequence of operations being applied to an object. For this reason, it's crucial to use method chaining thoughtfully and only when it enhances the readability and maintainability of your code. It's essential to consider the trade-offs and decide whether method chaining is the best approach for your specific situation.

[hints!] Method chaining can be a useful technique in data analysis because it allows you to concisely perform multiple operations on the same data in a single step, rather than having to write separate lines of code for each operation. For example, consider the following code, which uses method chaining to perform multiple operations on a pandas DataFrame:

```
df = pd.read_csv("data.csv")
df.columns = df.columns.str.lower()
df.rename(columns={"old_name": "new_name"},
          inplace=True)
df.set_index("id", inplace=True)
```

In this example, the `read_csv` method is called to read a CSV file into a DataFrame, and then several other methods are called to manipulate the DataFrame. By using method chaining, all these operations can be performed in a single line of code, which can make the code more readable and easier to understand. Method chaining is also useful in the context of communication because it can make it easier to share and collaborate on code. By using method chaining, you can cre-

ate more concise code that is easier for others to understand and modify. Method chaining is a useful technique in data analysis and communication because it allows you to concisely perform multiple operations on data and create more readable and maintainable code.

★ *Exercises with string methods*

Here are some exercises that you can try using the string methods discussed above:

- a) Write a Python program that takes a string as input and replaces all occurrences of the word “hello” with the word “goodbye”. Use the **str.replace()** method to perform the replacement.
- b) Write a Python program that takes a string as input and splits it into a list of words. Use the **str.split()** method to split the string into spaces.
- c) Write a Python program that takes a list of words as input and creates a new string by joining the words together, separated by spaces. Use the **str.join()** method to join the words together.
- d) Write a Python program that takes a string as input and removes any leading or trailing whitespace from the string. Use the **str.strip()** method to remove the whitespace.
- e) Write a Python program that takes a string as input and checks whether it contains the word “hello”. Use the **str.find()** method to find the index of the word “hello” within the string. If the word is found, print “Found!”; otherwise, print “Not found!”.

These exercises are designed to give you some practice using the string methods discussed above. You can try to solve them on your own, and then use the documentation or the **help()** function to learn more about the methods and how to use them.

Booleans

A **Boolean** is a **data type** that can have only two possible values: **True** or **False**. These values are often used to represent binary conditions, such as the result of a comparison or a test for the presence of an item in a sequence. For example, you might use a Boolean to check

whether a number is greater than or equal to another number or to check whether a string is empty. Here is an example of how you might use Booleans in Python:

```
# check if the number 5 is greater than or equal
to the number 10
if 5 >= 10:
    print("5 is greater than or equal to 10")
else:
    print("5 is not greater than or equal to 10")

# check if the string "hello" is empty
if "hello":
    print("The string is not empty")
else:
    print("The string is empty")
```

In the first example, the condition `5 >= 10` is `False`, so the code in the `else` block is executed and the message “5 is not greater than or equal to 10” is printed. In the second example, the condition `“hello”` is `True`, so the code in the `if` block is executed and the message “The string is not empty” is printed.

[hints!] Booleans are often used in data analysis to perform conditional operations, such as filtering data or choosing which code to execute based on certain conditions. For example, consider the following code, which uses Booleans to filter a pandas DataFrame:

```
df = pd.read_csv("data.csv")
mask = df["column"] > 0
filtered_df = df[mask]
```

In this example, the `mask` variable is a Boolean array that is `True` for rows in the DataFrame where the value in the “column” column is greater than 0, and `False` otherwise. The `filtered_df` variable is created by selecting only the rows in `df` where the corresponding element in the `mask` is `True`. Booleans are also useful in the context of communication because they can help you create more readable and maintainable code. By using Booleans to perform conditional operations, you can create code that is easier to understand and modify, which

can be helpful when working on a team or sharing your code with others. Booleans are an important aspect of programming in Python and are particularly useful for data analysis and communication tasks. By using Booleans, you can more effectively manipulate and analyze data, and create code that is more readable and maintainable.

Comparison operators

Comparison operators are used to comparing two values and determine whether one value is greater than, less than, or equal to the other. The comparison operators include `==` (equal to), `!=` (not equal to), `>` (greater than), `<` (less than), `>=` (greater than or equal to), and `<=` (less than or equal to). Here are some examples of how these operators can be used in Python:

```
# check if the number 5 is equal to the number 10
if 5 == 10:
    print("5 is equal to 10")
else:
    print("5 is not equal to 10")

# check if the number 5 is not equal to the number 10
if 5 != 10:
    print("5 is not equal to 10")
else:
    print("5 is equal to 10")

# check if the number 5 is greater than the number 10
if 5 > 10:
    print("5 is greater than 10")
else:
    print("5 is not greater than 10")

# check if the number 5 is less than the number 10
if 5 < 10:
    print("5 is less than 10")
else:
    print("5 is not less than 10")
```

```
# check if the number 5 is greater than or equal  
to the number 10  
if 5 >= 10:  
    print("5 is greater than or equal to 10")  
else:  
    print("5 is not greater than or equal to 10")  
  
# check if the number 5 is less than or equal to  
the number 10  
if 5 <= 10:  
    print("5 is less than or equal to 10")  
else:  
    print("5 is not less than or equal to 10")
```

In these examples, the code in the if block is only executed if the comparison operator is not greater than or equal to 10".

[hints!] Comparison operators are often used in data analysis to perform operations such as filtering data or making decisions based on certain conditions. For example, consider the following code, which uses comparison operators to filter a pandas DataFrame:

```
df = pd.read_csv("data.csv")  
filtered_df = df[df["column"] > 0]
```

In this example, the > operator is used to compare the values in the "column" column of the DataFrame to 0, and the resulting Boolean array is used to select only the rows in df where the comparison is True. Comparison operators are also useful in the context of communication because they can help you create more readable and maintainable code. By using comparison operators to perform conditional operations, you can create code that is easier to understand and modify, which can be helpful when working on a team or sharing your code with others. Comparison operators are an important aspect of programming in Python and are particularly useful for data analysis and communication tasks. By using comparison operators, you can more effectively manipulate and analyze data, and create code that is more readable and maintainable.

Comparing across types

In Python, it is possible to **compare values** of different types. For example, you can compare an **integer** and a **string** to see if the string contains the characters that make up the integer.

Here's an example:

```
# Define two values
x = 5
y = "5"

# Compare the values
if x == y:
    print("The values are equal")
else:
    print("The values are not equal")
```

In this example, the integer 5 is compared to the string "5". Since the string contains the characters that make up the integer, the comparison evaluates to True and the message "The values are equal" is printed. However, it's important to note that comparing values of different types can sometimes lead to unexpected behavior. For example, if you compare an integer and a string that contains a number that is different from the integer, the comparison will evaluate as False even though the string and integer may "look" the same.

Here's an example:

```
# Define two values
x = 5
y = "6"

# Compare the values
if x == y:
    print("The values are equal")
else:
    print("The values are not equal")
```

In this example, the integer 5 is compared to the string "6". Since the string does not contain the characters that make up the integer, the comparison evaluates to False and the message "The values are not

equal” is printed. In general, it is best to avoid comparing values of different types in Python. If you need to compare values of different types, you should first make sure that they are the same type using a type conversion function like `int()` or `str()`.

Truthiness & Falseyness

In Python, every value has a boolean truth value, which is either **True** or **False** (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620). This **true** value is used in conditionals and other contexts where a boolean value is expected. Some values are considered “truthy”, which means that they evaluate to **True** when used in a boolean context. For example, non-zero numbers, non-empty strings, and non-empty lists are all considered truthy. Other values are considered “falsey”, which means that they evaluate to **False** when used in a boolean context. The most common “falsey” values are **0**, the empty string “”, and the empty list [].

Here’s an example of how truthiness and falseyness can be used in a conditional:

```
# Define a truthy value
x = 5

# Define a falsey value
y = 0

# Check the truth value of x
if x:
    print("x is truthy")

# Check the truth value of y
if y:
    print("y is truthy")
```

In this example, the first **if** statement checks the truth value of **x**. Since **x** is a non-zero number, it is considered truthy and the message “x is truthy” is printed. The second **if** statement checks the truth value of **y**. Since **y** is **0**, it is considered falsey, and the **if** statement is not executed. It’s important to remember that truthiness and falseness only apply in boolean contexts, such as in **if** statements or when using the “**and**” and **or** operators. In other contexts, values may behave dif-

ferently. For example, a value that is falsey in a boolean context may be considered a valid input in other contexts, such as when used as a list index or as a number in a mathematical expression.

The “in” operator

The **in** operator is a Python operator that allows you to check whether a value is a member of a sequence, such as a list or a string.

Here’s an example of how to use the **in** operator to check if a value is in a list:

```
# Define a list
my_list = [1, 2, 3, 4, 5]

# Check if the value 3 is in the list
if 3 in my_list:
    print("3 is in the list")
```

In this example, the **in** operator is used to check if the value **3** is in the list **my_list**. Since **3** is an element of **my_list**, the **in** operator evaluates to **True** and the message “**3 is in the list**” is printed.

You can also use the **in** operator to check if a value is in a string:

```
# Define a string
my_string = "Hello, world!"

# Check if the value "world" is in the string
if "world" in my_string:
    print("world is in the string")
```

In this example, the **in** operator is used to check if the substring “world” is in the string **my_string**. Since “world” is a substring of **my_string**, the **in** operator evaluates to **True** and the message “world is in the string” is printed. The **in** operator can also be used with other sequence types in Python, such as **tuples** and **sets**. It can also be used to check if a key is in a dictionary. For example:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}

if 'key1' in my_dict:
    print('key1 is in the dictionary')
```

In this example, we create a dictionary called **my_dict** that has two keys, **key1** and **key2**. We then use the **in** operator to check if **key1** is in **my_dict**. If it is, the code prints “**key1 is in the dictionary**”. The “**in**” operator can be utilized with dictionaries to determine if a specific key is present in the dictionary. However, it can also be used with other data structures in Python, such as lists, sets, and tuples. For instance, you can use it to check if a particular element is in a list or if an element is in a set. The “**in**” operator is a versatile tool that allows you to quickly check the membership of an element in various data structures in Python.

Comparing strings

In Python, you can compare strings using the comparison operators, such as **==**, **!=**, **>**, **<**, **>=**, and **<=**. These operators will compare the values of the strings and return a Boolean value (**True** or **False**) based on the result of the comparison. “The Boolean satisfiability problem (sometimes referred to as the propositional satisfiability problem) is a problem in logic and computer science that involves assessing if an explanation exists that satisfies a particular Boolean formula. In other words, it determines whether a variable specified by a Boolean formula can be consistently substituted with a TRUE or FALSE value such that the formula evaluates to TRUE. If this is TRUE, the formula is satisfactory. If such assignments do not exist, the function described by the formula is FALSE for all conceivable variable terms, rendering the formula unsatisfiable” (Lu, W., 2022). Here’s an example of using the **==** operator to compare two strings:

```
string1 = "hello"
string2 = "hello"

if string1 == string2:
    print("The strings are equal")
```

In this example, we create two variables called **string1** and **string2** and assign the string “hello” to each of them. We then use the **==** operator to compare the values of the two strings. Since the values of the two strings are the same, the **if** statement will evaluate to **True** and the code will print “The strings are equal”. Another thing to keep in

mind when comparing strings in Python is that the comparison is case-sensitive. This means that the strings “Hello” and “hello” would not be considered equal, since the capitalization is different. To compare strings in a case-insensitive manner, you can convert the strings to either upper-case or lower-case before performing the comparison. Here’s an example:

```
string1 = "Hello"
string2 = "hello"

if string1.lower() == string2.lower():
    print("The strings are equal")
```

In this example, we convert both strings to lower-case using the `lower()` method before performing the comparison. This ensures that the comparison is case-insensitive and the `if` statement will evaluate to **True**, printing “The strings are equal”. Here are some more complex examples of Python code to illustrate the concept of the **Boolean** satisfiability problem:

```
def is_satisfiable(formula: str) -> bool:
    """Returns True if the given Boolean formula
    is satisfiable, False otherwise."""
    # TODO: Implement an algorithm to check the
    # satisfiability of a formula

    formula1 = "A and B"
    formula2 = "A and not B"
    formula3 = "A or B"
    formula4 = "A xor B"

    print(is_satisfiable(formula1))    # Output: True
    print(is_satisfiable(formula2))    # Output: True
    print(is_satisfiable(formula3))    # Output: True
    print(is_satisfiable(formula4))    # Output: True
```

```
from typing import List

def is_satisfiable(formula: str, variables:
List[str]) -> bool:
    """Returns True if the given Boolean formula
    is satisfiable, False otherwise."""
    # TODO: Implement an algorithm to check the
    satisfiability of a formula

formula1 = "A and B"
formula2 = "A and not B"
formula3 = "A or B"
formula4 = "A xor B"

variables1 = ["A", "B"]
variables2 = ["A", "B", "C"]
variables3 = ["A"]
variables4 = ["A", "B", "C", "D"]

print(is_satisfiable(formula1, variables1))
# Output: True
print(is_satisfiable(formula2, variables2))
# Output: True
print(is_satisfiable(formula3, variables3))
# Output: True
print(is_satisfiable(formula4, variables4))
# Output: True
```

These examples show how the **Boolean** satisfiability problem can be addressed in Python using a function that takes a **Boolean** formula and a list of variables as input and returns a **Boolean** value indicating whether the formula is satisfiable. The implementation of the `is_satisfiable()` function would depend on the specific algorithm used to solve the satisfiability problem.

★ *Exercises with Booleans*

Here are some examples of exercises that you can try using Booleans in Python:

- a) Write a Python program to check if a number is odd or even.

```
def is_odd_or_even(num):  
    if num % 2 == 0:  
        return True  
    else:  
        return False  
  
print(is_odd_or_even(2)) # True (even)  
print(is_odd_or_even(3)) # False (odd)
```

- b) Write a Python program to check if a string is uppercase or lowercase.

```
def is_uppercase_or_lowercase(string):  
    if string.isupper():  
        return True  
    elif string.islower():  
        return False  
  
print(is_uppercase_or_lowercase('HELLO')) # True  
      (uppercase)  
print(is_uppercase_or_lowercase('hello')) # False  
      (lowercase)
```

- c) Write a Python program to check if a number is positive, negative, or zero.

```
def is_positive_negative_or_zero(num):  
    if num > 0:  
        return 'positive'  
    elif num < 0:  
        return 'negative'  
    else:  
        return 'zero'  
  
print(is_positive_negative_or_zero(5)) # 'positive'  
print(is_positive_negative_or_zero(-5)) # 'negative'  
print(is_positive_negative_or_zero(0)) # 'zero'
```

- a) Write a program that prompts the user for two integers and prints whether the first integer is greater than, less than, or equal to the second.
- a) Write a function that takes a list of integers as an argument and returns True if the list contains at least one even number, and False otherwise.
- a) Write a program that reads a CSV file into a pandas DataFrame and filters the DataFrame to only include rows where the value in a specific column is greater than 0.
- a) Write a function that takes a string as an argument and returns True if the string is a palindrome (i.e., it reads the same forwards and backward), and False otherwise.
- a) Write a program that prompts the user for a password and checks whether the password is at least 8 characters long and contains at least one uppercase letter, one lowercase letter, and one digit.

Conditionals basics

A **conditional statement** is a type of control flow statement that allows you to execute a certain block of code only if a certain condition is met. This is done using the **if** keyword, followed by the condition that you want to check. For example:

```
if condition:
    # code to execute if the condition is True
```

You can also include an **else** clause to specify what should happen if the condition is not met:

```
if condition:
    # code to execute if the condition is True
else:
    # code to execute if the condition is False
```

You can also include an **elif** clause (short for “else if”) to specify additional conditions that should be checked:

```
if condition1:
    # code to execute if the condition1 is True
elif condition2:
    # code to execute if the condition1 is False
    and condition2 is True
else:
    # code to execute if the condition1 and
    condition2 are False
```

Here’s an example of using a conditional statement to check if a number is positive, negative, or zero:

```
num = 5

if num > 0:
    print(num, "is positive")
elif num < 0:
    print(num, "is negative")
else:
    print(num, "is zero")
```

This code would print “5 is positive” to the screen. It’s important to note that Python uses indentation to denote blocks of code, so it’s important to make sure that your code is properly indented to avoid syntax errors.

[hints!] Conditionals basic are often used in data analysis to perform operations such as filtering data or making decisions based on certain conditions. For example, consider the following code, which uses a conditional statement to filter a pandas DataFrame:

```
df = pd.read_csv("data.csv")
if df["column"].mean() > 0:
    df = df[df["column"] > 0]
```

In this example, the if statement is used to check whether the mean of the values in the “column” column of the DataFrame is greater than 0. If the condition is True, the df DataFrame is filtered to only include rows where the value in the “column” column is greater than 0. Conditional statements are also useful in the context

of communication because they can help you create more readable and maintainable code. By using conditional statements to perform operations based on certain conditions, you can create code that is easier to understand and modify, which can be helpful when working on a team or sharing your code with others. Conditional statements are an important aspect of programming in Python and are particularly useful for data analysis and communication tasks. By using conditional statements, you can more effectively manipulate and analyze data, and create code that is more readable and maintainable.

Name length codealong

Here's a simple example of how you can use a conditional statement to check the length of a person's name and print out a message based on that length:

```
name = "Ioana"

if len(name) < 4:
    print("That's a short name!")
elif len(name) > 4:
    print("That's a long name!")
else:
    print("That's a medium-length name!")
```

In this code, we first define a **named** variable and set it equal to "Ioana". We then use an if statement to check if the length of the **name** is less than 4. If it is, we print a message saying, "That's a short name!". Next, we use an **elif** clause to check if the length of the **name** is greater than 4. If it is, we print a message saying, "That's a long name!". Finally, we use an **else** clause to specify what should happen if none of the other conditions are met. In this case, we simply print a message saying, "That's a medium-length name!". Note that the **len()** function is used to get the length of the **name** string. This function returns the number of characters in the string, which we can then use to check against our conditions.

[hints!] Length is a property that can be used to determine the number of elements in a sequence, such as a list or string. In Python, you can use the **len** function to get the length of a se-

quence. Here is an example of using the `len` function to get the length of a list:

```
# Define a list
my_list = [1, 2, 3, 4, 5]

# Get the length of the list
list_length = len(my_list)

# Print the length
print(list_length) # Output: 5
```

You can also use the `len` function to get the length of a string:

```
# Define a string
my_string = "Hello, world!"

# Get the length of the string
string_length = len(my_string)

# Print the length
print(string_length) # Output: 13
```

A “codealong” is a type of tutorial or lesson in which you follow along with the instructor and write code as they explain different concepts. A “length codealong” might be a tutorial or lesson that focuses on using the `len` function to get the length of different types of sequences in Python.

A tangent on indentation

Indentation is an important aspect of Python’s syntax, as it is used to denote blocks of code. In Python, blocks of code are defined by their indentation level, and statements that are at the same indentation level are considered to be part of the same block. For example, consider the following code:

```
if condition:
    # code to execute if the condition is True
else:
    # code to execute if the condition is False
```

In this code, the **if** statement and the **else** clause are at the same indentation level, so they are considered to be part of the same block of code. This means that the code inside the **if** statement will only be executed if the condition is **True**, and the code inside the **else** clause will only be executed if the condition is **False**. In Python, the recommended indentation level is 4 spaces. However, you can use any number of spaces as long as it is consistent throughout your code. It's worth noting that indentation is significant in Python and not using the correct indentation can lead to syntax errors. Therefore, it's crucial to ensure that your code is correctly intended to avoid such issues. Proper indentation is important in Python and should be taken into consideration when writing and organizing your code.

Nesting conditionals

In Python, you can nest conditional statements inside other conditional statements to create more complex logical structures. This is useful when you want to check multiple conditions and take different actions based on those conditions. Here's an example of how you might nest conditional statements to check if a number is positive, negative, or zero:

```
num = 5

if num == 0:
    print(num, "is zero")
else:
    if num > 0:
        print(num, "is positive")
    else:
        print(num, "is negative")
```

In this code, we first check if the value of **num** is equal to 0. If it is, we print a message saying, "5 is zero". If the value of **num** is not 0, the **else** clause is executed, and we enter another **if** statement to check if the value of **num** is greater than 0. If it is, we print a message saying, "5 is positive". If the value of **num** is not greater than 0, we enter the **else** clause of the second if statement and print a message saying, "5 is negative". As you can see, nesting conditional statements can help you create more complex logical structures in your code. However, it's

important to use them carefully and not overdo them, as nested conditionals can make your code difficult to read and understand.

★ *Water boiling Codealong*

Here's an example of how you might use a conditional statement in Python to check if water is boiling at a given temperature:

```
temperature = 100

if temperature >= 100:
    print("The water is boiling!")
else:
    print("The water is not boiling.")
```

In this code, we first define a **temperature** variable and set it equal to 100. We then use an **if** statement to check if the temperature is greater than or equal to 100 (the boiling point of water at sea level). If it is, we print a message saying, "The water is boiling!". If the temperature is not greater than or equal to 100, the **else** clause is executed and we print a message saying "The water is not boiling.". Note that we use the greater than or equal to (**>=**) operator to check if the temperature is greater than or equal to 100. This operator returns a boolean value (either **True** or **False**) depending on whether the condition is met or not.

★ *BMI calculator exercise*

Here's an example of how you might use a conditional statement in Python to calculate a person's body mass index (BMI) and print out a message based on that value:

```
weight = 68 # in kilograms
height = 1.75 # in meters

bmi = weight / (height * height)

if bmi < 18.5:
    print("Underweight")
elif bmi >= 18.5 and bmi < 25:
    print("Normal weight")
```

```
elif bmi >= 25 and bmi < 30:
    print("Overweight")
else:
    print("Obese")
```

In this code, we first define two variables, **weight** and **height**, and set them equal to a person's weight and height in kilograms and meters, respectively. We then use these values to calculate the person's BMI using the following formula:

$$\text{BMI} = \text{weight} / (\text{height} * \text{height})$$

Next, we use a series of **if** and **elif** clauses to check the value of the **BMI** variable and print out a message based on that value. If the **BMI** is less than 18.5, we print a message saying "Underweight". If it is greater than or equal to 18.5 and less than 25, we print a message saying, "Normal weight". If it is greater than or equal to 25 and less than 30, we print a message saying "Overweight". And finally, if none of the other conditions are met, we print a message saying "Obese". Note that we use the **and** keyword to combine multiple conditions in our **elif** clauses. This keyword allows us to check if multiple conditions are met at the same time.

★ *Tweet checker exercise*

Here's an example of how you might approach this problem using a conditional statement in Python:

```
tweet = "Just learned about conditionals in
Python! #learning #python"

if "#learning" in the tweet and "#python" in the
tweet:
    print("This tweet is about learning Python!")
else:
    print("This tweet is not about learning
Python.")
```

In this code, we first define a **tweet** variable and set it equal to a sample tweet. We then use an if statement to check if both the "#learning" and "#python" hashtags are present in the tweet. If they are, we print a message saying, "This tweet is about learning Python!". If the

hashtags are not present, the **else** clause is executed and we print a message saying “This tweet is not about learning Python.”. Note that we use the **in** keyword to check if a given string is present in another string. This keyword returns a boolean value (either **True** or **False**) depending on whether the string is found or not.

Writing more complex logic

To write more complex logic in Python, you can use conditional statements, loops, and functions. Conditional statements allow you to check if a certain condition is true, and then execute a block of code only if the condition is true. For example, you can use an if statement to check if a number is greater than 5, and then print a message if it is:

```
if number > 5:  
    print("The number is greater than 5.")
```

Loops allow you to execute a block of code multiple times. For example, you can use a for loop to iterate over a list of numbers and print each one:

```
for number in [1, 2, 3, 4, 5]:  
    print(number)
```

Functions allow you to define a block of reusable code that can be called from anywhere in your program. Using **functions** in your code allows you to reuse the same code multiple times, making your program more modular and easier to read and understand. For instance, you can define a function that takes a number as an input and returns its square. This can be useful for performing a common operation in your code without having to repeat the same lines of code multiple times. Here’s an example of how you might define and use a function in Python:

```
def square(x: int) -> int:  
    return x ** 2  
  
print(square(2))    # Output: 4  
print(square(3))    # Output: 9  
print(square(4))    # Output: 16
```

In this example, the `square()` function takes an integer as an argument and returns the square of that number. The function is called three times with different values, and the squared result is printed each time. Functions are a powerful feature in Python that can help you write more organized and reusable code.

```
def square(number):  
    return number * number  
  
print(square(5)) # Output: 25  
print(square(10)) # Output: 100
```

By using these tools, you can write more complex logic in Python to solve a wide variety of problems.

[hints!] Writing more complex logic in data analysis and communication can be helpful for a variety of reasons:

- It allows you to solve more complex problems: By writing more complex logic, you can tackle more challenging problems and find more robust solutions.
- It makes your code more efficient: More complex logic can often be more efficient, as it can allow you to perform multiple operations in a single step rather than having to write separate lines of code for each operation.
- It makes your code more readable and maintainable: By encapsulating complex logic within a single function or block of code, you can create code that is easier to read and understand, which can be helpful when working on a team or sharing your code with others.

Writing more complex logic is an important skill to have in data analysis and communication, as it can help you tackle more challenging problems, make your code more efficient, and create code that is more readable and maintainable.

Logical AND

The logical **AND** operator is represented by the **and** keyword. It is used to combine two or more conditions and returns **True** if all the conditions are **True**, and **False** otherwise. For example, the following

code uses the “**and**” operator to check if a number is greater than 5 and less than 10:

```
if number > 5 and number < 10:
    print("The number is between 5 and 10.")
```

The **and** operator can also be used with other types of values, such as strings and lists. For example, the following code uses the **and** operator to check if a string contains the letter “a” and has more than 10 characters:

```
if "a" in string and len(string) > 10:
    print("The string contains the letter 'a' and
has more than 10 characters.")
```

Note that the **and** operator has higher precedence than the comparison operators (>, <, **in**, etc.), so the conditions in the above examples will be evaluated in the following order:

```
number > 5 and number < 10  # Evaluates to True or
False
"a" in string and len(string) > 10  # Evaluates to
True or False
```

You can use parentheses to change the order of evaluation if necessary. For example, the following code will evaluate the `len()` function before the `in` operator:

```
if (len(string) > 10) and ("a" in string):
    print("The string has more than 10 characters
and contains the letter 'a'.")
```

Logical OR

The logical **OR** operator is represented by the **or** keyword. It is used to combine two or more conditions and returns **True** if at least one of the conditions is **True**, and **False** otherwise. For example, the following code uses the **or** operator to check if a number is greater than 5 or less than 10:

```
if number > 5 or number < 10:
    print("The number is greater than 5 or less than
10.")
```

The **or** operator can also be used with other types of values, such as strings and lists. For example, the following code uses the **or** operator to check if a string contains the letter “a” or the letter “b”:

```
if "a" in string or "b" in string:
    print("The string contains the letter 'a' or the
    letter 'b'.")
```

Note that the **or** operator has lower precedence than the comparison operators (**>**, **<**, **in**, etc.), so the conditions in the above examples will be evaluated in the following order:

```
number > 5 or number < 10   # Evaluates to True or
False
"a" in string or "b" in string # Evaluates to
True or False
```

You can use parentheses to change the order of evaluation if necessary. For example, the following code will evaluate the **in** operator before the **or** operator:

```
if ("a" in string) or ("b" in string):
    print("The string contains the letter 'a' or the
    letter 'b'.")
```

Logical NOT

The logical **NOT** operator is represented by the **not** keyword. It is a boolean operator that takes a boolean value as its operand and returns the opposite boolean value. For example:

```
# The expression `not True` evaluates to `False`
print(not True)
# The expression `not False` evaluates to `True`
print(not False)
```

The logical **NOT** operator is represented also by the exclamation point (!). It is a unary operator that negates the boolean value of its operand. For example:

```
x = True
y = !x  # y is False

a = False
b = !a  # b is True
```

Here, `x` and `a` are boolean variables with the values **True** and **False**, respectively. The `!` operator negates their values, so `y` is **False** and `b` is **True**. To use the logical **NOT** operator, you simply place the `!` symbol before the operand. This will negate the boolean value of the operand, so if the operand is **True**, the result will be **False**, and vice versa.

Truthy/Falsey testing

The **bool** type is used to represent the truth or falsity of a value or expression. The two possible values for a **bool** are **True** and **False**, which are often abbreviated as **True** and **False**. To evaluate whether a value or expression is **True** or **False**, you can use the `bool()` function. For example:

```
bool(0)    # False
bool(1)    # True
bool(-1)   # True
bool('')   # False
bool(' ')  # True
bool('hello') # True
```

In Python, any non-zero number or non-empty string is considered **True**, while `0` and the empty string (`''`) are considered **False**. It's also possible to use the **not** operator to negate the truth value of a statement. For example:

```
not False # True
not True  # False
```

You can also use the **and** and **or** operators to combine multiple truth values. For example:

```
True and True # True
True and False # False
False and False # False

True or True # True
True or False # True
False or False # False
```

The **and** operator returns **True** only if both operands are **True**, while the **or** operator returns **True** if either operand is **True**.

Logical operator precedence

In Python (and most programming languages), the order in which logical operators are applied to evaluate a given expression is determined by the rules of operator precedence. The logical operators **and** and **or** have different precedence levels, with **and** having higher precedence than **or**. This means that when an expression contains both **and** and **or** operators, the **and** operator will be applied first. For example, consider the following expression:

```
True or False and False
```

Since **and** has higher precedence than **or**, this expression will be evaluated as:

```
True or (False and False)
```

The **and** operator is applied first, resulting in **False**, which is then passed as the right operand to the **or** operator. Since one of the operands of the **or** operator is **True**, the overall expression evaluates to **True** (Lubanovic, B., 2014). On the other hand, if we use parentheses to change the order of evaluation, the result of the expression will be different:

```
(True or False) and False
```

In this case, the **or** operator is applied first, resulting in **True**, which is then passed as the left operand to the **and** operator. Since the

other operand of the **and** operator is **False**, the overall expression evaluates to **False**. To avoid ambiguity and ensure that your code is correct, it's a good idea to use parentheses to explicitly specify the order of evaluation for logical operators in complex expressions.

★ *Exercises with logical AND, OR, and NOT*

Here are some examples of exercises that you can try to practice using the logical **AND**, **OR**, and **NOT** operators in Python:

- a) Write a Python program that takes two boolean values *x* and *y* as input and outputs the result of the logical AND operation on *x* and *y*.

```
x = True
y = False

# Your code here

result = x and y
print(result)  # This should output False
```

- b) Write a Python program that takes two boolean values *a* and *b* as input and outputs the result of the logical OR operation on *a* and *b*.

```
a = True
b = False

# Your code here

result = a or b
print(result)  # This should output True
```

- c) Write a Python program that takes a boolean value *c* as input and outputs the negated value of *c*.

```
c = True

# Your code here

result = not c
print(result)  # This should output False
```

- d) Write a Python program that takes three boolean values *x*, *y*, and *z* as input and outputs the result of the logical AND operation on *x* and *y*, and the result of the logical OR operation on *y* and *z*.

```
x = True
y = False
z = True

# Your code here

result_1 = x and y
result_2 = y or z
print(result_1)  # This should output False
print(result_2)  # This should output True
```

Loops

In programming, a **loop** is a control flow construct that allows you to repeat a block of code a certain number of times or until a certain condition is met. Loops are an essential part of any programming language and can be used to perform a wide variety of tasks. There are two main types of loops in Python: **for** loops and **while** loops. A **for** loop is used to iterate over a sequence of elements, such as a list or a string (Lubanovic, B., 2014). The syntax for a **for** loop is as follows:

```
for element in sequence:
    # code block to be executed
```

The **for** loop iterates over the elements in the given sequence, and for each element, it executes the code block inside the loop. Here's an example of a **for** loop that prints the elements of a list:

```
fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:
    print(fruit)
```

This code will print the following:

```
apple
banana
cherry
```

A **while** loop, on the other hand, is used to execute a code block until a certain condition is met. The syntax for a **while** loop is as follows:

```
while condition:
    # code block to be executed
```

The code block inside the **while** loop will be executed repeatedly if the condition evaluates to **True**. Here's an example of a **while** loop that prints the numbers from 1 to 10:

```
i = 1
while i <= 10:
    print(i)
    i += 1
```

This code will print the following:

```
1
2
3
4
5
6
7
8
9
10
```

[**hints!**] Loops are useful in data analysis and communication because they allow you to automate repetitive tasks and processes, which can save time and effort. In data analysis, loops can be used to iterate over a large dataset, perform calculations or transformations on the data, and then present the results clearly and concisely manner. For example, you might use a loop to calculate the mean, median, and standard deviation

of a set of numerical data, and then display the results in a table or graph to communicate your findings to others. In addition to being efficient, loops can also improve the clarity and readability of your code. By using loops, you can avoid writing out the same code repeatedly, which can make your code more organized and easier to understand. This is especially important when working on large, complex data analysis projects, where clear and concise code is essential for effective collaboration and communication.

Avoiding infinite loops

An infinite loop is a loop that continues to run indefinitely because its stopping condition is never met. In other words, the loop's condition always evaluates to **True**, so the loop never ends. Infinite loops can be dangerous because they can cause your program to crash or hang, and they can also consume a lot of resources, such as memory and CPU time. To avoid infinite loops, you should always make sure that the loop's stopping condition will eventually be met. In the case of a **for** loop, this means that the sequence you're iterating over must be finite (i.e., it must have a finite number of elements). In the case of a **while** loop, this means that the condition must eventually become **False**. Here are a few examples of infinite loops and how to fix them:

```
# Infinite for loop
for i in range(10):
    print(i)
    i -= 1 # this is the problem - the loop will
never end because i is always 10

# Fix: remove the line that decreases i
for i in range(10):
    print(i)
```

```
# Infinite while loop
i = 0
while True:  # this is the problem - the condition
             # is always True, so the loop will never end
    print(i)
    i += 1

# Fix: add a condition that makes the loop stop at
# some point
i = 0
while i < 10:  # the loop will run 10 times and
              # then stop
    print(i)
    i += 1
```

By following these tips, you can avoid creating infinite loops and keep your programs running smoothly.

The range() function

The **range()** function in Python is a built-in function that returns a sequence of numbers. It is commonly used in for loops to iterate over a sequence of numbers. Here is an example of using **the range()** function to iterate over a sequence of numbers and print them:

```
for i in range(10):
    print(i)
```

This code will print the numbers 0 through 9 (inclusive) on separate lines. The **range()** function takes up to three arguments: a start value, an end value, and a step value. The start value is the first number in the sequence, the end value is the last number in the sequence, and the step value is the difference between consecutive numbers in the sequence. If the start value is not provided, it defaults to 0. If the step value is not provided, it defaults to 1. Here are some examples of using the **range()** function with different arguments:

```
# Start at 1, end at 10, step by 1
for i in range(1, 10):
    print(i)
```

```
# Start at 0, end at 10, step by 2
for i in range(0, 10, 2):
    print(i)
```

In the first example, the `range()` function starts at 1 and ends at 10, with a step value of 1, so it generates the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9. In the second example, the `range()` function starts at 0 and ends at 10, with a step value of 2, so it generates the sequence 0, 2, 4, 6, 8.

Working with Nested Loops

Nested loops are loops that are written within the body of another loop. They are commonly used when you want to iterate over the elements of a nested data structure, such as a list of lists. Here is an example of a nested loop:

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

for row in numbers:
    for number in row:
        print(number)
```

In this code, the outer **for** loop iterates over the rows of the **numbers** list, and the inner **for** loop iterates over the numbers in each row. The output of this code will be the numbers 1 through 9, each on a separate line. When working with nested loops, it is important to keep track of the loop variables and their scope. In the example above, the outer loop uses the loop variable **row** to refer to each row of the **numbers** list, and the inner loop uses the loop variable **number** to refer to each number in a row. These loop variables are only defined within the body of the loop, so they are not accessible outside of the loop.

Break and continue keywords

The **break** keyword is used to exit a loop and transfer execution to the statement immediately following the loop. For example:

```
for i in range(1, 11):
    if i == 5:
        break
    print(i)
```

In this code, the loop will iterate over the numbers 1 through 10. When the value of `i` is 5, the **break** statement is executed and the loop is terminated. The output of this code will be numbers 1 through 4. The **continue** keyword is used to skip the rest of the current iteration of a loop and move on to the next iteration. For example:

```
for i in range(1, 11):  
    if i % 2 == 0:  
        continue  
    print(i)
```

In this code, the loop will iterate over the numbers 1 through 10. For each iteration, if the value of `i` is even, the **continue** statement is executed and the loop continues to the next iteration without executing the rest of the code in the loop. The output of this code will be the odd numbers from 1 to 10. Both the **break** and **continue** keywords can be useful when you want to control the flow of a loop based on certain conditions.

[hints!] The **break** and **continue** keywords are useful in data analysis and communication because they allow you to control the flow of your loops and selectively skip certain iterations. The **break** keyword is used to exit a loop prematurely, which can be useful if you want to stop iterating through a dataset once you have found the data you are looking for. For example, you might use a **break** statement to stop searching for a specific value in a dataset once it has been found. The **continue** keyword is used to skip the remainder of the current iteration of a loop and move on to the next iteration. This can be useful if you want to skip certain iterations based on certain criteria. For example, you might use a **continue** statement to skip over rows in a dataset that is missing certain data points. Both the **break** and **continue** keywords can be used to improve the efficiency of your code and to make it easier to understand. They can also be used to communicate your intent more clearly to others, as they allow you to explicitly state your intentions in your code.

★ 99 Bottles of Beer Codealong

Here is a Python program that uses a for loop to print the lyrics to the song “99 Bottles of Beer on the Wall”:

```
# Start at 99 and end at 0
for i in range(99, -1, -1):
    # If there are no more bottles, print a special
    # message
    if i == 0:
        print("No more bottles of beer on the wall, no
        more bottles of beer.")
        print("Go to the store and buy some more, 99
        bottles of beer on the wall.")
    else:
        # Print the current number of bottles
        print(f"{i} bottles of beer on the wall, {i}
        bottles of beer.")
        # Decrement the number of bottles
        j = i - 1
        # If there is only one bottle left, use the
        # singular form "bottle"
        if j == 1:
            print(f"Take one down and pass it around,
            {j} bottle of beer on the wall.")
        # Otherwise, use the plural form "bottles"
        else:
            print(f"Take one down and pass it around,
            {j} bottles of beer on the wall.")
        # Print an extra blank line
        print()
```

★ Loops problem set

Here is a simple problem set on loops in Python that you can try:

a) Write a for loop that prints all numbers from 1 to 10

```
for i in range(1, 11):
    print(i)
```


b) Write a for loop that prints all even numbers from 1 to 10

```
for i in range(1, 11):  
    if i % 2 == 0:  
        print(i)
```

c) Write a for loop that prints all numbers from 10 to 1 in reverse order

```
for i in range(10, 0, -1):  
    print(i)
```

d) Write a for loop that prints the sum of all numbers from 1 to 10

```
sum = 0  
for i in range(1, 11):  
    sum += i  
print(sum)
```

e) Write a for loop that prints the product of all numbers from 1 to 10

```
product = 1  
for i in range(1, 11):  
    product *= i  
print(product)
```

★ *Snake Eyes Codealong*

In this codealong, we're going to write a Python program that simulates the game of Snake Eyes, a simple dice game where the player rolls two dice and wins if the roll totals 2 (i.e. if both dice show a 1). First, let's import the random module, which we'll use to generate random numbers for the dice rolls:

```
import random
```

Next, let's write a **roll_dice()** function that simulates rolling a single die. This function should return a random number between 1 and 6 (inclusive), which represents the face of the die that is rolled:

```
def roll_dice():  
    return random.randint(1, 6)
```

Now, let's write the main part of our program. We'll start by creating a while loop that continues to run until the player wins (i.e., rolls a pair of 1s):

```
while True:
    # roll the dice
    die1 = roll_dice()
    die2 = roll_dice()

    # check if the player has won
    if die1 == 1 and die2 == 1:
        # player wins - break out of the loop
        break
```

Inside the loop, we roll the dice by calling the **roll_dice()** function twice and storing the results in the **die1** and **die2** variables. We then check if the player has won by using an if statement that checks if both dice show a 1. If this is the case, we break out of the loop using the **break** statement. Finally, here is the entire code:

```
import random

# Roll two dice
def roll():
    die1 = random.randint(1, 6)
    die2 = random.randint(1, 6)
    return die1, die2

# Play one round of "snake eyes"
def play_round():
    die1, die2 = roll()
    print(f"You rolled a {die1} and a {die2}")

    if die1 == 1 and die2 == 1:
        print("Snake eyes!")
        return True
    else:
        return False
```

```
# Play multiple rounds of "snake eyes" until the
player wins
def play_game():
    win = False
    rounds = 0

    while not win:
        win = play_round()
        rounds += 1

    print(f"It took you {rounds} rounds to roll
snake eyes.")

play_game()
```

This code defines a **roll()** function that simulates rolling two dice and returns their values as a tuple. The **play_round()** function uses the **roll()** function to roll two dice and checks if the result is snake eyes (i.e. if both dice show a 1). If it is snake eyes, the function returns **True**, otherwise, it returns **False**. The **play_game()** function plays multiple rounds of the game until the player rolls snake eyes, and then prints the number of rounds it took to win.

★ Dice Roller Exercise

Here is a simple dice-rolling game that you can try:

```
import random

while True:
    print("Rolling the dices...")
    dice1 = random.randint(1, 6)
    dice2 = random.randint(1, 6)

    print("The values are:")
    print(dice1)
    print(dice2)
```

```
play_again = input("Would you like to roll the  
dices again (yes/no)? ")  
if play_again.lower() != "yes":  
    break  
  
print("Thank you for playing!")
```

In this game, the **random** module is used to generate random numbers between 1 and 6, which represent the values of the two dices. The **while** loop will keep rolling the dice and printing the results until the player decides to stop by answering “no” to the prompt.

Functions

A **function** is a **block** of code that can be reusable and is designed to perform a specific task. **Functions** allow you to break down a larger program into smaller, more manageable pieces, which can make your code easier to understand and maintain. To define a function in Python, you use the `def` keyword followed by the name of the function and a set of parentheses. You can also specify one or more parameters inside the parentheses, which are variables that are passed to the function when it is called. For example:

```
def greet(name):  
    print("Hello, " + name)
```

This function, called `greet`, takes in a single parameter called `name` and prints a greeting message. To call the function, you can simply use its name followed by a set of parentheses, like this:

```
greet("Ion") # Output: "Hello, Ion"
```

Functions can also return a value using the `return` keyword. For example:

```
def add(a, b):  
    return a + b  
  
result = add(3, 4) # result is 7
```

This function, called `add`, takes in two parameters called **a** and **b** and returns their **sum**. Functions are a crucial part of any programming language and are widely used in data science and communication. They allow you to write modular, reusable code and structure your programs in a logical and organized way.

[hints!] Functions are good in data analysis and communication because they allow you to reuse code, which can save time and effort, and structure your programs in a logical and organized way. In data analysis, functions can be used to encapsulate complex calculations or data processing tasks, making it easier to understand and maintain your code. In addition, functions can improve the clarity and readability of your code by breaking it down into smaller, more manageable pieces. This can be especially helpful when working on large, complex data analysis projects, where clear and concise code is essential for effective collaboration and communication. Functions can also make it easier to test and debug your code, as you can isolate specific parts of your program and test them independently. This can help you identify and fix errors more efficiently, which can save time and improve the reliability of your data analysis. Functions are an important tool for data scientists and communication professionals, as they allow you to write efficient, modular, and readable code that can be used to analyze and communicate data effectively.

Our very first function!

To create your first **function** in Python, you can follow these steps: define the function using the **def** keyword, followed by the function name and a set of parentheses that may contain one or more parameters.

```
def my_function(param1, param2):
```

Indent the code block of the function, just like in loops and conditional statements. This is where you will write the code that the function will execute when it is called.

```
def my_function(param1, param2):  
    # Function code goes here
```

Write the code that the function will execute when it is called. This code can use the parameters passed to the function to perform some action or calculation.

```
def my_function(param1, param2):  
    result = param1 + param2  
    print(result)
```

After defining the function, you can call it by using the function name followed by the required parameters in parentheses.

```
my_function(1, 2)
```

This will call the **my_function** function and pass the values 1 and 2 as the **param1** and **param2** parameters, respectively. The function will then add these values together and print the result to the console.

Functions with an Input

The **input()** function in Python allows you to read input from the user. This can be useful when you want to ask the user for some information or give them the option to enter their data. Here is an example of a function that uses the **input()** function to read a string from the user and then prints it to the screen:

```
def print_input():  
    user_input = input("Enter a string: ")  
    print(user_input)
```

To call this function, you would simply use the function's name followed by any necessary arguments in parentheses:

```
print_input()
```

When you run this code, the function will prompt the user to enter a string, and then it will print the string to the screen. It's important to note that the **input()** function always returns a string, even if the user enters a number. If you want to read a number from the user and use it in your code, you will need to convert the string to a number using the **int()** or **float()** function. For example:

```
def print_input():  
    user_input = input("Enter a number: ")  
    # Convert the string to a float  
    number = float(user_input)  
    print(number)
```

[hint] Function “input” is good in data analysis and communication because it allows you to pass data or parameters to a function and customize its behavior. Function input allows you to write more flexible and reusable code that can be applied to various data and scenarios. In data analysis, the function “input” can be used to pass data from one part of your program to another, enabling you to perform calculations or transformations on the data. For instance, you could use the function “input” to pass a dataset to a function that calculates statistical measures or plots the data. By using the function “input”, you can write code that is more adaptable and can be used in a wide range of situations. Function “input” can also be used to specify configuration options or parameters for a function, allowing you to customize its behavior. For example, you might use the function input to specify the type of plot to generate or the data transformation to apply. Function “input” is an important tool for data scientists and communication professionals, as it allows you to write flexible, reusable code that can be customized to meet the needs of different data analysis tasks and scenarios. Here’s an example of how you might use function “input” in Python:

```
def calculate_mean(data: List[int]) -> float:
    """Calculates the mean of a list of integers."""
    return sum(data) / len(data)

data1 = [1, 2, 3, 4, 5]
data2 = [10, 20, 30, 40, 50]

mean1 = calculate_mean(data1)
mean2 = calculate_mean(data2)

print(f"Mean of data1: {mean1:.2f}") # Output:
Mean of data1: 3.00
print(f"Mean of data2: {mean2:.2f}") # Output:
Mean of data2: 30.00
```

In data analysis, function arguments can be used to pass data from one part of your program to another, enabling you to perform calculations or transformations on the data. Function arguments are a useful tool for writing flexible and reusable code in Python.

Functions with multiple arguments

In Python, you can define a function with multiple arguments by separating them with commas in the function definition. Here is an example of a function that takes two arguments, **x** and **y**, and returns the sum of the two values:

```
def add(x, y):
    return x + y
```

To call this function, you would use the function's name followed by the two arguments in parentheses, separated by a comma:

```
result = add(1, 2)
```

This would call the **add()** function with the arguments **1** and **2**, and the function would return the sum of **1** and **2**, which is **3**. After calling the function, you can save its output by assigning it to a variable or incorporating it into your program in some other way. It is crucial to remember that the function call's arguments must be in the same

order as the arguments specified in the function definition. In other words, in the **add()** function above, the first argument must be **x** and the second argument must be **y**.

Introducing Return!

The **return** keyword in Python is used to specify the value that a function should return. When a function encounters a return statement, it immediately stops executing and returns the specified value to the caller. Here is an example of a simple function that takes a single argument and returns the square of that value:

```
def square(x):  
    return x * x
```

To call this function, you would use the function's name followed by the argument in parentheses:

```
result = square(4)
```

This would call the **square()** function with argument 4, and the function would return the square of 4, which is **16**. You can then store the result of the function call in a variable or use it in some other way in your code. It's important to note that the return keyword is optional in Python. If you do not include a return statement in your function, the function will return **None** by default.

Using the Return keyword

The **return** keyword in Python is used to specify the value that a function should return. When a function encounters a **return** statement, it immediately stops executing and returns the specified value to the caller. Here is an example of a simple function that takes two arguments, **x** and **y**, and returns the sum of the two values:

```
def add(x, y):  
    return x + y
```

To call this function, you would use the function's name followed by the two arguments in parentheses, separated by a comma:

```
result = add(1, 2)
```

This would call the **add()** function with the arguments **1** and **2**, and the function would return the sum of **1** and **2**, which is **3**. You can then store the result of the function call in a variable or use it in some other way in your code. It's important to note that the **return** keyword is optional in Python. If you do not include a **return** statement in your function, the function will return **None** by default. Here is an example of a function that does not use the **return** keyword:

```
def print_sum(x, y):  
    print(x + y)
```

This function takes two arguments, **x** and **y**, and simply prints the sum of the two values to the screen. If you call this function and try to store the result in a variable, the variable will be assigned the value **None**, because the function does not return any value.

Default parameters

In Python, you can specify **default values** for function parameters, which allows you to call the function without providing all of the arguments. To specify a default value for a function parameter, you simply include an assignment operator (**=**) after the parameter name, followed by the default value. Here is an example of a function that takes two parameters, **x** and **y**, and has default values of **1** and **2** respectively:

```
def add(x=1, y=2):  
    return x + y
```

This function has two parameters, **x** and **y**, and both parameters have default values of **1** and **2**, respectively. If you call this function without providing any arguments, the default values will be used and the function will return **3** (the sum of **1** and **2**). However, if you provide one or more arguments when you call the function, the default values will be ignored and the provided values will be used instead. For example:

```
result = add(4)    # x=4, y=2 (default value)  
print(result)     # 6  
  
result = add(4, 5) # x=4, y=5  
print(result)     # 9
```

In the first call to **add()**, the function is called with a single argument of **4**, so the default value of **2** is used for the **y** parameter and the function returns **6** (the sum of **4** and **2**). In the second call to **add()**, the function is called with two arguments, **4** and **5**, so the default values are ignored and the function returns **9** (the sum of **4** and **5**).

Ordering default parameters

In Python, the order of the default parameters in a function definition does not matter. You can specify default values for parameters in any order, and you can also mix default and non-default parameters in any order. For example, the following two function definitions are equivalent:

```
def add(x=1, y=2):  
    return x + y  
  
def add(y=2, x=1):  
    return x + y
```

Both of these function definitions have two parameters, **x** and **y**, with default values of **1** and **2**, respectively. If you call the **add()** function without providing any arguments, both functions will return the same result (**3**, the sum of **1** and **2**). You can also mix default and non-default parameters in any order. For example, the following function definition is also equivalent to the ones above:

```
def add(y, x=1):  
    return x + y
```

This function definition has two parameters, **x** and **y**, with a default value of **1** for **x** and no default value for **y**. If you call the **add()** function without providing any arguments, it will raise an error because the **y** parameter does not have a default value. However, if you call the function with a single argument, the argument will be used as the value for **y** and the default value of **1** will be used for **x**, so the function will return the same result as the previous examples.

KeywordNamed argument

In Python, you can use **keyword-named** arguments to call a function. This means that you can specify the arguments by name instead of by position in the function call. This technique can make your code easier to read when you have a function with many arguments. To use **keyword-named arguments**, you simply provide the argument name followed by an equals sign (=) and the argument value when you call the function. For example, consider the following function definition:

```
def add(x, y):  
    return x + y
```

This function has two parameters, **x** and **y**, and it returns the sum of the two values. To call this function using **keyword-named arguments**, you would use the following syntax:

```
result = add(x=1, y=2)  
print(result) # 3
```

In this example, the function is called with the **keyword-named arguments** **x=1** and **y=2**, which specify the values for the **x** and **y** parameters, respectively. The function call is equivalent to calling the function with the arguments **1** and **2** in the usual way.

[hints!] Keyword-named arguments can be useful when you have a function with many parameters, and you want to make the code more readable by explicitly specifying the argument names. For example, consider the following function definition:

```
def calculate_cost(price, tax, discount):  
    return price + (price * tax) - (price * discount)
```

This function calculates the total cost of an item based on the price, tax rate, and discount rate. To call this function using keyword-named arguments, you could use the following code:

```
result = calculate_cost(price=100, tax=0.05,  
                        discount=0.10)  
print(result) # 95.0
```

In this example, the function is called with the keyword-named arguments `price=100`, `tax=0.05`, and `discount=0.10`, which specify the values for the price, tax, and discount parameters, respectively. The function call is equivalent to calling the function with the arguments 100, 0.05, and 0.10. The keyword-named arguments provide a way to specify the values of the parameters in a more explicit and readable way. This can make the code easier to understand, especially when there are many parameters or when the arguments are long or complex. `

★ *Function practice set*

Here are a few examples of function practice sets that you can use to improve your skills in Python.

- a) Write a function that takes two numbers as arguments and returns the sum of the two numbers.

```
def add(x, y):  
    return x + y  
  
result = add(1, 2)  
print(result)  # 3
```

- b) Write a function that takes a string as an argument and returns the string with all vowels (a, e, i, o, u) replaced with an underscore (_).

```
def replace_vowels(s):  
    vowels = "aeiou"  
    for vowel in vowels:  
        s = s.replace(vowel, "_")  
    return s  
  
result = replace_vowels("Hello, World!")  
print(result)  # H_ll_, W_rld!
```

- c) Write a function that takes a list of numbers as an argument and returns the average (mean) of the numbers in the list.

```
def average(numbers):  
    return sum(numbers) / len(numbers)  
  
result = average([1, 2, 3, 4, 5])  
print(result) # 3.0
```

Global Scope

In programming, the **global scope** refers to the visibility or accessibility of variables and other data within a program. Variables that are declared at the global level, outside of any function or class, are said to have global scope and can be accessed from anywhere in the program. These global variables can be accessed from anywhere in the program, and are not restricted to a specific function or class. Here is an example of global scope in Python:

```
# define a global variable  
x = 5  
  
def my_function():  
    # access the global variable within the function  
    print(x)  
  
my_function() # prints 5
```

In this example, the **x** variable is defined at the global level, outside of any function. It is then accessed from within the **my_function()** function, where it retains its global value of **5**.

[hints!] In Python, variables that are defined outside of any function or class are said to be in the global scope, which means they can be accessed from anywhere in the program. The global scope can be good in data analysis and communication because it allows you to access and use variables from any part of your program. This can be useful if you have data or values that you want to use in multiple functions or across different

modules of your program. For example, you might define a global variable that stores the data for a particular analysis, and then use that data in multiple functions or modules to perform different calculations or transformations. This can save you the effort of having to pass the data around between different parts of your program and can help you write more efficient and organized code. However, it's important to use global scope responsibly, as it can also lead to conflicts and unintended consequences if you are not careful. In general, it's a good practice to use global scope sparingly, and to use a local scope (variables defined within a function or class) whenever possible, to avoid confusion and improve the readability and maintainability of your code.

Local Scope

The local scope refers to the visibility or accessibility of variables and other data within a specific block of code, such as a function or class. Variables that are declared within a function or class, or within a block of code within a function or class, are said to have local scope, and can only be accessed from within that block of code. Here is an example of local scope in Python:

```
def my_function():  
    # define a local variable within the function  
    x = 10  
    print(x)  
  
my_function() # prints 10  
print(x) # causes an error, x is not defined at  
the global level
```

In this example, the `x` variable is defined within the `my_function()` function, so it has local scope and can only be accessed from within that function. If we try to access the `x` variable from outside the function, an error is raised because the variable is not defined at the global level.

Scope in loops and conditionals

The scope of a variable is determined by the block of code in which it is defined. This means that the same variable name can be used in different blocks of code, and each instance of the variable will have its local scope. For example, the scope of a variable defined within a loop or conditional statement will be limited to the block of code within that loop or conditional. This can be seen in the following example:

```
# define a global variable
x = 5

if x > 0:
    # define a local variable with the same name
    within the if block
    x = 10
    print(x)  # prints 10

print(x)  # prints 5
```

In this example, the **x** variable is defined at the global level with a value of **5**. Within the **if** block, a new local variable with the same name is defined and assigned a value of **10**. The local variable only exists within the **if** block, so when we try to access it outside the block, the global variable is used instead.

Enclosing Scope

The **enclosing scope** is the innermost block of code that contains a particular variable or function. This is relevant when working with nested functions, as a nested function can access variables in the enclosing scope in which it is defined. Here is an example of the enclosing scope in Python:

```
def outer_function():
    x = 5  # x is defined in the enclosing scope of
    the inner function

    def inner_function():
        print(x)  # accesses the x variable from the
        enclosing scope
```



```
    inner_function()

outer_function()  # prints 5
```

In this example, the **inner_function()** is defined within the **outer_function()**, so it has access to the **x** variable defined in the **enclosing scope** of the **outer_function()**. When we call the **inner_function()**, it prints the value of **x** from the **enclosing scope**.

Built-in Scope

The **built-in scope** refers to the built-in functions and variables that are available in every Python program. These built-ins are defined by the Python interpreter and are always available for use in a Python program. Some examples of built-in functions in Python include **len()**, **print()**, and **range()**. Some examples of built-in variables include **True**, **False**, and **None**. The **built-in scope** can be accessed from anywhere in a Python program, and its functions and variables do not need to be imported or declared to be used. However, if a variable or function is defined with the same name as a built-in, the local variable or function will take precedence over the built-in. Here is an example of accessing a built-in function in Python:

```
# use the built-in len() function to get the
length of a string
my_string = "Hello, World!"
string_length = len(my_string)

print(string_length)  # prints 13
```

In this example, we use the built-in **len()** function to get the length of the **my_string** variable and then print the result. Since **len()** is a built-in function, we do not need to import it or declare it before using it in our program.

Scope precedence rules

When a variable or function is accessed from within a block of code, the interpreter follows certain rules to determine which instance of the variable or function to use. These rules, known as scope precedence rules, dictate the order in which the interpreter looks for the variable or function in different scopes. Here is the order of precedence for the different scopes in Python, from highest to lowest:

- 1. Local scope**
- 2. Enclosing scope**
- 3. Global scope**
- 4. Built-in scope**

This means that, when accessing a variable or function, the interpreter will first look for the variable or function in the local scope (i.e., within the current block of code). If it is not found, it will look in the enclosed scope (i.e. the innermost block of code that contains the current block). If it is still not found, it will look in the global scope (i.e. the top-level of the script). Finally, if it is still not found, it will look in the built-in scope. Here is an example of the scope precedence rules in action:

```
x = 5  # x is defined in the global scope

def my_function():
    x = 10  # x is defined in the local scope of the
    function
    print(x)  # prints 10, uses the local x

my_function()
print(x)  # prints 5, uses the global x
```

In this example, the **x** variable is defined at the global level with a value of **5**. Within the **my_function()** function, a new local variable with the same name is defined and assigned a value of **10**. When we call the **my_function()** and print the value of **x** within the function, it uses the local **x** variable with a value of **10**, because it has higher precedence than the global **x** variable. However, when we print the value of **x** outside the function, it uses the global **x** variable with a value of **5**, because the local **x** variable is not defined outside the function.

The ‘Global’ keyword

The **global keyword** is used to indicate that a variable is global and should be accessed from the global scope. This is necessary when a local variable with the same name as a global variable is defined within a function. Without the **global keyword**, the local variable would take precedence over the global variable, and the global variable would be inaccessible within the function. Here is an example of using the **global keyword** in Python:

```
x = 5  # x is defined in the global scope

def my_function():
    global x
    x = 10  # x is now a global variable, with a
value of 10
    print(x)  # prints 10

my_function()
print(x)  # also prints 10
```

In this example, the **x** variable is defined at the global level with a value of **5**. Within the **my_function()** function, we use the **global keyword** to indicate that the **x** variable is a global variable, and then reassign it a value of **10**. When we print the value of **x** within the function, it uses the **global x** variable with a value of **10**, because we have indicated that it is a global variable. When we print the value of **x** outside the function, it also uses the **global x** variable with a value of **10**, because the local **x** variable is not defined outside the function. It is important to use the **global keyword** carefully, as it can lead to confusing and difficult-to-debug code if used improperly. In general, it is best to avoid using global variables whenever possible and to use local variables instead.

★ *Exercises for understanding the Global Scope*

Here are a few exercises you can try to help you understand the global scope of Python:

- a) Define a global variable in your program and assign it a value. Then, write a function that prints the value of the global variable. Call the function to see if it prints the correct value.
- b) Write a function that takes an argument and assigns it to a global variable. Then, write a second function that prints the value of the global variable. Call the second function to see if it prints the correct value.
- c) Define a global variable with the same name as a local variable in a function. Inside the function, print the value of both the global and local variables. What happens?
- d) Write a function that takes an argument and assigns it to a local variable with the same name as a global variable. Inside the function, print the value of both the global and local variables. What happens?
- e) Write a function that takes an argument and assigns it to a global variable. Then, write a second function that modifies the value of the global variable. Call the second function and then print the value of the global variable. What happens? (Jindal, A., & all, 2021).

These exercises will help you understand how global and local variables interact with each other, and how the global scope works in Python.

Lists the basics

In Python, a list is a data type that is used to store an ordered collection of elements. Lists are similar to arrays in other programming languages but have some key differences and additional features. To create a list in Python, use square brackets `[]` to enclose a comma-separated list of elements. For example:

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

This creates a list called `my_list` that contains the elements **1**, **2**, **3**, **4**, and **5**. Lists can contain elements of any data type, including other lists. For example:

```
my_list = [1, 2, 3, "hello", ["a", "b", "c"]]
```

This creates a list that contains the elements **1**, **2**, **3**, the string “**hello**”, and a list of the letters “**a**”, “**b**”, and “**c**”. To access an element of a list, use its index (the position of the element in the list) within square brackets. For example:

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

# access the first element of the list
first_element = my_list[0]

# access the third element of the list
third_element = my_list[2]
```

In Python, lists are **0**-indexed, meaning that the first element has an index of **0**, the second element has an index of **1**, and so on. To modify or add elements to a list, simply use the element’s index within square brackets and assign it a new value. For instance:

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

# change the value of the second element
my_list[1] = 10

# add a new element to the end of the list
my_list.append(6)
```

These examples change the value of the second element of `my_list` to **10** and add the element **6** to the end of the list.

[hints!] Lists are good in data analysis and communication because they allow you to store and manipulate collections of data flexibly and efficiently. In Python, a list is an ordered collection of items that can be of any data type, including numbers, strings, and other data structures. In data analysis, lists can be used to store and organize large datasets, perform calcula-

tions or transformations on the data, and then present the results clearly and concisely. For example, you might use a list to store a set of numerical data and then use built-in functions like `sum`, `mean`, and `median` to calculate statistical measures, or you might use a list comprehension to filter and transform the data in a single line of code. Lists are also useful in communication because they allow you to present data in a clear and organized way. For example, you might use a list to display the results of data analysis in a table or graph or to provide a list of options for users to choose from. Lists are an important tool for data scientists and communication professionals, as they allow you to store and manipulate collections of data flexibly and efficiently, and to present the results of your analysis in a clear and organized manner.

Accessing data in lists

You can access the elements in a list using their indices, which is the position of the element in the list. For example, if you have a list called **my_list**, you can access the first element in the list by using **my_list[0]**. Here's an example:

```
my_list = [10, 20, 30, 40]

# Access the first element in the list
first_element = my_list[0]

# Access the second element in the list
second_element = my_list[1]
```

In Python, the index of the first element in a list is **0**, not **1**. So, to access the second element in the list, you would use index **1**, not **2**. You can also use negative indices to access elements in a list. For example, if you have a list with five elements and you want to access the last element in the list, you can use index **-1**.

Updating list elements

In Python, you can alter an item in a list by using the index operator which grants access to a specific element in the list based on its position. For instance, to change the first element in the list called “**my_list**”, you can do the following:

```
my_list[0] = "new value"
```

You can also use the **.insert()** or **.append()** methods to add new elements to a list, or the **.remove()** or **.pop()** methods to remove elements from a list. Here is an example that demonstrates these methods:

```
# create a list
my_list = ["apple", "banana", "cherry"]

# add a new element to the end of the list
my_list.append("mango")

# add a new element at a specific position in the list
my_list.insert(1, "orange")

# remove an element from the list
my_list.remove("banana")

# remove an element at a specific position in the list
my_list.pop(1)
```

Remember that the index of the first element in a list is **0**, not **1**. So, to update the second element in a list, you would use index **1**, not **2**.

Append() and Extend()

To **append** a single element to the end of a list, the **.append()** method is utilized. To add multiple elements to the end of a list, the **.extend()** method is used. The following example shows the use of these methods:

```
# create a list
my_list = ["apple", "banana", "cherry"]
```

```
# add a single element to the end of the list
my_list.append("mango")

# add multiple elements to the end of the list
my_list.extend(["orange", "grape", "pear"])
```

As demonstrated, the `.append()` method requires a single argument, which is the element to be appended to the end of the list. On the other hand, the `.extend()` method requires an iterable object like a list as an argument and adds each element of the iterable to the end of the list. One important thing to note is that the `.extend()` method modifies the list in place, whereas the `.append()` method returns a new list with the element added. This means that if you want to save the new list after using the `.append()` method, you need to assign the returned list to a variable. For example:

```
# create a list
my_list = ["apple", "banana", "cherry"]

# add a single element to the end of the list
my_list = my_list.append("mango")
```

In this case, the `.append()` method returns a new list with the element “mango” added, and this new list is assigned to the `my_list` variable.

Insert()

The `.insert()` method is used to add a new element at a specific position in a list. This method takes two arguments: the index of the position where the element should be inserted, and the element to be inserted. Here is an example that demonstrates how to use the `.insert()` method:

```
# create a list
my_list = ["apple", "banana", "cherry"]

# add a new element at the beginning of the list
my_list.insert(0, "mango")

# add a new element at the second position in the list
my_list.insert(1, "orange")
```


The `.insert()` method allows you to specify the location where a new element should be placed in the list. This is different from the `.append()` method, which always adds the element to the end of the list. It is important to note that the first element in a list has an index of `0`, not `1`. To insert an element at the start of the list, the index `0` should be used. It is worth noting that the `.insert()` method modifies the list directly and does not require assignment to a variable. However, if you want to keep the modified list, you can assign the result of the `.insert()` method to a variable. For example:

```
# create a list
my_list = ["apple", "banana", "cherry"]

# add a new element at the beginning of the list
my_list = my_list.insert(0, "mango")
```

In this case, the `.insert()` method returns a new list with the element “mango” added at the beginning, and this new list is assigned to the `my_list` variable.

List Slices

A **slice** is a subset of a list. You can use slices to extract specific elements from a list or to modify multiple elements in a list at once. To create a **slice**, the index of the first element to be included in the slice is specified, followed by a colon (:), and then the index of the first element to be excluded from the slice (VanderPlas, J. 2016). For instance, if you have a list called “`my_list`”, you can create a **slice** containing the first three elements of the list as follows:

```
my_slice = my_list[0:3]
```

In this example, the slice “`my_slice`” will include the elements at indexes `0`, `1`, and `2` in the “`my_list`” list, but not the element at index `3`. If you want to include all elements in the list up to a specific index, the first index can be omitted. For instance, to create a slice that includes the first three elements of the “`my_list`” list, the following syntax can be used:

```
my_slice = my_list[:3]
```

This is the same as the earlier example and the “**my_slice**” slice will contain the same three elements. Additionally, if you want to include all elements in the list starting from a specific index, the second index can be left out. For instance, to create a slice that includes all elements in the “**my_list**” list starting from the second element, the following syntax can be used:

```
my_slice = my_list[1:]
This slice will include the elements at indexes 1,
2, 3, and so on, until the end of the my_list
list.
```

Slices can also be created using negative indexes. A negative index counts backward from the end of the list, so for a list with five elements, the index **-1** refers to the last element of the list, the index **-2** refers to the second to last element of the list, and so on. For instance, if you have a list “**my_list**” = [1, 2, 3, 4, 5], the slice “**my_list[-3:-1]**” would return the sublist [3, 4]. Negative indexes are useful when you want to create a slice that includes the final elements of a list without knowing the length of the list.

Deletion methods pop(), popitem(), remove()

There are several methods you can use to remove elements from a list in Python. The **pop()** method removes the element at a specified index from the list and returns the removed element. The **popitem()** method removes and returns an arbitrary (key, value) pair from a dictionary. The **remove()** method removes the first occurrence of a specified value from a list. Here is an example of how you can use these methods:

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

# Remove the element at index 2 (the third ele-
ment) and print it
print(my_list.pop(2)) # Output: 3

# Remove and return an arbitrary (key, value) pair
from a dictionary
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict.popitem()) # Output: ('c', 3)
```

```
# Remove the first occurrence of the value 2 from  
the list  
my_list.remove(2)  
print(my_list) # Output: [1, 4, 5]
```

It is important to keep in mind that if the list is empty or if the specified index or value does not exist in the list, the **pop()** and **remove()** methods will produce an error. To verify if an element exists in a list before calling these methods, the **in** operator can be used.

Iterating over lists

To **iterate** over the elements of a list, you can use a for a loop. **Iterating over lists** is an important concept in Python because it allows you to perform a specific action on each element of a list. This is useful when you have a large amount of data that you need to process or when you want to perform an operation on each element of a list and create a new list with the results. For example, you might want to create a new list that contains the squares of the elements of an existing list, or you might want to count the number of elements in a list that meet a certain condition. Iterating over lists allows you to do these types of operations efficiently and with minimal code. Here is an example:

```
my_list = [1, 2, 3, 4, 5]  
  
# Iterate over the elements of the list and print  
each element  
for element in my_list:  
    print(element)  
  
# Output:  
# 1  
# 2  
# 3  
# 4  
# 5
```

You can also use the **enumerate()** function to iterate over a list and get the index of each element at the same time. Here is an example:

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

# Iterate over the elements of the list and print
# each element along with its index
for index, element in enumerate(my_list):
    print(f'{index}: {element}')

# Output:
# 0: 1
# 1: 2
# 2: 3
# 3: 4
# 4: 5
```

You can use the **range()** function to iterate over a list and get the index of each element without using the **enumerate()** function. Here is an example:

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

# Iterate over the elements of the list and print
# each element along with its index
for index in range(len(my_list)):
    print(f'{index}: {my_list[index]}')

# Output:
# 0: 1
# 1: 2
# 2: 3
# 3: 4
# 4: 5
```

Source: Petrelli, M. (2021)

Remember that you can use negative indexes to access elements from the end of the list, so you can also use a negative range in the **range()** function to iterate over the elements of a list in reverse order.

Lists + loops patterns

There are many common patterns that you can use when working with lists and loops in Python. Here are a few examples:

Iterating over a list of elements: This is the most basic pattern for working with lists and loops. It involves using a for loop to iterate over each element in a list and perform some operation on each element.

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Iterate over the list of numbers
for number in numbers:
    print(number)
```

Iterating over a list of elements and their indexes: Sometimes you may need to keep track of the index of the current element while iterating over a list. This can be done using the built-in **enumerate()** function, which returns a tuple containing the index and the value of each element in the list.

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Iterate over the list of numbers and their indexes
for i, number in enumerate(numbers):
    print(f'The element at index {i} is {number}')
```

Iterating over a list in reverse order: As mentioned earlier, you can use the **range()** function to iterate over a list in reverse order by specifying a negative step size.

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Iterate over the list in reverse order
for i in range(len(numbers)-1, -1, -1):
    print(numbers[i])
```

Iterating over multiple lists simultaneously: Sometimes you may need to iterate over multiple lists simultaneously. This can be done using the built-in **zip()** function, which combines the elements of multiple lists into tuples and returns an iterator that yields the tuples. Here is an example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

for item1, item2 in zip(list1, list2):
    print(item1, item2)
```

This will print the following:

```
1 4
2 5
3 6
```

The **zip** function creates an iterator that combines the elements of both lists into tuples. The **for** loop then iterates over these tuples, unpacking the elements into the **item1** and **item2** variables. If the lists are of different lengths, the iterator will stop when it reaches the end of the shorter list. For example, if we change **list1** to be **[1, 2, 3, 4]** and leave **list2** as it is, the code above will only print the following:

```
1 4
2 5
3 6
```

You can also use the **zip** function to iterate over more than two lists simultaneously. Simply pass all of the lists to the **zip** function, and unpack the resulting tuples in the **for** a loop. Here is an example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

for item1, item2, item3 in zip(list1, list2,
list3):
    print(item1, item2, item3)
```

This will print the following:

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

As before, if the lists are of different lengths, the iterator will stop when it reaches the end of the shortest list.

Nested lists

A nested list is a list within a list. Here is an example of a nested list:

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

This is a list of three elements, where each element is itself a list containing three integers. You can access elements in a nested list using multiple indices. For example, to access the number **6** from the above list, you would use the following code:

```
nested_list[1][2]
```

This would return the second element (the list **[4, 5, 6]**) from the outer list, and then return the third element (the number **6**) from that inner list. You can also use negative indices to access elements in a nested list. For example, the following code would return the number **8**:

```
nested_list[-2][-1]
```

This returns the second-to-last element (the list **[7, 8, 9]**) from the outer list, and then returns the last element (the number **8**) from that inner list. You can also modify elements in a nested list using multiple indices. For example, the following code would change the number **3** in the first inner list to the number **10**:

```
nested_list[0][2] = 10
```

After running this code, the nested list would be changed to the following:

```
[[1, 2, 10], [4, 5, 6], [7, 8, 9]]
```

Nested lists are useful for representing data that has a hierarchical structure, such as the data in a tree. They can also be useful for organizing data in a way that makes it easy to access and modify.

List operators

In Python, several operators can be used with lists. These include the following:

- The **+** operator concatenates two lists, creating a new list that contains all the elements from both lists.
- The ***** operator repeats a list a given number of times, creating a new list that contains the original list repeated the specified number of times.
- The **in** keyword checks if a given element is contained in a list, and returns a boolean value (**True** or **False**) depending on the result.
- The **not in** keyword checks if a given element is not contained in a list, and returns a boolean value depending on the result.

Here are some examples of these operators in action:

```
# Concatenate two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1 + list2
print(list3)  # Output: [1, 2, 3, 4, 5, 6]

# Repeat a list
list1 = [1, 2, 3]
list2 = list1 * 3
print(list2)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

# Check if an element is in a list
list1 = [1, 2, 3]
print(2 in list1)  # Output: True
print(4 in list1)  # Output: False

# Check if an element is not in a list
list1 = [1, 2, 3]
print(2 not in list1)  # Output: False
print(4 not in list1)  # Output: True
```


Sort(), Reverse(), and Count()

The **sort()** method is used to sort a list in ascending order by default. It has the following syntax:

```
list.sort(key=None, reverse=False)
```

The **key** parameter specifies a function that will be called on each element of the list to determine the value that will be used for sorting. The **reverse** parameter is a boolean value that specifies whether the list should be sorted in ascending or descending order. The **reverse()** method is used to reverse the order of the elements in a list. It has the following syntax:

```
list.reverse()
```

The **count()** method is used to count the number of times a specific element appears in a list. It has the following syntax:

```
list.count(element)
```

Here are some examples of using these methods:

```
# Sort a list of numbers in ascending order
numbers = [3, 1, 5, 8, 2]
numbers.sort()
print(numbers) # Output: [1, 2, 3, 5, 8]

# Sort a list of numbers in descending order
numbers = [3, 1, 5, 8, 2]
numbers.sort(reverse=True)
print(numbers) # Output: [8, 5, 3, 2, 1]

# Sort a list of strings by their length
words = ['apple', 'pear', 'banana', 'cherry']
words.sort(key=len)
print(words) # Output: ['pear', 'apple', 'cherry', 'banana']

# Reverse the order of the elements in a list
numbers = [3, 1, 5, 8, 2]
numbers.reverse()
print(numbers) # Output: [2, 8, 5, 1, 3]
```

Lists are mutable

In programming, a **list is a collection of items** that can be stored, accessed, and modified. One of the key features of a list is that it is mutable, which means that its contents can be changed. This is in contrast to other data types, such as strings and tuples, which are immutable and cannot be modified once they have been created. Being able to change the contents of a list is useful because it allows you to add, remove, or modify items in the list as needed. This makes lists a very powerful and flexible data type that can be used in a wide range of applications. To illustrate how lists are mutable, consider the following example in Python:

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]

# Print the list
print(numbers)

# Add a new number to the list
numbers.append(6)

# Print the updated list
print(numbers)

# Remove an item from the list
numbers.remove(4)

# Print the updated list
print(numbers)

# Change the value of an item in the list
numbers[2] = 100

# Print the updated list
print(numbers)
```

In this example, we create a list of numbers and then use various list methods to add, remove, and modify items in the list. As you can see, the list is mutable and can be changed as needed.

Comparing lists == vs is

In Python, there are two ways to compare two lists: using the `==` operator and the `is` keyword. The `==` operator checks if the two lists have the same contents. This means that the lists are considered equal if they have the same number of items and the items in each list are in the same order and have the same values. For example:

```
# Create two lists
list1 = [1, 2, 3]
list2 = [1, 2, 3]

# Check if the lists are equal
print(list1 == list2) # Output: True
```

In this example, the `==` operator returns `True` because the two lists have the same contents. On the other hand, the `is` keyword checks if the two lists are the same object. This means that the lists are considered equal only if they refer to the same object in memory. For example:

```
# Create two lists
list1 = [1, 2, 3]
list2 = [1, 2, 3]

# Check if the lists are the same object
print(list1 is list2) # Output: False
```

In this example, the `is` keyword returns `False` because the two lists are not the same object. Even though they have the same contents, they are stored in different locations in memory, so they are different objects. In general, it is recommended to use the `==` operator to compare the contents of two lists, and the `is` keyword to check if two variables refer to the same object.

Join() and Split()

The `join()` and `split()` methods are string methods in Python that are used to manipulate strings. The `join()` method is used to concatenate a list of strings into a single string. It takes a list of strings as its argument and returns a string that consists of the items in the list joined together, with a specified separator string between each item. For example:

```
# Define a list of strings
words = ["Hello", "World", "!"]

# Join the list of strings into a single string
sentence = " ".join(words)

# Print the resulting string
print(sentence) # Output: "Hello World !"
```

In this example, the **join()** method is used to concatenate the list of strings into a single string, with a space character as the separator. The **split()** method is the opposite of the **join()** method. It is used to split a string into a list of strings, using a specified separator string. For example:

```
# Define a string
sentence = "Hello World !"

# Split the string into a list of strings
words = sentence.split(" ")

# Print the resulting list
print(words) # Output: ["Hello", "World", "!"]
```

In this case, the string is split into a list of strings using the **split()** method and the space character as the separator. To summarize, the **join()** method is used to combine a list of strings into a single string, whereas the **split()** method is used to divide a string into a list of strings. These methods can be useful for manipulating and working with strings in your code.

List unpacking

List unpacking is a technique that allows you to assign the elements of a list to individual variables. This is useful when you want to access or manipulate items in a list individually, rather than working with the entire list. To unpack a list, you place the list on the left side of an assignment statement and the individual elements of the list on the right side, separated by commas (Verma, O. P., 2019). For example:

```
# Define a list of numbers
numbers = [1, 2, 3]

# Unpack the list into individual variables
a, b, c = numbers

# Print the individual variables
print(a) # Output: 1
print(b) # Output: 2
print(c) # Output: 3
```

In this example, the list of numbers is unpacked into three separate variables, **a**, **b**, and **c**. This allows us to access and manipulate the individual items in the list separately. List unpacking can also be used to exchange the values of multiple variables in a single statement. For example:

```
# Define two variables
a = 1
b = 2

# Use list unpacking to exchange the values of the variables
a, b = b, a

# Print the new values of the variables
print(a) # Output: 2
print(b) # Output: 1
```

In this example, the values of the variables **a** and **b** are exchanged using list unpacking. This is a simple and elegant way to swap the values of two variables without using a temporary variable. In conclusion, **list unpacking** is a useful Python feature that enables you to assign the elements of a list to individual variables, which can be used to manipulate and work with the items in a list more easily and efficiently.

Copying lists

In Python, there are several ways to **copy a list**, depending on the desired behavior and the specific requirements of your code. One way to copy a list is to use the **copy()** method. This method creates a shal-

low copy of the list, which means that it creates a new list object with the same elements as the original list, but the new list and the original list are not connected. For example:

```
# Define a list
numbers = [1, 2, 3]

# Create a shallow copy of the list
numbers_copy = numbers.copy()

# Modify the original list
numbers.append(4)

# Print the original and copied lists
print(numbers) # Output: [1, 2, 3, 4]
print(numbers_copy) # Output: [1, 2, 3]
```

In this case, the `copy()` method is utilized to create a shallow copy of the list called “**numbers**”. Even if the original list is altered, the copied list remains unchanged. Another method for copying a list is by using the `list()` function. This function creates a new list object that is initialized with the elements of the original list. For example:

```
# Define a list
numbers = [1, 2, 3]

# Create a new list initialized with the elements
of the original list
numbers_copy = list(numbers)

# Modify the original list
numbers.append(4)

# Print the original and copied lists
print(numbers) # Output: [1, 2, 3, 4]
print(numbers_copy) # Output: [1, 2, 3]
```

In this example, the `list()` function is used to create a new list object that is initialized with the elements of the original list. As with the previous example, the copied list is not affected by any modifications to the original list.

★ *Exercises with lists*

Here are a few exercises that you can try with lists in Python:

- a) Create a list of numbers from 1 to 10, and print each element of the list.
- b) Create a list of colors, and print each color in the list using a for a loop.
- c) Use the range function to create a list of even numbers from 2 to 10, and print each element of the list.
- d) Use the append method to add an element to the end of a list, and print the resulting list.
- e) Use the insert method to add an element to the beginning of a list, and print the resulting list.

Here is some sample code that you can use as a starting point for these exercises:

```
# Exercise 1
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for number in numbers:
    print(number)

# Exercise 2
colors = ["red", "green", "blue", "yellow"]
for color in colors:
    print(color)

# Exercise 3
even_numbers = list(range(2, 11, 2))
for number in even_numbers:
    print(number)

# Exercise 4
numbers = [1, 2, 3, 4]
numbers.append(5)
print(numbers)

# Exercise 5
numbers = [1, 2, 3, 4]
numbers.insert(0, 0)
print(numbers)
```

★ *Todo list exercise intro*

To create a to-do list application in Python, you can use a list data structure to store the items on the to-do list. You can then add functionality to your application by defining functions for adding items, marking items as complete, and viewing the current to-do list. Here is an example of how you could implement a to-do list in Python:

```
# create an empty list to store the to-do items
todo_list = []

# define a function for adding items to the to-do list
def add_item(item):
    todo_list.append(item)

# define a function for marking items as completed
def mark_as_done(index):
    todo_list[index] = "DONE: " + todo_list[index]

# define a function for viewing the current to-do list
def view_list():
    for item in todo_list:
        print(item)

# add some items to the to-do list
add_item("Get groceries")
add_item("Finish homework")
add_item("Take out the trash")

# view the current to-do list
view_list()

# mark the first item as completed
mark_as_done(0)

# view the updated to-do list
view_list()
```


In this example, we first create an empty list to store the to-do items. We then define three functions: **add_item()** for adding items to the to-do list, **mark_as_done()** for marking items as completed, and **view_list()** for viewing the current to-do list. We use the **append()** method to add items to the `todo_list`, and we update the item at the specified index by prefixing it with “DONE:”. Finally, we add some items to the to-do list and view the list before and after marking an item as completed.

Dictionaries

In Python, a **dictionary** is a collection of **key-value pairs**. Each key is linked to a value and a key can be used to retrieve the corresponding value. **Dictionaries** are like lists, but instead of using an integer index to access items, a key is used to access the value associated with that key. An example of using a **dictionary** in Python is shown below:

```
# create an empty dictionary
d = {}

# add some key-value pairs to the dictionary
d["apple"] = "A fruit that is red or green in color."
d["dog"] = "A common household pet that is often loyal and affectionate."

# access values in the dictionary using the corresponding keys
print(d["apple"]) # prints "A fruit that is red or green in color."
print(d["dog"]) # prints "A common household pet that is often loyal and affectionate."
```

In this example, we create an empty **dictionary** and then add some **key-value pairs** to it. We then use the keys to access the associated values and print them on the screen. **Dictionaries** are useful for storing data when you need to access the values using a specific key. For example, you could use a **dictionary** to store information about a set of employees, where the keys are the employee IDs and the values are

the employee names and job titles. This way, you can easily look up an employee's name and job title using their ID.

[hints!] Dictionaries are good in data analysis and communication because they allow you to store and manipulate data flexibly and efficiently. In Python, a dictionary is a collection of key-value pairs, where each key is associated with a value. Dictionaries are similar to lists, but they are indexed by keys rather than by numeric positions, which makes them more flexible and efficient for certain tasks. In data analysis, dictionaries can be used to store and organize large datasets, perform calculations or transformations on the data, and then present the results clearly and concisely. For example, you might use a dictionary to store a set of data points, where the keys represent the data categories, and the values represent the data values. You could then use the keys to access and manipulate the data flexibly and efficiently. Dictionaries are also useful in communication because they allow you to present data in a clear and organized way. For example, you might use a dictionary to display the results of data analysis in a table or graph or to provide a list of options for users to choose from. Dictionaries are an important tool for data scientists and communication professionals, as they allow you to store and manipulate data flexibly and efficiently, and to present the results of your analysis in a clear and organized manner.

Creating your Dictionaries

To create a dictionary in Python, you can use the **dict()** constructor or the curly braces **{}**. For example, you can create an empty dictionary using either of the following methods:

```
# using the dict() constructor
d = dict()

# using curly braces
d = {}
```

You can also create a dictionary with initial **key-value** pairs by providing a list of comma-separated **key-value** pairs enclosed in curly braces. For example:

```
# create a dictionary with initial key-value pairs
d = {"apple": "A fruit that is red or green in color.", "dog": "A common household pet that is often loyal and affectionate."}
```

Once you have created a dictionary, you can add new key-value pairs using the square bracket notation and the assignment operator `=`. For example:

```
# create an empty dictionary
d = {}

# add some key-value pairs
d["apple"] = "A fruit that is red or green in color."
d["dog"] = "A common household pet that is often loyal and affectionate."
```

You can also update the value associated with a given key by using the square bracket notation and the assignment operator `=`. For example:

```
# create a dictionary with initial key-value pairs
d = {"apple": "A fruit that is red or green in color.", "dog": "A common household pet that is often loyal and affectionate."}

# update the value associated with the "apple" key
d["apple"] = "A tasty fruit that can be red, green, or yellow in color."
```

Keep in mind that dictionaries in Python are unordered, meaning that the items in a **dictionary** are not stored in a specific order. This means that the order in which you add items to a dictionary may not be the same order in which they are stored or accessed.

Accessing data in Dictionaries

A dictionary is a collection of **key-value pairs**. To access the data in a dictionary, you use the keys to retrieve the values. Here's an example:

```
# Create a dictionary
my_dict = {
    "name": "Ion",
    "age": 36,
    "city": "Bucharest"
}

# Get the value for the "name" key
name = my_dict["name"]

# Print the value
print(name) # Output: Ion
```

You can also use the **get()** method to access the values in a dictionary. This method takes the key as an argument and returns the corresponding value. If the key is not found, the **get()** method returns **None** or a default value that you can specify.

```
# Get the value for the "age" key using the get()
method
age = my_dict.get("age")

# Print the value
print(age) # Output: 36

# Get the value for the "country" key (which does-
n't exist)
country = my_dict.get("country")

# Print the value
print(country) # Output: None
```

You can also use the **items()** method to access the key-value pairs in a dictionary. This method returns a new object of the **dict_items** class, which is a view of the dictionary's items. You can then use a for loop to iterate over the key-value pairs.

```
# Get the key-value pairs in the dictionary
items = my_dict.items()

# Print the items
print(items) # Output: dict_items([('name',
'Ion'), ('age', 36), ('city', 'Bucharest')])

# Iterate over the key-value pairs
for key, value in items:
    print(f"{key}: {value}")

# Output:
# name: Ion
# age: 36
# city: Bucharest
```

Adding and updating data in Dictionaries

In Python, dictionaries are a type of data structure that stores data in key-value pairs. To **add or update data** in a dictionary, you can use square bracket notation to specify the key for the data you want to add or update, and then assign a new value to that key. Here's an example:

```
my_dict = {
    "key1": "value1",
    "key2": "value2",
}

# Add a new key-value pair to the dictionary
my_dict["key3"] = "value3"

# Update the value for an existing key in the dictionary
my_dict["key2"] = "new_value2"
```

In this example, we start with a dictionary that has two **key-value pairs**. We then add a new key-value pair to the dictionary and update the value for an existing key in the dictionary.

The Get() method and “in” operator

In Python, dictionaries have a **get()** method that can be used to retrieve the value associated with a given key. This method is useful because it allows you to specify a default value that will be returned if the specified key does not exist in the dictionary (Charatan, Q., & Kans, A., 2022). An example is provided below:

```
my_dict = {  
    "key1": "value1",  
    "key2": "value2",  
}  
  
# Retrieve the value associated with "key1"  
value = my_dict.get("key1")  
  
# The value of "value" is now "value1"  
  
# Retrieve the value associated with "key3" (which  
# doesn't exist in the dictionary)  
value = my_dict.get("key3", "default_value")  
  
# The value of "value" is now "default_value"
```

The **get()** method is useful because it allows you to avoid key errors when trying to access values in a dictionary. In Python, you can also use the **in** operator to check if a particular key exists in a dictionary. Here's an example:

```
my_dict = {  
    "key1": "value1",  
    "key2": "value2",  
}  
  
# Check if "key1" exists in the dictionary  
if "key1" in my_dict:  
    # Do something...  
  
# Check if "key3" exists in the dictionary  
if "key3" in my_dict:  
    # Do something...
```

In this example, we use the `in` operator to check if a particular key exists in the dictionary. This allows us to avoid key errors and write more robust code.

Dictionary Pop(), Clear(), and Del()

Python dictionaries have several methods for removing data from the dictionary. The **pop()** method allows you to remove a specific **key-value pair** from a dictionary and return the value associated with that key. An example is shown below:

```
my_dict = {
    "key1": "value1",
    "key2": "value2",
}

# Remove the key-value pair associated with "key1"
# and return the value
value = my_dict.pop("key1")

# The value of "value" is now "value1"
# The dictionary now only contains one key-value
# pair: {"key2": "value2"}
```

The **clear()** method allows you to remove all key-value pairs from a dictionary. Here's an example:

```
my_dict = {
    "key1": "value1",
    "key2": "value2",
}

# Remove all key-value pairs from the dictionary
my_dict.clear()

# The dictionary is now empty
```

Finally, you can use the `del` keyword to remove a specific key-value pair from a dictionary. Here's an example:

```
my_dict = {
    "key1": "value1",
    "key2": "value2",
}

# Remove the key-value pair associated with "key1"
del my_dict["key1"]

# The dictionary now only contains one key-value
pair: {"key2": "value2"}
```

These methods are useful for removing data from dictionaries when you no longer need it.

Dictionaries are mutable too!

In Python, dictionaries are **mutable**, which means that you can change the data stored in a dictionary after it has been created. This contrasts with immutable data types, such as strings and tuples, which cannot be modified once they have been created. Here's an example of how you can change the data in a dictionary:

```
my_dict = {
    "key1": "value1",
    "key2": "value2",
}

# Change the value associated with "key1"
my_dict["key1"] = "new_value1"

# Add a new key-value pair to the dictionary
my_dict["key3"] = "value3"

# Remove a key-value pair from the dictionary
del my_dict["key2"]
```

In this case, we begin with a dictionary that has two **key-value pairs**. The value associated with one of the keys is then modified, a new **key-value pair** is added to the dictionary, and a **key-value pair** is

removed from the dictionary. Because dictionaries are mutable, you can use the methods and operations discussed earlier in this conversation to add, update, and remove data from a dictionary. This makes dictionaries a very powerful and flexible data type in Python.

Iterating Dicts Keys(), Values(), and Items()

Python dictionaries have three methods for iterating over the keys, values or **key-value pairs** in a dictionary: **keys()**, **values()**, and **items()**. The **keys()** method returns an iterator that produces the keys in the dictionary (Hunt, J., 2019). An example of using this method to iterate over the keys in a dictionary is shown below:

```
my_dict = {  
    "key1": "value1",  
    "key2": "value2",  
}  
  
# Iterate over the keys in the dictionary  
for key in my_dict.keys():  
    # Do something with the key...
```

The **values()** method returns an iterator that yields the values in the dictionary. Here's an example of how you can use this method to iterate over the values in a dictionary:

```
my_dict = {  
    "key1": "value1",  
    "key2": "value2",  
}  
  
# Iterate over the values in the dictionary  
for value in my_dict.values():  
    # Do something with the value...
```

Finally, the **items()** method returns an iterator that yields the key-value pairs in the dictionary. Here's an example of how you can use this method to iterate over the key-value pairs in a dictionary:

```
my_dict = {
    "key1": "value1",
    "key2": "value2",
}

# Iterate over the key-value pairs in the dictionary
for key, value in my_dict.items():
    # Do something with the key and value...
```

These methods are useful for iterating over dictionaries and accessing the data stored in them.

Fancy Dictionary merging

In Python, you can use the **update()** method to merge the key-value pairs from one dictionary into another dictionary. Here's an example:

```
dict1 = {
    "key1": "value1",
    "key2": "value2",
}

dict2 = {
    "key3": "value3",
    "key4": "value4",
}

# Merge the key-value pairs from dict2 into dict1
dict1.update(dict2)

# The dictionary now contains all of the key-value
# pairs from both dictionaries
# {"key1": "value1", "key2": "value2", "key3":
# "value3", "key4": "value4"}
```

In this case, we begin with two dictionaries that both have two **key-value pairs**. The `update()` method is then used to merge the **key-value pairs** from `dict2` into `dict1`. This produces a dictionary that includes all the **key-value pairs** from both dictionaries. The `update()` method is useful for combining the data from multiple dictionaries into a single dictionary. Note that if the same key exists in both dictionaries, the value from the second dictionary will overwrite the value from the first dictionary.

Lists and Dicts combined

In Python, you can use dictionaries and lists together to create complex data structures. For example, you can create a list of dictionaries, where each dictionary contains data for a different item. Here's an example:

```
# Create a list of dictionaries
my_list = [
    {
        "name": "Mihai",
        "age": 30,
    },
    {
        "name": "Mihaela",
        "age": 25,
    },
    {
        "name": "Andrei",
        "age": 35,
    },
]

# Access the first item in the list
first_item = my_list[0]

# The value of "first_item" is a dictionary
# {"name": "Mihai", "age": 30}

# Access the value associated with the "name" key
# in the first item
first_name = my_list[0]["name"]

# The value of "first_name" is "Mihai"
```

In this example, we create a list of dictionaries, where each dictionary contains data for a different person. We then access the first item in the list and retrieve the value associated with the “**name**” key in that item. This data structure allows you to store complex data in a way that is organized and easy to access. You can use the techniques discussed earlier in this conversation to add, update, and remove items from the list and dictionaries.

Fromkeys()

In Python, the **dict.fromkeys()** method is a static method that creates a new dictionary with the specified keys and a default value. Here’s an example of how you can use this method:

```
# Create a new dictionary with three keys and a
# default value of 0
my_dict = dict.fromkeys(["key1", "key2", "key3"],
0)

# The dictionary now contains the following key-
# value pairs:
# {"key1": 0, "key2": 0, "key3": 0}
```

In this example, we use the **fromkeys()** method to create a new dictionary with the specified keys and a default value of 0. This is a convenient way to create a dictionary with a known set of keys and default values. You can also use the **dict()** constructor to achieve the same result:

```
# Create a new dictionary with three keys and a
# default value of 0
my_dict = dict(key1=0, key2=0, key3=0)

# The dictionary now contains the following key-
# value pairs:
# {"key1": 0, "key2": 0, "key3": 0}
```

In this example, we use the **dict()** constructor to create a new dictionary with the specified keys and default values. This is equivalent to calling the **fromkeys()** method.

Update()

In Python, the **dict.update()** method allows you to merge the key-value pairs from one dictionary into another dictionary. Here's an example of how you can use this method:

```
dict1 = {  
    "key1": "value1",  
    "key2": "value2",  
}  
  
dict2 = {  
    "key3": "value3",  
    "key4": "value4",  
}  
  
# Merge the key-value pairs from dict2 into dict1  
dict1.update(dict2)  
  
# The dictionary now contains all of the key-value  
# pairs from both dictionaries  
# {"key1": "value1", "key2": "value2", "key3":  
# "value3", "key4": "value4"}
```

In this example, we start with two dictionaries that each have two key-value pairs. We then use the **update()** method to merge the key-value pairs from dict2 into dict1. This results in a dictionary that contains all of the key-value pairs from both dictionaries. The **update()** method is useful for combining the data from multiple dictionaries into a single dictionary. Note that if the same key exists in both dictionaries, the value from the second dictionary will overwrite the value from the first dictionary.

★ *Peak Dictionary exercise*

Here is an example of how you might solve the peak dictionary exercise:

```
# Create a dictionary
my_dict = {
    "key1": "value1",
    "key2": "value2",
    "key3": "value3",
}

# Define a function that takes a dictionary as an
argument
def find_peak(d):
    # Find the highest and lowest values in the
    dictionary
    highest_value = max(d.values())
    lowest_value = min(d.values())

    # Find the keys associated with the highest
    and lowest values
    highest_key = [k for k, v in d.items() if v ==
highest_value]
    lowest_key = [k for k, v in d.items() if v ==
lowest_value]

    # Return a tuple containing the keys and val-
    ues
    return (highest_key, highest_value),
(lowest_key, lowest_value)

# Call the function and print the result
peak = find_peak(my_dict)
print(peak)

# The output should be:
# (['key3'], 'value3'), (['key1'], 'value1')
```

In this example, we create a dictionary with three key-value pairs. We then define a function that takes a dictionary as an argument and uses the `values()` method to find the highest and lowest values in the dictionary. Next, we use list comprehension to find the keys associated with the highest and lowest values. Finally, we return a tuple containing the keys and values for the highest and lowest peaks.

When we call the function and print the result, we see that it correctly identifies the keys and values for the highest and lowest peaks in the dictionary.

Sets and Tuples

Sets and Tuples are two other popular **data types** that can be used to store and manipulate data. Sets are similar to dictionaries in that they store a collection of unique elements. However, unlike dictionaries, sets do not have keys or values; they only contain a collection of elements. An example of creating and using a set is shown below:

```
# Create a set
my_set = {1, 2, 3, 4}

# Check if an element exists in the set
if 3 in my_set:
    # Do something...

# Add an element to the set
my_set.add(5)

# Remove an element from the set
my_set.remove(4)
```

In this example, we create a set that contains four elements. We then use the `in` operator to check if a particular element exists in the set, and use the `add()` and `remove()` methods to add and remove elements from the set. Sets are useful for storing and manipulating collections of unique elements. **Tuples**, on the other hand, are similar to lists in that they can store a collection of elements. However, unlike lists, tuples are immutable, which means that the elements in a tuple

cannot be changed once the tuple has been created. Here's an example of how you can create and use a tuple:

```
# Create a tuple
my_tuple = (1, 2, 3, 4)

# Access an element in the tuple
first_element = my_tuple[0]

# The value of "first_element" is 1
```

In this example, we create a tuple that contains four elements. We then access the first element in the tuple using square bracket notation. Tuples are useful for storing and manipulating collections of elements that should not be modified once the tuple has been created.

[hints!] In Python, a set is an unordered collection of unique items, while a tuple is an immutable (unchangeable) sequence of items. Both sets and tuples are useful in data analysis because they allow you to store and organize data, perform calculations or transformations on the data, and then present the results clearly and concisely. For example, you might use a set to store a collection of unique data points, and then use built-in functions like `len` to calculate the number of unique items in the set. You might also use a tuple to store a fixed set of data points, and then access and manipulate the data using indexing and slicing. Sets and tuples are also useful in communication because they allow you to present data in a clear and organized way. For example, you might use a set to display the unique values in a dataset, or you might use a tuple to display the results of data analysis in a table or graph. Sets and tuples are important tools for data scientists and communication professionals, as they allow you to store and manipulate data flexibly and efficiently, and to present the results of your analysis in a clear and organized manner.

Tuple functionality

Tuples have several built-in methods and functions that you can use to manipulate the data stored in a tuple. One common operation is to concatenate two or more tuples together to create a new tuple. You can do this using the `+` operator:

```
# Create two tuples
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenate the tuples to create a new tuple
new_tuple = tuple1 + tuple2

# The value of "new_tuple" is (1, 2, 3, 4, 5, 6)
```

In this example, we create two tuples that each contain three elements. We then use the `+` operator to concatenate the tuples and create a new tuple that contains all of the elements from both tuples. Another common operation is to find the length of a tuple, which is the number of elements in the tuple. You can do this using the `len()` function:

```
# Create a tuple
my_tuple = (1, 2, 3, 4)

# Find the length of the tuple
tuple_length = len(my_tuple)

# The value of "tuple_length" is 4
```

In this example, we create a tuple that contains four elements. We then use the `len()` function to find the length of the tuple, which is the number of elements in the tuple. These operations are just a few examples of the built-in functionality that is available for tuples in Python.

[hints!] Tuples are a type of data structure that can be used in Data Science to store multiple items in a single variable. They are similar to lists, but unlike lists, tuples are immutable, which means that they cannot be changed once they are created. This makes them useful for storing data that should not be modified, such as data from a database or data that has been

processed and cleaned for analysis. There are several reasons why tuples can be useful in Python. Here are a few examples:

- Tuples are faster than lists. Because tuples are immutable, Python can create them and access their elements more efficiently than it can with lists. This can be particularly beneficial in large datasets or when working with complex algorithms.
- Tuples provide a simple way to group related data. For example, you might use a tuple to store a person's name and age, or a latitude and longitude coordinate. This can make your code more organized and easier to read.
- Tuples can be used as keys in dictionaries. Unlike lists, which are not hashable and cannot be used as dictionary keys, tuples can be used to create dictionaries that are indexed by a combination of values. (Bynum, M. L., Hackebeit, G. A., Hart, W. E., Laird, C. D., Nicholson, B. L., Siirola, J. D., ... & Woodruff, D. L., 2021)

Tuples can be used to return multiple values from a function. In Python, it is possible to return multiple values from a function by placing them in a tuple. This can be a convenient way to return related values without having to create a custom object. While they may not be suitable for every situation, they can provide some benefits over lists in certain scenarios.

Sets introduction

A **set** is a collection of unique elements in a particular order. It is similar to a list or an array, but unlike these data structures, sets only allow a single occurrence of each element. Sets are often used in Data Science to perform mathematical operations such as union, intersection, difference, and symmetric difference. In Python, sets are defined using curly braces (`{}`) or the `set()` function. For example, the following code creates a set containing the integers 1, 2, and 3:

```
my_set = {1, 2, 3}
```

You can also create a **set from** a list using the `set()` function, like this:

```
my_list = [1, 2, 3]
my_set = set(my_list)
```

Sets are unordered, which means that the elements in a set do not have a specific order. This means that you cannot access or modify elements in a set by index like you can with a list. However, you can iterate over the elements of a set in a for loop, like this:

```
for element in my_set:
    print(element)
```

Sets also support many of the same mathematical operations as lists, such as union, intersection, and difference. For example, the following code computes the union and intersection of two sets:

```
# Define two sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Compute the union of the two sets
union = set1.union(set2)
print(union) # Output: {1, 2, 3, 4, 5}

# Compute the intersection of the two sets
intersection = set1.intersection(set2)
print(intersection) # Output: {3}
```

Sets are a useful data structure for Data Science because they allow you to perform mathematical operations on unique elements in a particular order.

Set operators: Intersection, Union, Difference

In mathematics and computer science, the terms “**intersection**,” “**union**,” and “**difference**” refer to the operations that can be performed on sets. The **intersection** of two sets is a new set that includes only the elements common to both original sets. The **union** of two sets is a new set that includes all elements that are in either of the original sets. The **difference** between the two sets is a new set that includes all elements that are in the first set but not in the second set. “There are two types of keyword searches: word keyword search and phrase keyword search. In a phrase keyword search, quotation marks indicate the ordered set of words. For example, “**set operations**” is composed

of two words, i.e., set and operations, where set must be the first word and operations should be the second word after set.” (Takefuji, Y., 2022: 174). For instance, if we have two sets:

Set A = {1, 2, 3, 4, 5}
Set B = {4, 5, 6, 7, 8}

The intersection of **Set A** and **Set B** would be {4, 5}, as these are the only elements shared by both sets. The union of **Set A** and **Set B** would be {1, 2, 3, 4, 5, 6, 7, 8}, as this set includes all elements from both sets. The difference between **Set A** and **Set B** would be {1, 2, 3}, as these are the elements that are in **Set A** but not in **Set B**.

[hints!] When Use Sets in Data Science? **Sets** are used in a variety of mathematical and computer science contexts, including:

- In algebra, sets are often used to represent collections of objects or numbers. For example, a set might be used to represent the set of all prime numbers or the set of all even numbers.
- In geometry, sets can be used to represent collections of points, lines, or other geometric objects.
- In computer science, sets are often used to represent collections of data. For example, a set might be used to represent a group of users or a collection of files.
- In statistics, sets are often used to represent collections of data points. For example, a set might be used to represent the results of a survey.

In logic, sets are used to represent collections of objects or concepts. For example, a set might be used to represent the set of all natural numbers, or the set of all countries in the world (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620).

Sets are commonly used in Data Science to represent collections of data points or objects. For example, a set might be used to represent a group of users, a collection of files, or a set of observations from a dataset. In Data Science, sets are often used in conjunction with set operations (such as intersection, union, and difference) to analyze and manipulate data. For instance, the intersection of two sets could be used to find common elements between the sets, while the union of two sets might be used to merge the sets into a single collection. Additionally, sets are frequently utilized in Data Science to identify

unique elements in a collection of data. For example, a set might be used to remove duplicate entries from a dataset or to identify unique values in a column of a data frame. Sets are a useful tool in Data Science for organizing and analyzing data.

★ *Exercises with sets*

Here are some examples of exercises that involve working with sets:

- a) Given two sets A and B, find the intersection of the sets. For example, if $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$, the intersection would be $\{3\}$.
- b) Given two sets A and B, find the union of the sets. For example, if $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$, the union would be $\{1, 2, 3, 4, 5\}$.
- c) Given two sets A and B, find the difference between the sets. For example, if $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$, the difference would be $\{1, 2\}$.
- d) Given a set A, find all of the subsets of A. For example, if $A = \{1, 2, 3\}$, the subsets would be $\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$.
- e) Given a set A, find the complement of A. The complement of a set is the set of all elements that are not in the original set. For example, if $A = \{1, 2, 3\}$ and the universe (the set of all possible elements) is $\{1, 2, 3, 4, 5\}$, the complement of A would be $\{4, 5\}$.

Back to functions. Introducing args

In programming, “args” (short for “arguments”) refers to the input values that are passed to a function or method when it is called. These input values, provided by the caller of the function, are used by the function to carry out its intended operation. In most programming languages, the arguments passed to a function are usually declared inside the parentheses following the function’s name, separated by commas. For example, in the following code, **x** and **y** are the arguments passed to the add function:

```
def add(x, y):  
    return x + y
```

When calling the add function, the caller must provide values for the **x** and **y** arguments, which will be used by the function to perform its calculation. For example:

```
result = add(1, 2)  # result will be 3
```

In some cases, a function may have a variable number of arguments, in which case the function can use the **args** keyword to represent all of the remaining arguments. For example:

```
def sum(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total
```

In this case, the **sum** function uses the **args** keyword to represent all the remaining arguments that are passed to the function. This allows the caller to pass any number of arguments to the function, which will be added together to calculate the result. For example:

```
result = sum(1, 2, 3, 4, 5)  # result will be 15
```

The **args** keyword is a useful way to represent a variable number of arguments in a function, allowing the function to be more flexible and adaptable to different input values.

[hints!] Arguments in functions are good in data analysis and communication because they allow you to customize the behavior of a function and pass data or parameters to it. “Utilizing arguments in functions allows you to write more flexible and reusable code that can be applied to various data and scenarios. In data analysis, function arguments can be used to pass data from one part of your program to another, enabling you to perform calculations or transformations on the data. For instance, function arguments may be used to pass a dataset to a function that calculates statistical measures or plots the data. Function arguments can also be used to specify configuration options or parameters for a function, allowing you to customize its behavior. For example, you might use function arguments to specify the type of plot to generate

or the data transformation to apply. Arguments in functions are an important tool for data scientists and communication professionals, as they allow you to write flexible, reusable code that can be customized to meet the needs of different data analysis tasks and scenarios.

Introducing Kwargs

In programming, the term “**kwargs**” (short for “**keyword arguments**”) refers to a feature in some programming languages that allows a function or method to be called with named arguments, rather than just positional arguments. This can make the code easier to read and understand, and can also make it easier to pass arguments to a function in the correct order. In most programming languages, arguments are typically passed to a function by position, meaning that the caller must specify the values of the arguments in the same order that they are declared in the function. For example, in the following code, the `add` function expects the first argument to be the `x` value and the second argument to be the `y` value:

```
def add(x, y):  
    return x + y  
  
result = add(1, 2)  # result will be 3
```

However, when using **kwargs**, the caller can specify the arguments by name, rather than by position. This means that the arguments can be provided in any order, as long as the names are specified. For example:

```
def add(x, y):  
    return x + y  
  
result = add(x=1, y=2)  # result will be 3
```

In this case, the `add` function is called with named arguments, using the `x=` and `y=` syntax to specify the values for the `x` and `y` arguments. This allows the caller to specify the arguments in any order, making the code more readable and easier to understand. Additionally, **kwargs** can be used to specify default values for function arguments. For example:

```
def add(x=0, y=0):  
    return x + y  
  
result = add(y=1, x=2)  # result will be 3
```

In this case, the `add` function is defined with default values for the `x` and `y` arguments (0 and 0, respectively).

Parameter list ordering

The order of **parameters** in a function definition is important. When calling a function, the arguments provided must match the order of the parameters defined in the function. For example, consider the following function:

```
def greet(name, message):  
    print(f"Hello, {name}. {message}")
```

This function has two parameters: `name` and `message`. To call this function and provide values for these parameters, you would do something like this:

```
greet("Andreea", "How are you doing?")
```

In this case, the value of the **name** parameter is "Andreea" and the value of the **message** parameter is "How are you doing?". The order in which the arguments are provided must match the order of the parameters in the function definition, so the first argument is assigned to the `name` parameter and the second argument is assigned to the `message` parameter.

A common gotcha mutable default Args

One common mistake that many Python programmers make is using mutable objects as default arguments for functions. This can lead to some surprising behavior because the default value for the argument is only evaluated once the function is defined. This means that if the default value is a mutable object, and the function modifies that object, the changes will persist across multiple calls to the function. For example, consider the following function:

```
def append_to_list(value, lst=[]):  
    lst.append(value)  
    return lst
```

This function has a default value for the `lst` parameter, which is an empty list. If you call this function without providing a value for the `lst` parameter, it will use the default value of an empty list. However, because the default value is a mutable object (a list), and the function modifies that object (by appending a value to it), the changes to the object will persist across multiple calls to the function. For example, you might expect the following code to produce two separate lists, but it produces a single list with two elements:

```
print(append_to_list(1))  
print(append_to_list(2))
```

This happens because the default value for the `lst` parameter is evaluated only once when the function is defined, and the same list object is used as the default value for every call to the function. Since the function modifies this list, the changes persist across multiple calls. To avoid this behavior, you should never use mutable objects as default arguments for functions. Instead, you should provide a default value of **None** for the argument, and then check inside the function whether the argument has been provided. If it has not been provided, you can initialize it to the desired value.

Unpacking Args

Unpacking arguments is a way to assign the elements of an iterable (such as a **list** or **tuple**) to individual variables. This can be done using the ***** operator in a function call. For example:

```
def my_function(x, y, z):  
    print(f"x = {x}, y = {y}, z = {z}")  
  
my_list = [1, 2, 3]  
  
# Unpack the elements of my_list and pass them as  
# arguments to my_function  
my_function(*my_list)
```

This will print **x = 1, y = 2, z = 3**, showing that the elements of **my_list** have been unpacked and assigned to the individual variables **x**, **y**, and **z** in the function call.

★ *ArgsKwargs Exercises*

Here are a few exercises that you can try to practice using **args** and **kwargs** in Python:

- Write a function `sum_all` that takes any number of arguments and returns the sum of all of them. Use `*args` to accept a variable number of arguments.
- Write a function `concat_all` that takes any number of strings and concatenates them all into a single string, with a space between each string. Use `*args` to accept a variable number of arguments.
- Write a function `print_kwargs` that takes any number of keyword arguments and prints them all, along with their values. Use `**kwargs` to accept a variable number of keyword arguments.
- Write a function `combine_dicts` that takes any number of dictionaries and combines them into a single dictionary. Use `**kwargs` to accept a variable number of dictionaries and the `update()` method to combine them.
- Write a function `multiply` that takes two arguments, `x` and `y`, and an optional keyword argument `z`. If `z` is not provided, the function should return the product of `x` and `y`. If `z` is provided, the function

should return the product of *x*, *y*, and *z*. Use default values to set a default value for *z*.

Working with Errors

In Python, errors are represented by objects of type **Exception**. These objects are raised (or thrown) when a program encounters an error or exceptional situation that it cannot handle. To handle errors in Python, you can use **try** and **except** statements. The **try** statement specifies a block of code that the interpreter should try to execute. If an error occurs during the execution of this block, the interpreter will raise an **Exception** object. The **except** statement specifies a block of code that will be executed if an **Exception** is raised in the corresponding **try** block. For example:

```
try:
    # Some code that may raise an error
except Exception:
    # Code to handle the error
```

If an error occurs in the **try** block, the interpreter will immediately stop executing that block and move on to the **except** block. If no error occurs, the **except** block will be skipped and the code will continue to run as normal. You can also specify a specific type of **Exception** that you want to handle, rather than catching all exceptions as in the example above. For example:

```
try:
    # Some code that may raise an error
except ValueError:
    # Code to handle a ValueError
```

This will only handle **ValueError** exceptions, which are raised when a function or operation receives an argument of the wrong type or value. Any other type of **Exception** will not be caught by this **except** block and will be handled by the interpreter in the usual way. Additionally, you can use the **else** and **finally** statements in combination with **try** and **except** to provide additional code that should be executed in different scenarios. The **else** statement specifies a block of

code that will be executed if no **Exception** is raised in the corresponding try block.

[hints!] Working with errors is good for data analysis and communication because it allows you to identify and troubleshoot problems in your code, which can help you improve the reliability and accuracy of your data analysis. In Python, errors are represented by exception objects, which are raised when something goes wrong in your code. Various types of errors can occur in Python, including syntax errors, runtime errors, and semantic errors. To work with errors in Python, you can use try-except blocks to handle exceptions and prevent your code from crashing. For example:

```
try:
    # code that might raise an exception
except Exception:
    # code to handle the exception
```

You can also use the `raise` keyword to raise an exception manually, which can be useful for debugging or for signaling to the user that something has gone wrong. Working with errors is an important part of data analysis and communication, as it allows you to identify and troubleshoot problems in your code, and ensure that your data analysis is reliable and accurate.

Common error types

Several common error types can occur in Python. Some of the most common include:

- **SyntaxError:** This error occurs when the code is not written in the correct syntax according to the rules of the Python language. For example, if you forget to include a colon at the end of a for loop, you will get a `SyntaxError`.
- **NameError:** This error occurs when a variable is used in the code, but it has not been defined. For example, if you try to use a variable that you haven't defined yet, you will get a `NameError`.

- **TypeError:** This error occurs when you try to operate on a variable that is of the wrong type. For example, if you try to add a string and an integer together, you will get a `TypeError`.
- **IndexError:** This error occurs when you try to access an element in a list or a string using an index that is out of range. For example, if you try to access the 10th element of a list that only has 5 elements, you will get an `IndexError`.
- **KeyError:** This error occurs when you try to access a dictionary key that does not exist in the dictionary. For example, if you try to access the value for a key that is not in the dictionary, you will get a `KeyError`.
- **ValueError:** This error occurs when you try to pass an argument to a function that has the wrong type or value. For example, if you try to pass a string where an integer is expected, you will get a `ValueError` (VanderPlas, J. 2016).

These are just a few of the common error types that can occur in Python. To avoid these errors, it is important to carefully read and understand the error messages that Python provides, and to test your code thoroughly before running it.

Raising exceptions

Exceptions are events that occur during the execution of a program that disrupts the normal flow of instructions. In Python, exceptions can be raised (or thrown) using the `raise` keyword. For example, if you want to raise a **ValueError** exception because a function received an invalid argument, you can use the following code:

```
def some_function(arg):  
    if not isinstance(arg, int):  
        raise ValueError('arg must be an integer')
```

In this code, the `raise` keyword is used to raise a **ValueError** exception if the `arg` argument is not an integer. The string that is passed to the **ValueError** constructor is the error message that will be displayed to the user. You can also raise exceptions that are not built-in to Python by creating a custom exception class and then raising an instance of that class. For example:

```
class CustomError(Exception):  
    pass  
  
def some_function(arg):  
    if not isinstance(arg, int):  
        raise CustomError('arg must be an integer')
```

In this code, a custom exception class called **CustomError** is defined, and then an instance of that class is raised if the `arg` argument is not an integer. It is important to use exceptions carefully and only raise them when necessary. Raising exceptions can make your code more difficult to debug and can make it harder for users of your code to understand what is happening.

When to raise

When to raise an exception in your code is largely a matter of personal preference and the specific requirements of your program. However, there are some general guidelines that you can follow to help you decide when to raise an exception. First, you should raise an exception when a function receives an invalid argument. For example, if a function expects an integer but receives a string, you can raise a **TypeError** exception to indicate that the function has received an invalid argument. Second, you should raise an exception when a function encounters an error that it cannot handle itself. For example, if a function tries to open a file but the file does not exist, you can raise an **IOError** exception to indicate that the function was unable to complete its task. Third, you should raise an exception when a function cannot return a result that is meaningful or correct. For example, if a function is supposed to return the average of a list of numbers but the list is empty, you can raise a **ZeroDivisionError** exception to indicate that the function cannot return a valid result. Overall, the key is

to use exceptions to indicate when something has gone wrong in your code that cannot be handled by the code itself. By doing so, you can make your code more readable and easier to debug.

Try and except

The **try** and **except** keywords are used to handle exceptions that may be raised in your code. The **try** keyword is used to define a block of code that will be monitored for exceptions, and the **except** keyword is used to define a block of code that will be executed if an exception occurs. For example, the following code uses the **try** and **except** keywords to handle a **ZeroDivisionError** exception that may be raised when trying to divide two numbers:

```
try:
    result = x / y
except ZeroDivisionError:
    print('Cannot divide by zero')
```

In this code, the **try** block contains the code that may raise an exception (**dividing x by y**), and the **except** block contains the code that will be executed if a **ZeroDivisionError** is raised. If a **ZeroDivisionError** is raised, the **except** block will print a message to the user indicating that the division cannot be performed. The **try** and **except** keywords can also be used to handle multiple exceptions. For example:

```
try:
    result = x / y
except ZeroDivisionError:
    print('Cannot divide by zero')
except TypeError:
    print('x and y must be numbers')
```

In this code, the **try** block is the same as before, but now there are two **except** blocks. The first **except** block will handle a **ZeroDivisionError**, and the second **except** block will handle a **TypeError**. This allows the code to handle different types of exceptions and provide a more specific response to the user. The **try** and **except** keywords are an important part of handling exceptions in Python, and they can help you write more robust and user-friendly code.

LBYL and EAFP

LBYL and **EAFP** are two different approaches to handling exceptions in Python. **LBYL** stands for “**look before you leap**” and it is a technique where the code checks for potential errors before attempting to execute a block of code. **EAFP** stands for “**easier to ask for forgiveness than permission**” and it is a technique where the code tries to execute a block of code and then handles any exceptions that may be raised. Here is an example of **LBYL** in Python:

```
if os.path.exists(file_name):  
    with open(file_name) as f:  
        # read from the file  
else:  
    # handle the case where the file does not exist
```

In this code, the **LBYL** approach is used to check if a file exists before trying to open it. If the file does not exist, the code will handle the error by executing the code in the else block. Here is an example of **EAFP** in Python:

```
try:  
    with open(file_name) as f:  
        # read from the file  
except IOError:  
    # handle the case where the file does not exist
```

In this code, the **EAFP** approach is used to try to open the file, and then the except block will handle any **IOError** exceptions that may be raised if the file does not exist. Both **LBYL** and **EAFP** are valid approaches to handling exceptions in Python, and which one you choose to use will depend on your personal preference and the specific requirements of your program. Some developers prefer **LBYL** because it can make the code easier to read and understand, while others prefer **EAFP** because it can make the code more concise and elegant. Ultimately, the choice is up to you.

★ *Exercises with correct error handling*

Here are a few examples of exercises with **correct error** handling in Python:

- a) Write a function that takes a list of numbers and returns the sum of the numbers. If the list is empty, the function should raise a `ValueError` exception with the error message: “The list is empty.”

```
def sum_numbers(numbers):  
    if len(numbers) == 0:  
        raise ValueError('The list is empty.')  
    else:  
        return sum(numbers)
```

- b) Write a function that takes a string and an integer `n`, and returns the first `n` characters of the string. If the string is shorter than `n` characters, the function should raise an `IndexError` exception with the error message “The string is too short.”

```
def first_n_chars(string, n):  
    if len(string) < n:  
        raise IndexError('The string is too  
short.')  
    else:  
        return string[:n]
```

- c) Write a function that takes a dictionary and a key, and returns the value associated with the key. If the key does not exist in the dictionary, the function should raise a `KeyError` exception with the error message “The key does not exist in the dictionary.”

```
def get_value(dictionary, key):  
    if key not in dictionary:  
        raise KeyError('The key does not exist in  
the dictionary.')  
    else:  
        return dictionary[key]
```

These are just a few examples of exercises with correct error handling in Python. In each of these examples, the code uses the `raise` keyword to raise an exception if an error occurs and provides a clear and informative error message that will help the user understand what went wrong.

Modules

In Python, a module is a file that contains a collection of related functions and variables that can be used in other Python programs. Modules allow you to organize your code into logical units and reuse those units in multiple programs. To use a module in your code, you first need to import it using the `import` keyword. For example, to import the `math` module, which contains mathematical functions and constants, you would use the following code:

```
import math
```

Once you have imported a module, you can access the functions and variables it contains using the dot notation. For example, to use the `sqrt` function from the `math` module, you would use the following code:

```
import math

result = math.sqrt(9)  # result will be 3
```

In this code, the `math.sqrt` function is used to calculate the square root of 9. You can also import specific functions or variables from a module using the `from` keyword. For example, to import only the `sqrt` function from the `math` module, you would use the following code:

```
from math import sqrt

result = sqrt(9)  # result will be 3
```

In this code, the `sqrt` function is imported directly into the current namespace, so you can use it without needing to specify the `math` module name. Modules are an important part of Python, and they can help you organize your code and reuse it in multiple programs. By using modules, you can avoid duplicating code and make your programs more efficient and maintainable.

[hints!] Modules are good in data analysis and communication because they allow you to write modular, reusable code and structure your programs in a logical and organized way. In Python, a module is a file that contains a collection of related functions, classes, and variables. Modules allow you to di-

vide your code into smaller, more manageable pieces, making it easier to understand and maintain. In data analysis, modules can be used to encapsulate complex calculations or data processing tasks, improving the understandability and maintainability of your code. For example, you might create a module containing functions for performing statistical calculations, and then import and use those functions in other parts of your program. Modules also benefit communication by allowing you to present your code in a clear and organized way. For example, you might use modules to group related functions and classes or to separate code for different tasks or analyses. Modules are an important tool for data scientists and communication professionals, as they allow you to write modular, reusable code and structure your programs in a logical and organized way.

Working with built-in modules

Python includes several built-in modules that provide useful functions and variables for a wide range of tasks. Some of the most used built-in modules include:

- **math:** This module provides mathematical functions and constants, such as **sqrt** for calculating square roots and **pi** for the value of π .
- **random:** This module provides functions for generating random numbers and selecting random items from a list.
- **os:** This module provides functions for interacting with the operating system, such as **mkdir** for creating directories and **path.exists** for checking if a file or directory exists.
- **datetime:** This module provides functions for working with dates and times, such as **datetime.now** for getting the current date and time and **timedelta** for calculating the difference between two dates.

To use a built-in module in your code, you first need to import it using the **import** keyword. For example, to use the **math** module to calculate the **square root of 9**, you would use the following code:

```
import math
result = math.sqrt(9)    # result will be 3
```

You can also import specific functions or variables from a built-in module using the `from` keyword. For example, to import only the `sqrt` function from the `math` module, you would use the following code:

```
from math import sqrt

result = sqrt(9)  # result will be 3
```

Built-in modules are an important part of Python, and they can help you perform a wide range of tasks without needing to write your code. By using built-in modules, you can save time and make your programs more efficient and maintainable.

Most popular built-in modules for Data Science

Some of the most popular built-in modules for Data Science in Python include:

- **NumPy:** This module provides support for large, multi-dimensional arrays and matrices of numerical data, as well as functions for mathematical and statistical operations on these arrays.
- **Pandas:** This module provides data structures and tools for working with tabular data, such as dataframes and series. It also includes functions for data manipulation, aggregation, and visualization.
- **Scikit-learn:** This module provides a wide range of algorithms and tools for machine learning and data analysis, including classification, regression, clustering, and dimensionality reduction.
- **Matplotlib:** This module provides functions for generating static, animated, and interactive visualizations of data in 2D and 3D.
- **Seaborn:** This module is built on top of matplotlib and provides a higher-level interface for creating attractive statistical graphics and visualizations of large datasets.

These are just a few examples of popular modules for Data Science in Python. There are many other modules available, and the best ones to use will depend on your specific data analysis and machine learning tasks.

Fancy import syntax

In Python, there is a feature called “**fancy import syntax**” that allows you to import modules more concisely and flexibly. This feature is enabled by using the `import` keyword with a special syntax, which can take the following forms: **`import module as alias`**. This syntax allows you to import a module using a different name, called an “**alias**.” For example, to import the `math` module using the alias `m`, you would use the following code:

```
import math as m
```

`from module import *`: This syntax allows you to import all the functions and variables from a module into the current namespace. For example, to import all the functions and variables from the `math` module, you would use the following code:

```
from math import *
```

`from module import function1, function2, ...`: This syntax allows you to import specific functions or variables from a module, rather than importing the entire module. For example, to import only the `sqrt` and `pi` functions from the `math` module, you would use the following code:

```
from math import sqrt, pi
```

Fancy import syntax is a useful feature of Python that can make your code more concise and readable. By using this syntax, you can import modules and their functions and variables in a variety of different ways, depending on your needs and preferences.

Creating custom modules

In Python, you can create your custom modules by defining your functions and variables in a separate Python file and then importing that file into your main program. This allows you to organize your code into logical units and reuse those units in multiple programs. To create a custom module, you first need to create a new Python file and define your functions and variables in that file. For example, suppose you have a file called **my_functions.py** that contains the following code:

```
def say_hello():  
    print('Hello!')  
  
def say_goodbye():  
    print('Goodbye!')  
  
my_name = 'Ioana'
```

This code defines two functions, **say_hello** and **say_goodbye**, as well as a variable **called my_name**. These functions and variables can be used in other programs by importing the **my_functions** module. To import the **my_functions** module into your main program, you would use the **import** keyword followed by the name of the module. For example, suppose you have a file called **main.py** that contains the following code:

```
import my_functions  
  
my_functions.say_hello() # will print 'Hello!'  
my_functions.say_goodbye() # will print 'Good-  
bye!'  
  
print(my_functions.my_name) # will print 'Ioana'
```

In this code, the **my_functions** module is imported using the **import** keyword, and then the **say_hello** and **say_goodbye** functions and the **my_name** variable are accessed using the dot notation. Creating custom modules in Python is a useful way to organize your code and reuse it in multiple programs.

3rd party modules Pip & PyPI

Pip is a package manager for Python packages or modules. It is used to install and manage software packages written in Python. **PyPI**, the **Python Package Index**, is a repository of software packages for the Python programming language. It is used by **pip** to download and install packages from the internet. To use a specific module or library in your Python code, you can install it using **pip**. For instance, if you want to use the requests module to make **HTTP** requests, you can install it with the following command:

```
pip install requests
```

Once the package is installed, you can use it in your code by importing it:

```
import requests
```

Pip and **PyPI** make it easy to find, install, and use third-party modules in your Python projects. They are essential tools for any Python programmer.

Our first Pip package!

To create and publish your Python package using pip, you will need to follow these steps:

- Create a Python package: This involves creating a Python module or a set of modules and arranging them in a directory structure that defines the package.
- Install the setuptools and wheel packages: These packages provide the tools necessary for building and distributing Python packages. You can install them using the following command:
 - `pip install setuptools wheel`
- Create a setup.py file: This file is used by setuptools to define the metadata for your package, such as the package's name, version, and dependencies.
- Build your package: Use the Python setup.py sdist bdist_wheel command to build your package and create a distribution archive.

- Upload your package to PyPI: Use the twine tool to upload your package to PyPI, the Python Package Index.
- Once your package is uploaded to PyPI, other users can install it using pip. For example, they can use the following command to install your package:

```
pip install <your_package_name>
```

Publishing your Python package can be a great way to share your code with others and contribute to the Python community. It can also be a valuable learning experience and a good way to improve your skills as a Python programmer.

★ *Sentiment analysis fun project installation*

To install and use a **sentiment analysis fun project**, you will need to follow these steps:

Install the required packages: The project will likely require some Python packages, such as **nlTK** and **scikit-learn**. You can install these packages using **pip**, the Python package manager. For example, you can use the following command to install **nlTK**:

```
pip install nlTK
```

Download any necessary data: Some sentiment analysis projects may require additional data, such as pre-trained word vectors or sentiment lexicons. These data files should be included in the project's documentation, and you can download them and save them to your local machine. Follow the project's instructions: Each sentiment analysis project will have its own set of instructions for how to use it. These instructions should be included in the project's documentation, and you will need to follow them carefully to set up and use the project. Once you have installed the required packages and downloaded any necessary data, you should be able to use the sentiment analysis project. You can then run the project's code on your text data to perform sentiment analysis.

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of “**objects**”, which can contain data and code that manipulates that data. In **OOP**, programs are organized around the data they manipulate, and the code that manipulates the data is encapsulated in objects. One of the key features of **OOP** is encapsulation, which refers to the bundling of data and the methods that operate on that data within a single unit. This allows for modular and reusable code, as objects can be easily passed around in a program and used in different contexts. Another key feature of **OOP** is inheritance, which allows objects to inherit characteristics from parent objects. This allows for efficient code reuse, as objects can be derived from existing objects and only need to define the differences from their parent objects. **OOP** is a powerful programming paradigm that can help programmers write more organized and reusable code. Some of the benefits of **OOP** include code reuse, modularity, extensibility, and maintainability.

[**hints!**] Object-oriented programming (OOP) is good in data analysis and communication because it allows you to write modular, reusable code and structure your programs in a logical and organized way. OOP is a programming paradigm that is based on the concept of “objects”, which are data structures that contain both data and methods (functions) that operate on that data. In data analysis, OOP can be used to encapsulate complex calculations or data processing tasks, making it easier to understand and maintain your code. OOP can be used to create a class representing a dataset and define methods for calculating or transforming data. It can also improve communication by allowing you to group related functions and variables into classes or separate code for various tasks or analyses. This can make your code more modular and reusable, and help you organize and structure your programs logically. OOP is a valuable skill for data scientists and com-

munication professionals to master, as it allows them to write efficient and organized code.

Class Syntax

In object-oriented programming, a class is a blueprint for creating objects. It defines the properties and behaviors that an object of that class will have. Here is an example of a simple **Person** class in Python:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am
{self.age} years old.")
```

The **Person** class has two properties, `name`, and `age`, and one method, `greet()`. The `__init__()` method is a special method that is called when an object of the **Person** class is created. It is used to initialize the object's properties. To create an object of the **Person** class, you can use the **Person()** constructor, like this:

```
p = Person("Ioana Pop", 30)
```

This creates a **Person** object with the name "**Ioana Pop**" and age 30. To access the object's properties and methods, you can use the dot notation, like this:

```
p.name    # "Ioana Pop"
p.age     # 30
p.greet()  # "Hello, my name is Ioana Pop and I
am 30 years old."
```

Classes are a fundamental concept in object-oriented programming, and they provide a powerful way to organize and reuse code.

Writing our first class

To write your first class in Python, you will need to follow these steps: choose a name for your class: The name of your class should be descriptive and represent the concept or object that your class represents. For example, if you are creating a class to represent a **dog**, you could name your class **Dog**. Define the `__init__()` method: The `__init__()` method is a special method that is called when an object of your class is created. It is used to initialize the object's properties. The `__init__()` method should take at least one argument, `self`, which refers to the object being created. In addition to `self`, the `__init__()` method can take additional arguments to initialize the object's properties. For example, if you are creating a **Dog** class, the `__init__()` method might take arguments for the **dog**'s name and breed.

Define other methods: In addition to the `__init__()` method, your class can define other methods that represent the actions or behaviors that objects of that class can perform (Atwi, H., Lin, B., Tsantalis, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., ... & Lanza, M., 2021). For example, if you are creating a **Dog** class, you might define a `bark()` method that makes the dog bark. Here is an example of a **Dog** class in Python:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print("Woof!")
```

Instance methods

In Python, **instance methods** are methods that are defined on a class and are bound to an instance of that class. This means that they can only be called by an instance of the class, rather than on the class itself. For example, suppose we have a class `Person` with an instance method `greet`:

```
class Person:
    def greet(self):
        print("Hello!")

person = Person()
person.greet() # Output: "Hello!"
```

In this example, the **greet method** is an instance method because it is defined in the **Person** class and is bound to the person instance of that class. We call the method using the dot notation `person.greet()`, which specifies that the **greet method** is being called on the person instance. **Instance methods** are defined to operate on an instance of a class and can only be called on an object. An example of this is the `greet` method we previously defined, which can only be used by a **Person** object and not called on its own. This allows us to create behavior that is specific to a particular instance of a class. In Python, the first argument of an **instance method** is always `self`, which refers to the instance on which the method is being called. In the **greet method** above, `self` refers to the person instance. The self-argument is used to access attributes and methods of the instance within the method. For example, we could modify the **greet method** to include the name of the person it is being called on:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

person = Person("Ion")
person.greet() # Output: "Hello, Ion!"
```

★ *Practicing Instance methods*

Instance methods are defined inside a class and are used to access information or modify the attributes of a particular instance of that class. To practice using **instance methods**, you can try defining a simple class with some attributes and a couple of instance methods that operate on those attributes. For example, you could define a **Person** class with attributes like name, age, and gender, and then define **instance methods** like `greet` and `increment_age` that allow you to access and modify those attributes for a particular instance of the **Person** class. Here's an example of how that might look:

```
class Person:
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def greet(self):
        print(f"Hello, my name is {self.name}")

    def increment_age(self):
        self.age += 1
```

To use these **instance methods**, you would first need to create an instance of the **Person** class, like this:

```
person = Person("Ion", 34, "male")
```

Then you can call the instance methods on that instance like this:

```
person.greet() # prints "Hello, my name is Ion"
person.increment_age()
print(person.age) # prints 35
```

This is just a simple example, but it should give you an idea of how **instance methods** work and how you can use them to access and modify the attributes of a particular instance of a class.

Class Attributes

Class attributes are attributes that are defined at the class level, rather than at the instance level. In Python, **class attributes** are defined outside of any instance methods, like this:

```
class Person:
    # class attribute
    species = "Homo sapiens"

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    # instance method
    def greet(self):
        print(f"Hello, my name is {self.name}")
```

In this example, the `species` attribute is a **class attribute** that is shared by all instances of the **Person** class. You can access the **class attribute** using the **class name** and the **attribute name**, like this:

```
print(Person.species) # prints "Homo sapiens"
```

You can also access **class attributes** using an instance of the class, like this:

```
person = Person("Mihai", 34, "male")
print(person.species) # prints "Homo sapiens"
```

However, keep in mind that if you modify the value of a **class attribute** using an instance, that change will affect all other instances of the class as well, since **class attributes** are shared among all instances of the class. For example:

```
person.species = "Homo neanderthalensis"
print(Person.species) # prints "Homo
neanderthalensis"
```

In this case, changing the `species` attribute for the `person` instance also changed the value of the `species` attribute for the **Person** class, and therefore for all other instances of the **Person** class as well. This is something to be aware of when using class attributes in your code.

Class Methods

Class methods are methods that are defined at the class level, rather than at the instance level. In other words, they are methods that are associated with a class itself, rather than with a particular instance of that class. **Class methods** are marked with the **@classmethod decorator**, which is applied immediately before the method definition. Here's an example of a class method in Python:

```
class Person:
    # class attribute
    species = "Homo sapiens"

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    # instance method
    def greet(self):
        print(f"Hello, my name is {self.name}")

    # class method
    @classmethod
    def get_species(cls):
        return cls.species
```

In this example, the `get_species` method is a **class method** that returns the value of the `species` **class attribute**. Notice how the first argument to the method is `cls`, rather than `self`, which is the convention for **class methods**. This is because the `cls` argument refers to the **class** itself, rather than to a particular **instance** of the **class**. To call a **class method**, you use the same syntax as for calling an instance method, but you use the class name rather than an instance of the class (VanderPlas, J. 2016). For example:

```
print(Person.get_species()) # prints "Homo sapiens"
```

You can also call a **class method** using an instance of the class, but in this case, the instance will be ignored, and the class method will be called using the class itself. For example:

```
person = Person("Ion", 34, "male")
print(person.get_species()) # prints "Homo sapiens"
```

Inheritance basics

Inheritance is a fundamental concept in object-oriented programming (OOP) languages. It allows programmers to create new classes that are derived from existing classes. The new classes, known as derived classes, inherit all the attributes and behaviors of the existing classes, which are known as base classes. This means that the derived class can use all of the properties and methods of the base class and can also define additional properties and methods of its own. For example, imagine that you are creating a program to simulate different types of vehicles. You might define a base class called **Vehicle**, which includes properties such as **make**, **model**, and **year**, and methods such as **start()** and **stop()**. You could then create derived classes for specific types of vehicles, such as **Cars**, **Trucks**, and **Motorcycles**, which would inherit all the attributes and behaviors of the **Vehicle** class. The **Car** class might include additional properties such as **num_doors** and **trunk_size**, and additional methods such as **honk()** and **open_trunk()**. **Inheritance** is a powerful tool because it allows you to reuse code and avoid repeating yourself. It also helps to organize your code and make it more modular and extensible.

The Super() function

The **super()** function is used to give a derived class access to the methods and properties of a base class. It is typically used in the **__init__()** method of the derived class, to call the **__init__()** method of the base class. This allows the derived class to initialize all the attributes and behaviors of the base class, and then add additional attributes and behaviors of its own. Here is an example of how the **super()** function might be used:


```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start(self):
        print("Starting the engine!")

class Car(Vehicle):
    def __init__(self, make, model, year,
num_doors):
        super().__init__(make, model, year)
        self.num_doors = num_doors

    def honk(self):
        print("Honk! Honk!")

my_car = Car("Dacia", "1310", 1969, 4)
my_car.start() # Output: "Starting the engine!"
my_car.honk()  # Output: "Honk! Honk!"
```

In this example, the **Car** class is derived from the **Vehicle** class, and the `__init__()` method of the **Car** class uses the `super()` function to call the `__init__()` method of the **Vehicle** class. This allows the **Car** class to initialize all the attributes of the **Vehicle** class, such as `make`, `model`, and `year`, and also add its additional attribute, `num_doors`.

PART II

Visualization in Data Science using the most important Python modules

Once, we have been introduced to the universal Python **programming**, in the first part, the **Communication and Data Science** specialist needs to discover, analyze, and use this data in his core business, which consists of extracting key data, structuring and visualizing it in graphs ideal for his purposes. Python has a wide range of libraries/modules and frameworks specifically designed for Data Science tasks. In this Part II of the book, we will learn essentials about key modules such as **Pandas** for visualization, data manipulation, and analysis, **NumPy** for numerical computation, **Scikit-learn** for machine learning, data visualization with **Matplotlib** and **Seaborn**, **Web Scraping** for extracting data from websites, etc. Python can be used for a variety of data science tasks including:

- Cleaning and preparing data for analysis
- Exploring and visualizing data
- Performing statistical analysis and hypothesis testing
- Building and evaluating machine learning models
- Implementing algorithms for natural language processing (**NLP**) and computer vision
- Developing and deploying data-driven applications and services (Navlani, A., Fandango, A., & Idris, I., 2021).

Some of the key advantages of using Python for data science include its simplicity and readability, its support for integration and in-

teroperability, and its extensive ecosystem of packages and libraries. This makes it a great choice for both experienced data scientists and those who are just starting in the field. Some other most used libraries for data visualization and data analysis are **Matplotlib** and **Seaborn**. **Matplotlib** is a **2D** plotting library that provides a wide range of visualization tools and options. It is often used to create static charts, plots, and graphs, and to customize the appearance and formatting of these visualizations. **Seaborn** is a **data visualization** library that is built on top of **Matplotlib**. It provides a high-level interface for creating attractive and informative statistical graphics. **Seaborn** is particularly useful for exploring and visualizing multivariate data sets, and for creating more complex visualizations such as heatmaps, time series plots, and violin plots. **Pandas** is a library for data manipulation and analysis. It provides a high-performance and easy-to-use data structure called a **DataFrame**, which allows you to manipulate and analyze tabular data in a similar way to a spreadsheet. **Pandas** also include a range of tools for performing operations such as filtering, sorting, aggregation, and pivoting on data sets. **Beautiful Soup (BS4)** is a library for extracting data from **HTML** and **XML** documents. It is commonly used for web scraping, which involves collecting data from websites and storing it in a structured format for analysis. **Selenium** is a library for automating web browsers. It can be used to simulate user interactions with a website, such as clicking links, filling out forms, and scrolling through pages. This can be useful for testing the functionality of a website, or for performing tasks that would be time-consuming or tedious to do manually. **Scrapy** is a web scraping framework for Python. It provides a set of tools and libraries for extracting data from websites and storing it in a structured format. **Scrapy** offers several features that make it well-suited to large-scale scraping projects, including the ability to pause and resume scraping, support for concurrent requests, and an extensible architecture that allows developers to add custom functionality. Additionally, **Scrapy** is built on top of the **Twisted** networking engine, which makes it highly efficient and scalable.

In conclusion, why is it good to learn important modules to do Data Science? There are several reasons why it is good to learn important modules to do data science. First and foremost, learning these modules will give you the skills and knowledge you need to effectively work with data and perform various data science tasks. This can include

everything from collecting, cleaning, and organizing data, to performing complex analyses and creating visualizations. Additionally, learning these modules will allow you to integrate with the broader data science community more easily, as many of these modules are widely used and supported. This can help you to collaborate with others, access useful resources and support, and stay up to date with the latest developments in the field. Furthermore, having a strong foundation in these modules can help to make you more competitive in the job market, as many employers are looking for data scientists who have experience working with these tools and technologies. Learning these modules can also open new career opportunities and allow you to work on a wider range of projects and tasks. Learning important modules to do data science is an essential part of becoming a successful data scientist and staying current in the field.

[hints!] Visualization is an important tool for data analysis and communication because it allows us to understand and communicate insights and trends in data quickly and effectively. When working with large or complex datasets, it can be difficult to understand and make sense of the data using just text and numbers. Visualization helps to bring the data to life by presenting it in a graphical or visual format, which makes it easier for people to process and understand. Additionally, visualization can help to highlight patterns, trends, and relationships in the data that might not be immediately apparent from looking at the raw data. This can be particularly useful when communicating findings to others, as it allows you to communicate the key takeaways clearly and effectively from the analysis.

Let's start applying **Python** in **Data Science**!

Introduction to Pandas module

Pandas is a popular open-source Python library for data analysis and manipulation. It provides a powerful set of tools and data structures for working with structured and unstructured data and is widely used in the fields of data science, machine learning, and scientific computing. The core data structure in Pandas is the **DataFrame**, which is a two-dimensional table of data with rows and columns. **DataFrames** can

be created from a variety of sources, including **CSV** files, **Excel** sheets, and **SQL** databases. They can also be created from other data structures, such as **NumPy** arrays and dictionaries. Pandas offer a rich set of functions and methods for working with **DataFrames**, including functions for reading, writing, and importing data, as well as functions for aggregating, summarizing, and transforming data. Pandas also have powerful visualization capabilities and integrate seamlessly with other popular Python libraries, such as **NumPy**, **Matplotlib**, and **Scikit-learn**. Pandas is a valuable tool for any data scientist and is essential for working with and analyzing data in Python.

[hints!] Pandas is a widely used Python library that is particularly helpful for data analysis and manipulation. It offers many functions and methods that can be used to work with and modify data in different formats, making it a useful tool for data analysis and communication. In addition to these data manipulation capabilities, Pandas also provide several functions for summarizing and aggregating data, such as calculating means, medians, and standard deviations. This can be particularly useful for exploring and understanding patterns and trends in the data. The combination of these capabilities makes Pandas a powerful and versatile tool for data analysis and communication. It allows data scientists to work with and manipulate data, and generate insights, and communicate findings clearly and effectively quickly and easily.

How to install Pandas?

To install **Pandas** in Python, you can use the **pip** package manager. First, make sure that **pip** is installed on your system by running the following command:

```
pip --version
```

If **pip** is not installed, you can install it by running the following command:

```
python -m ensurepip --default-pip
```

Once **pip** is installed, you can use it to install **Pandas** by running the following command:

```
pip install pandas
```

Alternatively, you can use the **Anaconda** distribution of Python, which includes **Pandas** and many other popular data science libraries. To install **Anaconda**, visit the Anaconda website and follow the instructions for your operating system. Once **Pandas** is installed, you can use it in your Python scripts by importing it at the beginning of your code:

```
import pandas as pd
```

After importing **Pandas**, you can start using its data structures and functions to work with and analyze your data.

Create a Series in Pandas

To create a series in Pandas, you can use the **pd.Series** constructor. Here's an example:

```
import pandas as pd

# Create a list of values
values = [1, 2, 3, 4, 5]

# Create a series using the list of values
series = pd.Series(values)

# Print the series
print(series)
```

This code will create a series with the values **1, 2, 3, 4, and 5**, and print it to the console. You can also specify the index for the series, which can be a list of labels for each value in the series. Here's an example:

```
import pandas as pd

# Create a list of values
values = [1, 2, 3, 4, 5]

# Create a list of labels for the values
index = ['a', 'b', 'c', 'd', 'e']
```

```
# Create a series using the list of values and the
list of labels
series = pd.Series(values, index=index)

# Print the series
print(series)
```

This code will create a series with the same values as before, but with the labels 'a', 'b', 'c', 'd', and 'e' for each value. When you print the series, the labels will be shown alongside the values.

Create a DataFrame in Pandas

To create a dataframe in Pandas, you can use **pandas.DataFrame** constructor. Here's an example:

```
import pandas as pd

# Create a list of values
values = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Create a dataframe using the list of values
df = pd.DataFrame(values)

# Print the dataframe
print(df)
```

This code will create a **dataframe** with **3 rows** and **3 columns**, with the values **1, 2, 3** in the first row, **4, 5, 6** in the second row, and **7, 8, 9** in the third row. You can also specify the column names and row labels for the **dataframe**. Here's an example:

```
import pandas as pd

# Create a list of values
values = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Create lists of labels for the columns and rows
columns = ['a', 'b', 'c']
index = ['x', 'y', 'z']
```



```
# Create a dataframe using the list of values and  
the lists of labels  
df = pd.DataFrame(values, columns=columns,  
index=index)  
  
# Print the dataframe  
print(df)
```

This code will create a **dataframe** with the same values as before, but with the column labels ‘a’, ‘b’, and ‘c’, and the row labels ‘x’, ‘y’, and ‘z’. When you print the **dataframe**, the labels will be shown alongside the values.

[hints!] The DataFrame is a fundamental data structure in Pandas that are commonly used for storing and working with tabular data. It is particularly useful for data analysis and communication because it allows you to easily manipulate and analyze large datasets flexibly and efficiently. One key feature of the DataFrame is its ability to store and manipulate data in a tabular format, with rows and columns. This makes it easy to work with data that is organized into a standard table or spreadsheet structure. Additionally, the DataFrame provides a wide range of functions and methods for working with and manipulating data, such as selecting, filtering, and aggregating data, as well as performing mathematical operations on the data. Another reason why the DataFrame is useful for data analysis and communication is because it allows you to easily visualize the data using tools such as Matplotlib and Seaborn. This can be particularly useful for communicating findings and insights to others, as visualization can help to convey the key takeaways clearly and effectively from the analysis. The DataFrame is a powerful and flexible tool that is widely used in data analysis and communication and is an essential part of the Pandas library.

Read a CSV file with Pandas

A **CSV** (Comma Separated Values) file is a common and convenient way to store data in a tabular format. It's a simple text file that contains rows of data, with each row containing several values separated by a delimiter character (usually a comma). **CSV files** are important in Panda's data analysis because they provide a simple and flexible way to load and manipulate data. Pandas provide a few functions and methods that can be used to read **CSV files**, manipulate the data, and perform various data analysis tasks. This makes it easy to work with data from a **CSV file** in Pandas. In addition, **CSV files** are widely used and supported by many different applications and tools, so it's easy to share data in this format with others. This makes **CSV files** a popular choice for data storage and exchange. To read a **CSV file** with Pandas, you can use **pandas.read_csv** function. Here's an example:

```
import pandas as pd

# Read the CSV file
df = pd.read_csv('my_file.csv')

# Print the first 5 rows of the dataframe
print(df.head())
```

This code will read the **CSV file** 'my_file.csv' and create a dataframe with the data from the file. The **head()** method is used to print the first 5 rows of the **dataframe** (Long, J. D., & Teetor, P., 2019). You can also specify additional options when reading the **CSV** file. For example, you can specify the delimiter character (if it's not a comma), the names of the columns, and more. Here's an example:

```
import pandas as pd

# Read the CSV file, specifying the delimiter
character and the column names
df = pd.read_csv('my_file.csv', delimiter=';',
names=['col1', 'col2', 'col3'])

# Print the first 5 rows of the dataframe
print(df.head())
```

This code will read the **CSV file** `'my_file.csv'` using a semicolon (`;`) as the delimiter character, and specifying the names `'col1'`, `'col2'`, and `'col3'` for the columns. The **dataframe** will be created with these column names, and the first **5 rows** will be printed to the console.

Advanced parameters

Many advanced parameters can be used in Pandas for different purposes. Here are some examples:

- **index_col**: Specifies the column(s) to use as the row labels of the DataFrame.
- **usecols**: Specifies the column(s) to use, by column name or by column index.
- **skiprows**: Specifies the number of rows to skip at the beginning of the file.
- **skipfooter**: Specifies the number of rows to skip at the end of the file.
- **na_values**: Specifies the values to consider as missing or null values.
- **nrows**: Specifies the number of rows to read from the file.
- **iterator**: Returns a TextFileReader object for iteration or lazy loading of the data.
- **chunksize**: Specifies the number of rows to include in each chunk when using the iterator (McKinney, W., & Team, P. D., 2015).

These parameters can be used in various functions and methods in Pandas, such as **read_csv**, **read_excel**, and **read_json**. For example, to read a **CSV file** and skip the first **3 rows**, you can use the **skiprows** parameter like this:

```
import pandas as pd

# Read the CSV file, skipping the first 3 rows
df = pd.read_csv('my_file.csv', skiprows=3)

# Print the first 5 rows of the dataframe
print(df.head())
```

This code will read the **CSV file 'my_file.csv'**, skipping the first **3** rows, and create a **dataframe** with the data from the file. The **head()** method is used to print the first **5 rows** of the **dataframe**. The code uses several advanced parameters in the **pandas.read_csv** function. Here's what each parameter does:

- **sep=',':** This specifies the delimiter character used in the **CSV file**. In this case, it's a comma.
- **names=['col1', 'col2']:** This specifies the names of the columns in the **dataframe**. In this case, the columns will be named **'col1'** and **'col2'**.
- **index_col=0:** This specifies that the first column in the **CSV file** should be used as the index of the **dataframe**.
- **encoding='utf-8':** This specifies the character encoding used in the **CSV file**. In this case, it's **'utf-8'**.
- **nrows=3:** This specifies that only the first **3** rows of the **CSV file** should be read and included in the **dataframe** (Long, J. D., & Tector, P., 2019).

This code will read the **CSV file 'filename.csv'** using these parameters, and create a **dataframe** with the specified column names, index, and several rows.

Selecting rows and columns in Pandas

To select rows and columns in Pandas, you can use **pandas.DataFrame.loc** and **pandas.DataFrame.iloc** methods. The **loc** method is used to select rows and columns by label. For example, to select the row with **label 'x'** and the column with **label 'b'**, you can use the following code:

```
import pandas as pd

# Create a dataframe with some sample data
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6],
                  'c': [7, 8, 9]},
                  index=['x', 'y', 'z'])

# Select the row with label 'x' and the column
# with label 'b'
value = df.loc['x', 'b']

# Print the selected value
print(value)
```

This code will select the value **4** from the **dataframe**, which is the intersection of the row with label **'x'** and the column with label **'b'**. The **iloc** method is used to select rows and columns by position (integer index). For example, to select the first row and the second column, you can use the following code:

```
import pandas as pd

# Create a dataframe with some sample data
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6],
                  'c': [7, 8, 9]},
                  index=['x', 'y', 'z'])

# Select the first row and the second column
value = df.iloc[0, 1]

# Print the selected value
print(value)
```

This code will select the value **2** from the **dataframe**, which is the intersection of the first row and the second column. Both the **loc** and **iloc** methods can be used to select multiple rows and columns by specifying a slice or a list of **labels/indices** (McKinney, W., & Team, P. D., 2015). For example, to select the first and third rows and the second and third columns, you can use the following code:

```
import pandas as pd

# Create a dataframe with some sample data
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6],
                  'c': [7, 8, 9]},
                  index=['x', 'y', 'z'])

# Select the first and third rows, and the second
# and third columns
subset = df.loc[['x', 'z'], ['b', 'c']]

# Print the selected subset of the dataframe
print(subset)
```

This code will select the subset of the **dataframe** that contains the rows with labels '**x**' and '**z**', and the columns with labels '**b**' and '**c**'. The resulting **dataframe** will contain these rows and columns and will be printed to the console. For selecting rows and columns in a Pandas dataframe, here's what each line of code does:

- **df['col1']**: This selects the column with label 'col1' from the dataframe df.
- **df[['col1', 'col2']]**: This selects the columns with labels 'col1' and 'col2' from the dataframe df.
- **df.head(2)**: This shows the first 2 rows of the dataframe df.
- **df.tail(2)**: This shows the last 2 rows of the dataframe df.
- **df.loc['A']**: This selects the row with label 'A' from the dataframe df.
- **df.loc[['A', 'B']]**: This selects the rows with labels 'A' and 'B' from the dataframe df.

These methods use the **loc** and **head/tail** methods, which were explained in the previous answer. These are some of the ways you can use these methods to select specific rows and columns from a Pandas **dataframe**.

Data wrangling in Pandas

Data wrangling in Pandas is the process of cleaning, transforming, and preparing data for further analysis or visualization. It typically involves several steps, such as:

- **Reading** the data from a file or database
- **Handling** missing or invalid values
- **Filtering** or subsetting the data
- **Combining** or merging multiple datasets
- **Reshaping** or pivoting the data
- **Grouping** and summarizing the data
- **Sorting** or ordering the data

Pandas provide several functions and methods that can be used to perform these tasks, such as **read_csv**, **dropna**, **groupby**, and **pivot_table** (McKinney, W., & Team, P. D., 2015). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. “ O'Reilly

Media, Inc.”). For example, to read a **CSV file**, filter out rows with missing values, and group the data by one or more columns, you can use the following code:

```
import pandas as pd

# Read the CSV file, skipping rows with missing
values
df = pd.read_csv('my_file.csv', na_values='',
keep_default_na=False)

# Group the data by one or more columns
grouped = df.groupby(['col1', 'col2'])

# Print the first 5 rows of each group
for name, group in grouped:
    print(name)
    print(group.head())
```

This code will read the **CSV file** ‘my_file.csv’, skipping rows with missing values (empty cells) in any column. Then, it will group the data by the values in the columns ‘col1’ and ‘col2’, and print the first **5 rows** of each group to the console. These are just some examples of how Pandas can be used for **data wrangling**. You can combine these and other functions and methods in different ways to perform a wide range of tasks. The following codes are important for performing various operations on a Pandas **dataframe**. Here’s what each line of code does:

- **df[df[‘col1’] > 1]**: This filters the **dataframe df** to include only rows where the value in the ‘col1’ column is greater than 1.
- **df.sort_values(‘col1’)**: This sorts the **dataframe df** by the values in the ‘col1’ column, in ascending order.
- **df.sort_values([‘col1’, ‘col2’], ascending=[False, True])**: This sorts the **dataframe df** by the values in the ‘col1’ and ‘col2’ columns, in descending and ascending order, respectively.
- **df.duplicated()**: This identifies duplicate rows in the **dataframe df**, and returns a boolean mask indicating which rows are duplicates.
- **df[‘col1’].unique()**: This identifies unique values in the ‘col1’ column of the **dataframe df**, and returns an array of the unique values.

- **df = df.transpose()**: This transposes the **dataframe df**, swapping rows and columns.
- **df = df.drop('col1', axis=1)**: This drops the column with label 'col1' from the **dataframe df**.
- **clone = df.copy()**: This creates a shallow copy of the **dataframe df**, which is stored in the variable **clone**.

In this code **pd.concat()** function is used to concatenate the **df** and **df3** data frames horizontally, by setting the axis parameter to **1**. This will combine the two data frames by joining the rows with the same indices and adding the additional columns from **df3** to the resulting data frame.

In this code **df.merge()** function is used to merge the **df** and **df3** data frames using an inner join. This means that only the rows that have matching indices in both data frames will be included in the resulting data frame. This is equivalent to an inner join in **SQL**, where only the rows that have matching values in the specified columns will be included in the result. For example, if **df** and **df3** were defined as follows:

```
df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                   index=['A', 'B', 'C'],
                   columns=['col1', 'col2',
                           'col3'])

df3 = pd.DataFrame([[7], [8], [9]],
                    index=['A', 'B', 'C'],
                    columns=['col3'])
```

Then the result of **df.merge(df3)** would be:

	col1	col2	col3
0	1	2	7
1	4	5	8
2	7	8	9

Note that only the rows with indices 'A', 'B', and 'C' were included in the resulting data frame because these were the only rows that had matching indices in both **df** and **df3**.

In this code **df.merge()** function is used to merge the **df** and **df3** data frames using a left outer join. This means that all the rows from the **df** data frame will be included in the resulting data frame, even if there is no corresponding row in **df3** with a matching index. The additional columns from **df3** will be filled with null or missing values for the rows that do not have matching indices. For example, if **df** and **df3** were defined as follows:

```
df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                  index=['A', 'B', 'C'],
                  columns=['col1', 'col2',
                          'col3'])

df3 = pd.DataFrame([[7], [8], [9]],
                   index=['A', 'B', 'C'],
                   columns=['col3'])
```

Then the result of **df.merge(df3, how='left')** would be:

	col1	col2	col3
0	1	2	7
1	4	5	8
2	7	8	9

The result of **df.merge(df3, how='right')** would be:

	col1	col2	col3
0	1.0	2.0	7
1	4.0	5.0	8
2	7.0	8.0	9

The result of **df.merge(df3, how='outer')** would be:

	col1	col2	col3
0	1.0	2.0	7.0
1	4.0	5.0	8.0
2	7.0	8.0	9.0

And the result of **df.merge(df3, left_index=True, right_index=True)**

In this code, a user-defined function called **func()** is defined that takes a single argument **x** and returns the result of raising **2** to the power of **x**. This function is then applied to the **df** data frame using

the **df.apply()** function, which applies the function to each element in the data frame and returns a new data frame with the transformed values. For example, if **df** were defined as follows:

```
df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],  
                  index=['A', 'B', 'C'],  
                  columns=['col1', 'col2', 'col3'])
```

Then the result of **df.apply(func)** would be:

```
col1 col2 col3  
A    2    4    8  
B   16   32   64  
C  128  256 2048
```

Note that the **func()** function is applied to each element in the **df** data frame, and the resulting values are returned in a new data frame.

Arithmetics and statistics in Pandas

Pandas provide a wide range of functions for performing arithmetic and statistical operations on data frames. These functions can be used to perform operations such as **adding**, **subtracting**, **multiplying**, or **dividing** the values in a data frame by a constant value or another data frame. They can also be used to calculate summary statistics such as the mean, median, mode, standard deviation, and other statistical measures. To perform arithmetic operations on a data frame, you can use the **df.add()**, **df.sub()**, **df.mul()**, and **df.div()** functions, which perform addition, subtraction, multiplication, and division, respectively (Stepanek, H., 2020). For example, if **df1** and **df2** are two data frames, you can perform element-wise addition as follows:

```
df1.add(df2)
```

To calculate summary statistics for a data frame, you can use the **df.mean()**, **df.median()**, **df.mode()**, **df.std()**, and other similar functions. For example, to calculate the mean of each column in a data frame, you can use the **df.mean()** function as follows:

```
df.mean()
```

These functions can also be applied to specific columns or rows in a data frame by passing the appropriate indices or labels as arguments. For example, to calculate the mean of the values in the first column of a data frame, you could use the following code:

```
df['coll'].mean()
```

Pandas also provide a wide range of other functions for performing common statistical operations on data frames, such as calculating the minimum, maximum, and range of values in a data frame, or generating random samples from a data frame. You can find more information about these functions in the Pandas documentation.

In this code, **df + 10** expression adds **10** to all the values in the **df** data frame. This is equivalent to calling the **df.add()** function with the constant value **10** as the argument.

In this code, **df.sum()** function is used to calculate the sum of the values in each column of the **df** data frame. This function returns a new data frame with the same indices as **df**, but with a single column for each of the original columns in **df**, containing the sum of the values in that column.

In this code, **df.cumsum()** function is used to calculate the cumulative sum of the values in each column of the **df** data frame. This function returns a new data frame with the same indices as **df**, but with a single column for each of the original columns in **df**, containing the cumulative sum of the values in that column.

In this code, **df.mean()** function is used to calculate the meaning of the values in each column of the **df** data frame. This function returns a new data frame with the same indices as **df**, but with a single column for each of the original columns in **df**, containing the meaning of the values in that column.

In this code, **df.std()** function is used to calculate the standard deviation of the values in each column of the **df** data frame. This function returns a new data frame with the same indices as **df**, but with a single column for each of the original columns in **df**, containing the standard deviation of the values in that column.

In this code, **df['coll'].value_counts()** expression is used to count the number of unique values in the **coll** column of the **df** data frame.

This expression returns a **Series** object containing the counts of each unique value in the **coll** column.

In this code, **df.describe()** function is used to summarize the descriptive statistics for each column of the **df** data frame. (Stepanek, H., 2020)

This function returns a new data frame with several columns containing the count, mean, standard deviation, minimum, maximum, and other statistical measures for each of the columns in **df**. For example, if **df** were defined as follows:

```
df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                  index=['A', 'B', 'C'],
                  columns=['col1', 'col2', 'col3'])
```

Then the result of **df + 10** would be:

```
   coll  col2  col3
A    11    12    13
B    14    15    16
C    17    18    19
```

The result of **df.sum()** would be:

```
coll    12
col2    15
col3    18
dtype: int64
```

The result of **df.cumsum()** would be:

```
   coll  col2  col3
A     1     2     3
B     5     7     9
C    12    15    18
```

The result of **df.mean()** would be:

```
coll    4.0
col2    5.0
```

Hierarchical indexing in Pandas

In pandas, **hierarchical indexing** is a way to represent data with multiple levels of indexing on a **DataFrame**. This can be useful when working with large datasets that have multiple levels of hierarchy in the data. For example, if you have data on the population of coun-

tries, states, and cities, you can use **hierarchical indexing** to represent this data in a **DataFrame**. The first level of the index would be the country, the second level would be the state, and the third level would be the city. To create a **hierarchical index** on a **DataFrame**, you can use the `set_index()` method and specify the column or columns to use as the index. You can also use the `sort_index()` method to sort the index levels in a hierarchical **DataFrame**. Here is an example of how to create a **hierarchical index** on a **DataFrame**:

```
import pandas as pd

# Create a DataFrame with data on the population
df = pd.DataFrame({'country': ['SUA', 'SUA',
                              'SUA', 'France', 'France'],
                  'state': ['NY', 'CA', 'TX',
                           'Paris Region', 'Brittany'],
                  'city': ['New York', 'Los
Angeles', 'Houston', 'Paris', 'Rennes'],
                  'population': [8.4, 3.9, 2.3,
                                2.14, 0.2]})

# Set the country, state, and city columns as the
index
df = df.set_index(['country', 'state', 'city'])

# Sort the index levels
df = df.sort_index()

# View the DataFrame
print(df)
```

You can access data in a hierarchical **DataFrame** by using the `loc[]` method and specifying the index levels you want to access, in order. For example, to access the population data for the city of Paris in Paris Region, France, you would use the following code:

```
# Access population data for Paris
df.loc[('France', 'Paris Region', 'Paris'),
       'population']
```

This would return the population data for Paris: 2.14

Aggregation in Pandas

In pandas, **aggregation** refers to the process of combining data to get a single summary value for each group. This is typically done by applying an aggregation function to a group of data, such as summing the values in each group or calculating the average value of each group. Pandas provide several built-in functions for **aggregation**, including **sum()**, **mean()**, **min()**, and **max()**. You can use these functions with the **groupby()** method to perform **aggregation** on a **DataFrame**. Here is an example of how to use the **groupby()** and **mean()** methods to calculate the average population for each country in a **DataFrame**:

```
import pandas as pd

# Create a DataFrame with data on the population
df = pd.DataFrame({'country': ['USA', 'USA',
                                'USA', 'China', 'China'],
                   'state': ['NY', 'CA', 'TX',
                              'Sichuan', 'Yunnan'],
                   'city': ['New York', 'Los Angeles',
                             'Houston', 'Chengdu', 'Kunming'],
                   'population': [8.4, 3.9, 2.3,
                                  14.0, 5.1]})

# Group the data by country and calculate the
# average population
country_mean = df.groupby('country')['population']
               .mean()

# View the result
print(country_mean)
```

You can also use the **agg()** method to perform multiple aggregations on a **DataFrame** at once. This method takes a dictionary as input, where the keys are the column **names** and the values are the **aggregation** functions to apply to each column. Here is an example of how to use the **agg()** method to calculate the sum and mean of the population data for each country:

```
import pandas as pd

# Create a DataFrame with data on the population
df = pd.DataFrame({'country': ['USA', 'USA',
                                'USA', 'China', 'China'],
                   'state': ['NY', 'CA', 'TX',
                              'Sichuan', 'Yunnan'],
                   'city': ['New York', 'Los Angeles',
                             'Houston', 'Chengdu', 'Kunming'],
                   'population': [8.4, 3.9, 2.3,
                                  14.0, 5.1]})

# Group the data by country and calculate the sum
# and mean of the population
country_agg = df.groupby('country').agg({'population':
                                          ['sum', 'mean']})

# View the results
print(country_agg)
```

Data Export in Pandas

In the Python Pandas library, data export is a very useful feature that allows you to save your data as a file on your computer. There are several different types of files that you can use to export your data, including **CSV**, **Excel**, and **JSON** files. To export data from Pandas, you can use the `to_csv()`, `to_excel()`, or `to_json()` methods, depending on the type of file you want to create. Here is an example of how you might use the `to_csv()` method to export data from a Pandas **DataFrame** to a **CSV** file:

```
import pandas as pd

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

# Export the DataFrame to a CSV file
df.to_csv('data.csv')
```

In this example, we create a **DataFrame** with some sample data, and then use the `to_csv()` method to export the data to a **CSV file** named `data.csv`. This file will be saved in the current working directory, which is the directory where the Python script is located. You can also specify a different file path to save the **CSV file** to a different location on your computer.

Pivot and Pivot Table in Pandas

In the context of the Python Pandas library, a **pivot** refers to a transformation of a data set that rearranges the data into a different format. This can be useful for creating summaries of data, or for creating new data sets that are more useful for a specific task. A **pivot table** is a specific type of pivot that summarizes data by grouping the data by one or more columns and calculating one or more summary statistics for each group. This can be useful for quickly generating summaries of data, such as counts, sums, or averages, for different groups of data. Here is an example of how you might use the Pandas `pivot_table()` method to create a pivot table:

```
# Create a DataFrame with some sample data
df = pd.DataFrame({'A': [1, 2, 3, 1, 2, 3],
                   'B': [4, 5, 6, 5, 6, 7],
                   'C': [7, 8, 9, 8, 9, 10]})

# Create a pivot table that calculates the mean of
# 'C' for each unique combination of 'A' and 'B'
pivot_table = pd.pivot_table(df, values='C',
                              index=['A', 'B'], aggfunc='mean')

# Print the pivot table
print(pivot_table)
```

In this example, we create a **pivot table** that calculates the mean of the **'C'** column for each unique combination of **'A'** and **'B'** in the data set. In this **pivot table**, the rows are grouped by the unique combination of values in the **'A'** and **'B'** columns, and the mean of the **'C'** column is calculated for each group.

Visualization in Pandas

The Pandas library in Python includes several useful tools for **visualizing data**. Pandas use the **matplotlib library** to create graphs and plots of data, so you will need to **install matplotlib** if you don't already have it. Once you have **matplotlib installed**, you can use the **plot()** method on a Pandas **DataFrame** to create a basic line or scatter plot of the data (Yim, A., Chung, C., & Yu, A., 2018). For example, to create a line plot of the values in a **DataFrame**, you can use the following code:

```
import pandas as pd

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

# Plot the values in the DataFrame
df.plot()
```

This will create a line plot of the values in the **DataFrame**, with the 'A' column on the **x-axis** and the 'B' column on the **y-axis**. You can also specify which columns you want to use for the **x-** and **y-axes** by passing the column names to the **x** and **y** arguments of the **plot()** method, like this:

```
import pandas as pd

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

# Plot the 'A' and 'B' columns using a scatter plot
df.plot(x='A', y='B', kind='scatter')
```

This will create a scatter plot of the 'A' and 'B' columns, with the 'A' values on the **x-axis** and the 'B' values on the **y-axis**. In addition to the line and **scatter plots**, Pandas also provides tools for creating more advanced visualizations, such as **histograms**, **bar charts**, and **pie charts**. You can use the **plot.bar()**, **plot.hist()**, or **plot.pie()** methods to create these types of **plots** (Yim, A., Chung, C., & Yu, A., 2018). For ex-

ample, to create a **bar chart** of the values in a **DataFrame**, you can use the following code:

```
import pandas as pd

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

# Create a bar chart of the 'A' and 'B' columns
df.plot.bar()
```

This will create a **bar chart** with the 'A' values on the **x-axis** and the 'B' values on the **y-axis**.

To create a new figure or diagram using the **matplotlib** library in Python, you can use the **plt.figure()** method. This method creates a new figure or diagram, which is a container for the plot or plots that you want to create. Here is an example of how you might use the **plt.figure()** method to create a new figure:

```
import matplotlib.pyplot as plt

# Create a new figure
fig = plt.figure()

# Add a line plot to the figure
plt.plot([1, 2, 3], [4, 5, 6])
```

To create a new figure window containing a line plot of some data, you can use the **plt.figure()** and **plt.plot()** methods from the **matplotlib** library in Python. The **plt.figure()** method allows you to create a new figure, while the **plt.plot()** method adds a line plot to the figure. By default, this will create a new window with a line plot of the data. However, you can customize the appearance of the figure by specifying additional parameters in the **plt.figure()** method. For example, you can use the **figsize** parameter to specify the size of the figure in inches, and the **dpi** parameter to set the resolution of the figure in dots per inch. Here's an example of how you might use these parameters:

```
plt.figure(figsize=(10, 6), dpi=100)
plt.plot(x, y)
```

This will create a figure with a width of **10** inches and a height of **6** inches, at a resolution of **100** dpi. You can adjust these values to suit your needs. There are many other parameters that you can use to customize the appearance of the figure, such as the background color, the font size of the tick marks and labels, and the line width and style of the plot. You can find more information about these parameters in the **matplotlib** documentation.

```
import matplotlib.pyplot as plt

# Create a new figure with a specified size and
resolution
fig = plt.figure(figsize=(8, 6), dpi=200)

# Add a line plot to the figure
plt.plot([1, 2, 3], [4, 5, 6])
```

In this example, we create a new figure with a width of **8** inches and a height of **6** inches, and a resolution of **200** dots per inch (**dpi**). The resulting figure will have the specified size and resolution and will contain the line plot that we added.

To create a **scatter plot** using Pandas, you can use the **scatter()** method of a **dataframe** object. This method takes in the **x-axis** data and **y-axis** data as arguments, and it also has several optional parameters that you can use to customize the appearance of the plot. Here is an example of how to use the **scatter()** method to create a scatter plot in Pandas:

```
import pandas as pd

# Load the data into a dataframe
df = pd.read_csv("data.csv")

# Create the scatter plot
df.plot(x="X", y="Y", kind="scatter")
```

In this example, we load the data from a **CSV file** into a Pandas **dataframe** and then use the **scatter()** method to create a scatter plot of the data. The **x** and **y** parameters specify the columns of the **dataframe**

that contain the **x-axis** and **y-axis** data, respectively. The `kind` parameter specifies that we want to create a scatter plot. (VanderPlas, J. 2016). Once the plot is created, you can use the `show()` method to display it. You can also use the various other methods and attributes of the **matplotlib.pyplot** library to customize the appearance of the plot, such as changing the title, axis labels, and so on. Here is an example of how to customize the **scatter** plot using **matplotlib.pyplot** library:

```
import matplotlib.pyplot as plt

# Customize the plot
plt.title("Scatter Plot of X and Y")
plt.xlabel("X")
plt.ylabel("Y")

# Show the plot
plt.show()
```

In this example, we use the `title()`, `xlabel()`, and `ylabel()` methods to add a **title**, **x-axis label**, and **y-axis label** to the plot, respectively. Then, we use the `show()` method to display the plot. **Scatter plots** are a powerful and versatile tool for visualizing the relationship between two quantitative variables. Using the `scatter()` method in Pandas, you can easily create high-quality scatter plots that can help you gain insights into your data.

To create a **bar plot** using Pandas, you can use the `plot.bar()` method of a **dataframe** object. This method takes in the **x-axis** data and **y-axis** data as arguments, and it also has several optional parameters that you can use to customize the appearance of the plot. Here is an example of how to use the `plot.bar()` method to create a **bar plot** in Pandas:

```
import pandas as pd

# Load the data into a dataframe
df = pd.read_csv("data.csv")

# Create the bar plot
df.plot.bar(x="X", y="Y")
```

In this example, we load the data from a **CSV file** into a Pandas **dataframe** and then use the **plot.bar()** method to create a **bar plot** of the data. The **x** and **y** parameters specify the columns of the **dataframe** that contain the **x-axis** and **y-axis data**, respectively. Once the plot is created, you can use the **show()** method to display it. You can also use the various other methods and attributes of the **matplotlib.pyplot** library to customize the appearance of the plot, such as changing the title, axis labels, and so on. Here is an example of how to customize the **bar plot** using **matplotlib.pyplot** library:

```
import matplotlib.pyplot as plt

# Customize the plot
plt.title("Bar Plot of X and Y")
plt.xlabel("X")
plt.ylabel("Y")

# Show the plot
plt.show()
```

In this example, we use the **title()**, **xlabel()**, and **ylabel()** methods to add a title, **x-axis** label, and **y-axis** label to the plot, respectively. Then, we use the **show()** method to display the plot. **Bar plots** are a useful tool for visualizing the relationship between a categorical variable and a numerical variable. Using the **plot.bar()** method in Pandas, you can easily create high-quality bar plots that can help you gain insights into your data.

To create a **line plot** using Pandas, you can use the **plot.line()** method of a **dataframe** object. This method takes in the **x-axis** data and **y-axis** data as arguments, and it also has several optional parameters that you can use to customize the appearance of the plot. Here is an example of how to use the **plot.line()** method to create a **line plot** in Pandas:

```
import pandas as pd

# Load the data into a dataframe
df = pd.read_csv("data.csv")

# Create the line plot
df.plot.line(x="X", y="Y")
```

In this example, we load the data from a **CSV file** into a Pandas **dataframe** and then use the **plot.line()** method to create a **line plot** of the data. The **x** and **y** parameters specify the columns of the **dataframe** that contain the **x-axis** and **y-axis** data, respectively. Here is an example of how to customize the **line plot** using **matplotlib.pyplot** library:

```
import matplotlib.pyplot as plt

# Customize the plot
plt.title("Line Plot of X and Y")
plt.xlabel("X")
plt.ylabel("Y")

# Show the plot
plt.show()
```

In this example, we use the **title()**, **xlabel()**, and **ylabel()** methods to add a title, **x-axis** label, and **y-axis** label to the plot, respectively. Then, we use the **show()** method to display the plot. Line plots are a useful tool for visualizing the relationship between two quantitative variables over time. Using the **plot.line()** method in Pandas, you can easily create high-quality line plots that can help you gain insights into your data.

To create a **box plot** using Pandas, you can use the **plot.box()** method of a **dataframe** object. This method takes in the **y-axis** data as an argument, and it also has several optional parameters that you can use to customize the appearance of the plot. Here is an example of how to use the **plot.box()** method to create a **box plot** in Pandas:

```
import pandas as pd

# Load the data into a dataframe
df = pd.read_csv("data.csv")

# Create the box plot
df.plot.box(y="Y")
```

In this example, we load the data from a **CSV file** into a Pandas **dataframe** and then use the **plot.box()** method to create a **box plot** of the data. The **y** parameter specifies the column of the **dataframe** that contains the **y-axis** data. Here is an example of how to customize the **box plot** using **matplotlib.pyplot** library:

```
import matplotlib.pyplot as plt

# Customize the plot
plt.title("Box Plot of Y")
plt.ylabel("Y")

# Show the plot
plt.show()
```

In this example, we use the **title()** and **ylabel()** methods to add a title and **y-axis** label to the plot, respectively. Then, we use the **show()** method to display the plot.

To create a **histogram over one column** in Pandas, you can use the **plot.hist()** method of a **dataframe** object. This method takes in the **y-axis** data as an argument, and it also has several optional parameters that you can use to customize the appearance of the plot. Here is an example of how to use the **plot.hist()** method to create a **histogram over one column** in Pandas:

```
import pandas as pd

# Load the data into a dataframe
df = pd.read_csv("data.csv")

# Create the histogram
df.plot.hist(y="Y")
```

In this example, we load the data from a **CSV file** into a Pandas **dataframe**, and then use the **plot.hist()** method to create a **histogram** of the data. The **y** parameter specifies the column of the **dataframe** that contains the **y-axis** data. Once the **histogram** is created, you can use the **show()** method to display it. You can also use the various other methods and attributes of the **matplotlib.pyplot** library to cus-

tomize the appearance of the plot, such as changing the title, axis labels, and so on. Here is an example of how to customize the **histogram** using the **matplotlib.pyplot** library:

```
import matplotlib.pyplot as plt

# Customize the plot
plt.title("Histogram of Y")
plt.xlabel("Y")
plt.ylabel("Frequency")

# Show the plot
plt.show()
```

In this example, we use the **title()**, **xlabel()**, and **ylabel()** methods to add a title, **x-axis** label, and **y-axis** label to the plot, respectively. Then, we use the **show()** method to display the plot. **Histograms** are a useful tool for visualizing the distribution of a quantitative variable. Using the **plot.hist()** method in Pandas, you can easily create high-quality **histograms** that can help you gain insights into your data.

To create a **pie chart** using pandas, you can use the **plot.pie()** method of a **dataframe**. Here is an example of how you can create a **pie chart** to visualize the distribution of a categorical variable in a **dataframe**:

```
import pandas as pd
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv('data.csv')

# Select a categorical column
data = df['column_name']

# Create a pie chart
plot = data.plot.pie(figsize=(6, 6))

# Display the chart
plt.show()
```


You can also customize the pie chart by specifying various parameters in the **plot.pie()** method. For example, you can set the title, colors, and legend of the chart.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv('data.csv')

# Select a categorical column
data = df['column_name']

# Create a pie chart
plot = data.plot.pie(figsize=(6, 6),
                    title='Pie Chart Title',
                    colors=['#ff9999', '#66b3ff', '#99ff99'],
                    legend=True)

# Display the chart
plt.show()
```

To set **tick marks** on a plot in **matplotlib**, you can use the **xticks()** and **yticks()** functions. Here is an example of how you can use these functions to set **tick marks** on the **x-axis** and **y-axis** of a plot:

```
import matplotlib.pyplot as plt

# Set x-axis tick marks
labels = ['A', 'B', 'C', 'D']
positions = [1, 2, 3, 4]
plt.xticks(positions, labels)

# Set y-axis tick marks
labels = ['E', 'F', 'G', 'H']
positions = [1, 2, 3, 4]
plt.yticks(positions, labels)

# Display the plot
plt.show()
```

You have the option to customize the appearance of tick marks by specifying additional parameters such as **font size**, **rotation**, and **color**.

```
import matplotlib.pyplot as plt

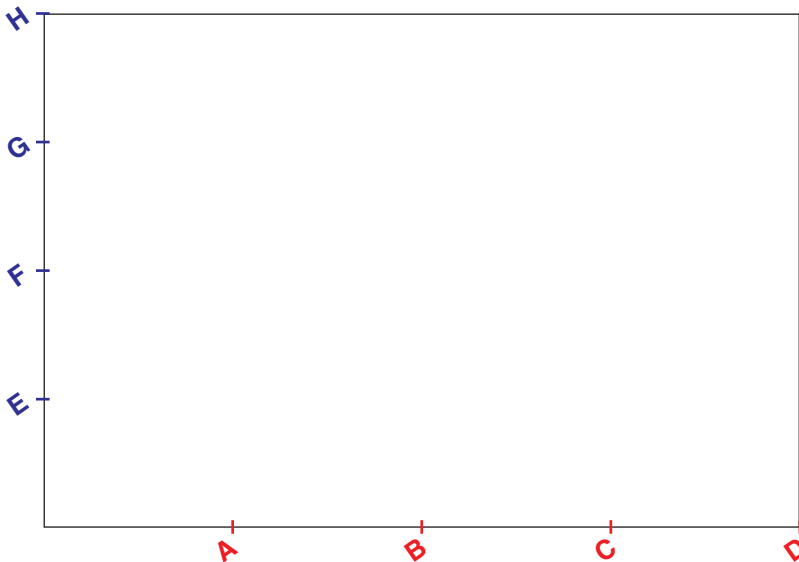
# Set x-axis tick marks
labels = ['A', 'B', 'C', 'D']
positions = [1, 2, 3, 4]
plt.xticks(positions, labels, fontsize=14,
           rotation=45, color='red')

# Set y-axis tick marks
labels = ['E', 'F', 'G', 'H']
positions = [1, 2, 3, 4]
plt.yticks(positions, labels, fontsize=14,
           rotation=45, color='blue')

# Display the plot
plt.show()
```

The graphical result is:

Fig. 1 Appearance of the **tick marks** by specifying additional parameters



To **label a diagram** and its axes in **matplotlib**, you can use the **title()**, **xlabel()**, and **ylabel()** functions. Here is an example of how you can use these functions to **label a diagram** and its axes:

```
import matplotlib.pyplot as plt

# Set the title of the plot
plt.title('Correlation')

# Set the label for the x-axis
plt.xlabel('Nunstu?ck')

# Set the label for the y-axis
plt.ylabel('Slotermeyer')

# Display the plot
plt.show()
```

You can also customize the appearance of the **labels** by specifying additional parameters, such as the **fontsize**, **color**, and **fontweight**.

```
import matplotlib.pyplot as plt

# Set the title of the plot
plt.title('Correlation', fontsize=16, color='red',
fontweight='bold')

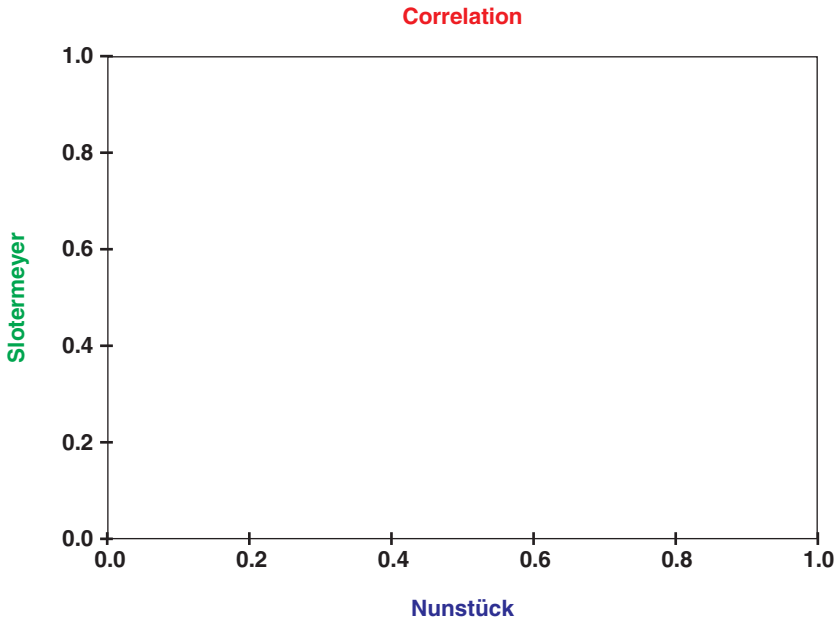
# Set the label for the x-axis
plt.xlabel('Nunstu?ck', fontsize=14, color='blue',
fontweight='bold')

# Set the label for the y-axis
plt.ylabel('Slotermeyer', fontsize=14,
color='green', fontweight='bold')

# Display the plot
plt.show()
```

The graphical result is:

Fig. 2 Customize the appearance of the **labels** by specifying additional parameters



To **save** the most recent plot in **matplotlib**, you can use the **savefig()** function. Here is an example of how you can use the **savefig()** function to save the most recent plot as a **PNG file**:

```
import matplotlib.pyplot as plt

# Create a plot
plt.plot([1, 2, 3, 4])

# Save the plot as a PNG file
plt.savefig('plot.png')
```

You can also specify the resolution of the **saved** image by setting the **dpi** (dots per inch) parameter.

```
import matplotlib.pyplot as plt

# Create a plot
plt.plot([1, 2, 3, 4])

# Save the plot as a high-resolution PNG file
plt.savefig('plot.png', dpi=300)
```

You can also **save** the plot as a vector graphics file, such as an **SVG file**, by setting the file extension appropriately.

```
import matplotlib.pyplot as plt

# Create a plot
plt.plot([1, 2, 3, 4])

# Save the plot as an SVG file
plt.savefig('plot.svg')
```

★ *A few exercises with Pandas*

Here are some exercises you can try to practice working with **pandas**:

- a) Load a **CSV file** into a **pandas dataframe** and display the first few rows.
- b) Select a subset of the data by selecting a specific column or a group of columns.
- c) Filter the data by selecting rows that meet certain criteria.
- d) Group the data by a categorical variable and compute summary statistics for each group.
- e) Join two **dataframes** by a common key and display the resulting **dataframe**.
- f) **Pivot** the data to create a new **dataframe** with a different structure.
- g) Visualize the data using a **bar chart** or a **scatter plot**.
- h) Write the data in a new **CSV file** or an Excel spreadsheet.

- i) Import a **CSV** file into a Pandas DataFrame and explore the data. You can use the `read_csv()` function to import the data, and then use the `head()` and `describe()` methods to get a feel for the data.
- j) Select a subset of the data using indexing and boolean indexing. You can use the `[]` operator to select columns, and the `loc[]` and `iloc[]` attributes to select rows based on their position or label.
- k) Clean and transform the data using Pandas functions. You can use functions like `dropna()`, `fillna()`, and `replace()` to remove or replace missing values, and `apply()` to apply a function to every element in a column.
- l) Aggregate and summarize the data using groupby and pivot tables. You can use the `groupby()` function to group the data by one or more columns, and then use aggregation functions like `mean()`, `sum()`, and `size()` to compute statistics for each group. You can also use the `pivot_table()` function to create a pivot table that summarizes the data (McKinney, W., & Team, P. D., 2015).

NumPy, a perfect tool for working with Arrays

NumPy is a Python library that is widely used for working with **arrays**. It provides several functions and methods that allow you to manipulate and analyze **arrays** efficiently. Some of the key features of **NumPy** are:

- **N-dimensional arrays:** **NumPy** provides support for **arrays** with any number of dimensions. This makes it easy to work with **multi-dimensional data**, such as **images** or **matrices**.
- **Array operations:** **NumPy** provides several functions that allow you to perform mathematical operations on **arrays**, such as element-wise **addition**, **multiplication**, and **exponentiation**.
- **Broadcasting:** **NumPy** allows you to perform operations on **arrays** with different shapes if they are compatible. This is known as **broadcasting**, and it allows you to write concise and efficient code.
- **Linear algebra:** **NumPy** includes functions for performing linear algebra operations, such as **matrix multiplication**, **singular value decomposition**, and **eigenvalue decomposition**.

- **Random number generation:** NumPy provides functions for generating **random numbers** and sampling from various probability distributions (Johansson, R., 2015).

NumPy is an essential tool for working with **arrays** in Python, and it is widely used in scientific computing, data analysis, and machine learning. **NumPy arrays** are multi-dimensional **arrays** of a homogeneous data type, which means that all elements in an **array** must have the same data type. **NumPy arrays** are more efficient and more flexible than Python's built-in lists and tuples, as they allow you to perform element-wise operations and mathematical functions on entire **arrays**, rather than having to loop over the elements of the **array** yourself. Here is an example of how you can create and manipulate NumPy arrays:

```
import numpy as np

# Create a 1-dimensional array
a = np.array([1, 2, 3, 4])
print(a)

# Create a 2-dimensional array
b = np.array([[1, 2], [3, 4]])
print(b)

# Create an array of all zeros
c = np.zeros((3, 3))
print(c)

# Create an array of all ones
d = np.ones((2, 2))
print(d)

# Create an array of evenly spaced values
e = np.arange(0, 10, 2)
print(e)

# Create an array of random values
f = np.random.rand(3, 3)
print(f)
```

```
# Access and modify elements of an array
print(a[2])
a[2] = 10
print(a)

# Perform mathematical operations on arrays
print(a + e)
print(b * d)
```

Here are some other creative examples of using the `np.array()` function:

```
import numpy as np

# Create a 1-dimensional array
a = np.array([1, 2, 3])
print(a)

# Create a 2-dimensional array
b = np.array([(1.5, 2, 3), (4, 5, 6)],
             dtype=float)
print(b)

# Create a 3-dimensional array
c = np.array([[[1.5, 2, 3], [4, 5, 6]], [[3, 2, 1], [4, 5, 6]]], dtype=float)
print(c)
```

The `np.array()` function takes a list of values as its first argument, and it returns an **array** with the specified shape and data type. You can specify the data type of the **array** by setting the `dtype` parameter. If you do not specify the data type, **NumPy** will try to infer it from the input data. **NumPy** provides several functions for creating arrays with initial placeholders. Here are some examples of how you can use these functions to create **arrays**:


```
import numpy as np

# Create an array of zeros
a = np.zeros((3, 4))
print(a)

# Create an array of ones
b = np.ones((2, 3, 4), dtype=np.int16)
print(b)

# Create an array of evenly spaced values
c = np.arange(10, 25, 5)
print(c)

# Create an array of evenly spaced values over a
# specified interval
d = np.linspace(0, 2, 9)
print(d)

# Create an array of a specified shape and value
e = np.full((2, 2), 7)
print(e)

# Create an identity matrix
f = np.eye(2)
print(f)

# Create an array of random values
g = np.random.random((2, 2))
print(g)

# Create an array with uninitialized values
h = np.empty((3, 2))
print(h)
```

NumPy provides functions for saving and loading **arrays** to and from disk. To **save** a single **array** to disk, you can use the **np.save()** function. This function takes the name of the file and the array as ar-

guments, and it saves the array to a file with the specified name and the **.npy** extension.

```
import numpy as np

# Create an array
a = np.array([1, 2, 3])

# Save the array to a file
np.save('my_array', a)
```

To save multiple **arrays** to a single file, you can use the **np.savez()** function. This function takes the name of the file and the **arrays** as arguments, and it **saves** the **arrays** to a file with the specified name and the **.npz** extension.

```
import numpy as np

# Create two arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Save the arrays to a file
np.savez('array.npz', a, b)
```

To load an **array** from a file, you can use the **np.load()** function. This function takes the name of the file as an argument and returns the **array** stored in the file.

```
import numpy as np

# Load an array from a file
a = np.load('my_array.npy')
```

NumPy provides functions for saving and loading **arrays** to and from **text files**. To load an **array** from a text file, you can use the **np.loadtxt()** function. This function takes the name of the file as an argument and returns the array stored in the file (Johansson, R., 2015). By default, the **loadtxt()** function expects the data to be separated by whitespace.

```
import numpy as np

# Load an array from a text file
a = np.loadtxt('my_file.txt')
```

To import an **array** from a file with a different delimiter, like a **CSV** file, you can utilize **np.genfromtxt()**. This function requires the file's name and delimiter as arguments and returns the **array** stored in the file.

```
import numpy as np

# Load an array from a CSV file
a = np.genfromtxt('my_file.csv', delimiter=',')
```

To save an **array** to a **text file**, you can use the **np.savetxt()** function. This function takes the name of the file and the **array** as arguments, and it saves the array to a file with the specified name. By default, the **savetxt()** function separates the data with whitespace, but you can specify a different delimiter using the **delimiter** parameter.

```
import numpy as np

# Create an array
a = np.array([1, 2, 3])

# Save the array to a text file
np.savetxt('myarray.txt', a, delimiter=' ')
```

NumPy provides several functions and attributes for **inspecting arrays**. To get the shape of an **array**, you can use the **shape** attribute. The shape of an **array** is a **tuple** that indicates the size of the **array** along each dimension.

```
import numpy as np

# Create an array
a = np.array([[1, 2, 3], [4, 5, 6]])

# Get the shape of the array
print(a.shape)
```

To get the length of an **array** along a specific dimension, you can use the **len()** function. For example, to get the number of rows in a 2-dimensional **array**, you can use **len(a)**.

```
import numpy as np

# Create an array
a = np.array([[1, 2, 3], [4, 5, 6]])

# Get the number of rows in the array
print(len(a))
```

To get the number of dimensions of an **array**, you can use the **ndim** attribute.

```
import numpy as np

# Create an array
a = np.array([[1, 2, 3], [4, 5, 6]])
```

[hints!] NumPy provides support for a wide range of data types, including **integer**, **floating-point**, **complex**, **boolean**, and **string** types. Here are some examples of common data types in NumPy:

- **np.int64**: 64-bit integer data type
- **np.float32**: 32-bit floating-point data type
- **np.complex**: Complex data type (real and imaginary parts)
- **np.bool**: Boolean data type (True or False)
- **np.object**: Object data type (can store any Python object)
- **np.string_**: Fixed-length string data type
- **np.unicode_**: Fixed-length Unicode string data type

You can specify the data type of an **array** when you create it using the **dtype** parameter.

```
import numpy as np

# Create an array with a specific data type
a = np.array([1, 2, 3], dtype=np.float32)
print(a)

# Create an array of complex numbers
b = np.array([1+2j, 3+4j], dtype=np.complex)
print(b)

# Create an array of booleans
c = np.array([True, False, True], dtype=np.bool)
print(c)

# Create an array of strings
d = np.array(['a', 'b', 'c'], dtype=np.string_)
print(d)

# Create an array of Unicode strings
```

Matrix Manipulation in Numpy

NumPy provides several functions and methods for manipulating **matrices**. Here are some examples:

- **Transposing a matrix:** To transpose a **matrix**, you can use the **T** attribute. This attribute returns a new **matrix** with the rows and columns of the original **matrix** swapped.

```
import numpy as np

# Create a matrix
A = np.array([[1, 2, 3], [4, 5, 6]])

# Transpose the matrix
A_transposed = A.T

print(A_transposed)
```

- **Inverting a matrix:** To invert a **matrix**, you can use the **inv()** function. This function returns the inverse of the **matrix** if it exists.

```
import numpy as np

# Create a matrix
A = np.array([[1, 2], [3, 4]])

# Invert the matrix
A_inverted = np.linalg.inv(A)

print(A_inverted)
```

- **Solving a system of linear equations:** To solve a **system of linear equations**, you can use the **solve()** function from the **linalg** module. This function takes a **matrix** of coefficients and a vector of constants as arguments, and it returns the solution to the system of equations. Here is an example of how you can use the **solve()** function to **solve a system of linear equations**:

```
import numpy as np

# Create a matrix of coefficients
A = np.array([[1, 2], [3, 4]])

# Create a vector of constants
b = np.array([5, 6])

# Solve the system of equations
x = np.linalg.solve(A, b)

print(x)
```

In this example, the system of equations is $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where **A** is the **matrix** of coefficients, **x** is the vector of variables, and **b** is the vector of constants. The **solve()** function returns the solution **x**, which is a vector of values that satisfy the system of equations.

Array Mathematics in Numpy

NumPy provides several functions and operators for performing arithmetic operations on **arrays**. Here are some examples of how you can use **NumPy** to perform arithmetic operations on **arrays**:

Subtraction: To subtract one **array** from another, you can use the **-** operator or the **subtract()** function.

```
import numpy as np

# Create two arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([[0.5, 0], [3, 3]])

# Subtract the arrays using the - operator
c = a - b
print(c)

# Subtract the arrays using the subtract() function
d = np.subtract(a, b)
print(d)
```

Addition: To add two **arrays**, you can use the **+** operator or the **add()** function.

```
import numpy as np

# Create two arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([[0.5, 0], [3, 3]])

# Add the arrays using the + operator
c = a + b
print(c)

# Add the arrays using the add() function
d = np.add(a, b)
print(d)
```

Division: To divide one **array** by another, you can use the `/` operator or the **divide()** function.

```
import numpy as np

# Create two arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([[0.5, 0], [3, 3]])

# Divide the arrays using the / operator
c = a
```

NumPy provides several functions for computing **aggregate statistics** on **arrays**, such as **sums**, **means**, **medians**, and **standard deviations**. Here are some examples of how you can use **NumPy** to compute **aggregate statistics** on **arrays**:

Sum: To compute the **sum** of all elements in an **array**, you can use the **sum()** function.

```
import numpy as np

# Create an array
a = np.array([[1, 2], [3, 4]])

# Compute the sum of all elements
s = a.sum()
print(s)
```

Minimum: To compute the **minimum** value in an **array**, you can use the **min()** function.

```
import numpy as np

# Create an array
a = np.array([[1, 2], [3, 4]])

# Compute the minimum value
m = a.min()
print(m)
```


Maximum: To compute the **maximum** value in an **array**, you can use the **max()** function. You can use the **axis** parameter to specify the dimension along which the **maximum** value is to be computed.

```
import numpy as np

# Create an array
b = np.array([[1, 2], [3, 4]])

# Compute the maximum value along the rows
m = b.max(axis=0)
print(m)

# Compute the maximum value along the columns
m = b.max(axis=1)
print(m)
```

Cumulative sum: To compute the **cumulative sum** of the elements in an **array**, you can use the **cumsum()** function. You can use the **axis** parameter to specify the dimension along which the **cumulative sum** is to be computed.

```
import numpy as np

# Create an array
b = np.array([[1, 2], [3, 4]])

# Compute the cumulative sum along the rows
s = b.cumsum(axis=0)
print(s)

# Compute the cumulative
```

In **NumPy**, you can create copies of **arrays** using the **view()** and **copy()** functions.

View: To create a **view** of an **array**, you can use the **view()** function. A **view** is a new **array** that shares the same data as the original **array**, but it has a different shape, stride, and data type. Any changes made to the **view** will be reflected in the original **array**.

```
import numpy as np

# Create an array
a = np.array([[1, 2], [3, 4]])

# Create a view of the array
b = a.view()

# Modify the view
b[0, 0] = 10

# Print the original array
print(a)
```

Copy: To create a **deep copy** of an **array**, you can use the **copy()** function. A **deep copy** creates a new **array** with its **copy** of the data, so it is independent of the original **array**. Any changes made to the **copy** will not be reflected in the original **array**.

```
import numpy as np

# Create an array
a = np.array([[1, 2], [3, 4]])

# Create a deep copy of the array
b = a.copy()

# Modify the copy
b[0, 0] = 10

# Print the original array
print(a)
```

NumPy's `sort()` function allows you to rearrange the elements of an **array**. You can specify the dimension along which you want to sort the **array** using the `axis` parameter. The following example demonstrates how to use the `sort()` function to sort an **array**:

```
import numpy as np

# Create an array
a = np.array([[3, 2], [1, 4]])

# Sort the array
a.sort()
print(a)

# Sort the array along the rows
a.sort(axis=0)
print(a)
```

NumPy also provides several functions for manipulating **arrays**, such as **reshaping**, **repeating**, and **joining arrays**. Here are some examples of these functions:

Reshaping: To reshape an **array**, you can use the `reshape()` function. This function returns a new **array** with the same data as the original **array**, but with a different shape.

```
import numpy as np

# Create an array
a = np.array([[1, 2], [3, 4]])

# Reshape the array
b = a.reshape((4, 1))
print(b)
```

Repeating: To repeat the elements of an **array**, you can use the **repeat()** function. This function returns a new **array** with the elements of the original **array** repeated a specified number of times.

```
import numpy as np

# Create an array
a = np.array([1, 2, 3])

# Repeat the elements of the array
b = np.repeat(a, 3)
print(b)
```

Joining: To join two or more **arrays**, you can use the **concatenate()** function. This function returns a new **array** that contains the elements of the original **arrays**, concatenated along a specified axis.

```
import numpy as np

# Create two arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Join the arrays
```

Array Manipulation

In the NumPy library, the **transpose** function returns a new **array** with axes **transposed**. It takes an **array** as input and returns the **transposed array**. For example:

```
import numpy as np

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

print(a)
# Output: [[1 2 3]
#          [4 5 6]]

b = np.transpose(a)
```

```
print(b)
# Output: [[1 4]
#          [2 5]
#          [3 6]]
```

The **T** attribute of an **array** is also a way to get the **transposed array**. It is a convenient shorthand for calling the **transpose** function. For example:

```
import numpy as np

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

print(a)
# Output: [[1 2 3]
#          [4 5 6]]

b = a.T

print(b)
# Output: [[1 4]
#          [2 5]
#          [3 6]]
```

So in the code you provided, **i** is assigned the transposed **array** of **b**, and then **i.T** is the **transposed array** of **i**, which is the same as the original **array b**.

There are several ways to change the shape of an **array** in **NumPy**. One way is to use the **ravel** function, which returns a flattened version of the input **array**.

```
import numpy as np

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

print(a)
# Output: [[1 2 3]
#          [4 5 6]]

b = a.ravel()

print(b)
# Output: [1 2 3 4 5 6]
```

Another way to change the shape of an **array** is to use the **reshape** function, which returns a new **array** with a different shape. The new shape can be specified as a **tuple** of **integers**. For example:

```
import numpy as np

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

print(a)
# Output: [[1 2 3]
#          [4 5 6]]

b = a.reshape(3, 2)

print(b)
# Output: [[1 2]
#          [3 4]
#          [5 6]]
```

In the code you provided, **b.ravel()** returns a flattened version of the **array b**, and **g.reshape(3, -2)** returns a new **array** with shape **(3, -2)**, where the second dimension is inferred from the size of the **array** and the first dimension.

NumPy provides several functions for **adding** and **removing** elements from an **array**. The **resize** function can be used to change the shape and size of an **array** in place. For example:

```
import numpy as np

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

print(a)
# Output: [[1 2 3]
#          [4 5 6]]

a.resize((2, 6))

print(a)
# Output: [[1 2 3 0 0 0]
#          [4 5 6 0 0 0]]
```

The **append** function can be used to **append** values to the end of an **array**. For example:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = np.append(a, b)

print(c)
# Output: [1 2 3 4 5 6]
```

The **insert** function can be used to insert a value into an **array** at a specified index. For example:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.insert(a, 1, 5)

print(b)
# Output: [1 5 2 3]
```

The **delete** function can be used to **delete** elements from an **array** based on an index or a list of indices. For example:

```
import numpy as np

a = np.array([1, 2, 3, 4, 5, 6])
b = np.delete(a, [1, 3])

print(b)
# Output: [1 3 5 6]
```

NumPy provides several functions for combining **arrays**, including **concatenate**, **vstack**, and **hstack**. The **concatenate** function can be used to **concatenate** two or more **arrays** along a specified axis. The **axis** parameter specifies the axis along which the **arrays** are **concatenated**. For example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[7, 8, 9], [10, 11, 12]])

c = np.concatenate((a, b), axis=0)

print(c)
# Output: [[1 2 3]
#          [4 5 6]
#          [7 8 9]
#          [10 11 12]]
```

The **vstack** function can be used to stack **arrays** vertically, i.e., to **concatenate** them along the first axis. It is a convenient shorthand for calling **concatenate** with **axis=0**. For example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[7, 8, 9], [10, 11, 12]])

c = np.vstack((a, b))

print(c)
# Output: [[1 2 3]
#          [4 5 6]
#          [7 8 9]
#          [10 11 12]]
```

The **hstack** function can be used to **stack arrays horizontally**, i.e., to concatenate them along the second axis. It is a convenient shorthand for calling **concatenate** with **axis=1**. For example:


```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[7, 8, 9], [10, 11, 12]])

c = np.hstack((a, b))

print(c)
# Output: [[1 2 3 7 8 9]
#          [4 5 6 10 11 12]]
```

NumPy provides several functions and techniques for **splitting** and **subsetting arrays**. The **hsplit** function can be used to split an **array horizontally**, i.e., to divide it into **subarrays** along its second axis (**axis 1**). It takes an **array** and the number of **subarrays** as input and returns a list of **subarrays**. For example:

```
import numpy as np

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

b = np.hsplit(a, 3)

print(b)
# Output: [array([[1],
#                [4]]),
#          array([[2],
#                [5]]),
#          array([[3],
#                [6]])]
```

The **vsplit** function can be used to split an **array** vertically, i.e., to **divide** it into **subarrays** along its first axis (**axis 0**). It takes an **array** and the number of **subarrays** as input and returns a list of **subarrays**. For example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9],
              [10, 11, 12]])

b = np.vsplit(a, 2)

print(b)
# Output: [array([[1, 2, 3],
#                [4, 5, 6]]),
#          array([[7, 8, 9],
#                [10, 11, 12]])]
```

To access a specific element in an **array**, you can use **indexing**. For example:

```
import numpy as np

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

b = a[1, 2]

print(b)
# Output: 6
```

To access a **slice** of an **array**, you can use **slicing**. **Slicing** allows you to select a range of elements along a specified axis. For example:

```
import numpy as np

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

b = a[0:2]

print(b)
# Output: [[1 2 3]
#          [4 5 6]]
```

Boolean indexing allows you to select elements from an **array** based on a **boolean** condition. For example:

```
import numpy as np

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

b = a[a < 2]

print(b)
# Output: [1]
```

★ *Exercises with NumPy*

Here are some exercises that can help you learn and practice **NumPy** in Python:

- a) Create a NumPy array with 5 random integers between 0 and 10, and print it.
- b) Create a NumPy array with the values 10, 20, 30, 40, and 50, and print its shape.
- c) Create a 2D NumPy array with the values 1, 2, 3, 4, 5, 6, 7, and 8, and print its shape and size.
- d) Create a 3D NumPy array with the values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12, and print its shape, size, and number of dimensions.
- e) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and slice it to get a subarray with the values 3 and 4.
- f) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use boolean indexing to select all elements that are less than or equal to 3.
- g) Create two NumPy arrays with the values 1, 2, 3, and 4, and use the concatenate function to concatenate them along the second axis.
- h) Create two NumPy arrays with the values 1, 2, 3, and 4, and use the vstack function to stack them vertically.
- i) Create two NumPy arrays with the values 1, 2, 3, and 4, and use the hstack function to stack them horizontally.

- j) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `hsplit` function to split it into two subarrays along the first axis.
- k) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `vsplit` function to split it into two subarrays along the second axis.
- l) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `ravel` function to flatten it into a 1D array.
- m) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `reshape` function to change its shape to (2, 3).
- n) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `resize` function to change its shape and size to (4, 2).
- o) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `append` function to append the value 8 to the end of it.
- p) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `insert` function to insert the value 8 at index 4.
- q) Create a NumPy array with the values 1, 2, 3, 4, 5, and 6, and use the `delete` function to delete the element at index 5. (Vander-Plas, J. 2016)

Let's delve into DataVisualization in Python

There are many tools available in Python for data visualization, including:

Matplotlib: This is a powerful library for creating static, animated, and interactive visualizations in Python. It is highly customizable, allowing you to specify the appearance and behavior of your plots in detail.

Seaborn: This library is built on top of Matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics. It is particularly useful for exploring and visualizing statistical relationships in datasets.

Plotly: An interactive browser-based plotting library, this platform enables the creation of numerous visualizations like bar plots, scatter plots, and box plots.

Bokeh: This library is focused on creating interactive visualizations that can be easily embedded in web pages. It is particularly well-suited for visualizing large and complex datasets.

Altair: This is a declarative visualization library that allows you to build visualizations by specifying the mapping between data and visual encoding. It is particularly well-suited for quickly prototyping visualizations.

It is important to choose the right tool for the job, depending on the needs of your project. For example, if you need to create static plots for publication, **Matplotlib** or **Seaborn** might be the best choice. If you need to create interactive visualizations for the web, **Plotly** or **Bokeh** might be more appropriate.

[**hints!**] Data visualization is an important skill for data scientists and communication specialists because it allows you to effectively communicate the insights and findings of your data analysis to a wider audience. By creating visualizations of your data, you can make complex information more easily understood and more engaging to your audience. In addition, data visualization is a powerful tool for exploring and understanding your data. It allows you to quickly identify patterns, trends, and outliers in your data, which can help you to form hypotheses and generate ideas for further analysis. Python has several powerful libraries for creating data visualizations, including Matplotlib, Seaborn, and Plotly. These libraries provide a wide range of visualization types and styles and make it easy to customize and interact with your visualizations. Overall, learning data visualization with Python can be an asset for both data scientists and communication specialists, as it can help you to effectively communicate the insights of your data analysis and better understand your data.

Data Visualization with Matplotlib

Matplotlib is a powerful library for creating static, animated, and interactive **2D plots** in Python. It allows you to create a wide range of visualizations, including **line plots**, **scatter plots**, **bar plots**, **error bars**, **box plots**, **histograms**, and more. To use **Matplotlib**, you will first

need to **install** it using **pip install matplotlib**. Then, you can import the **pyplot** module from **Matplotlib** and use it to create and customize your plots. Here is a simple example of how to use **Matplotlib** to create a line plot:

```
import matplotlib.pyplot as plt

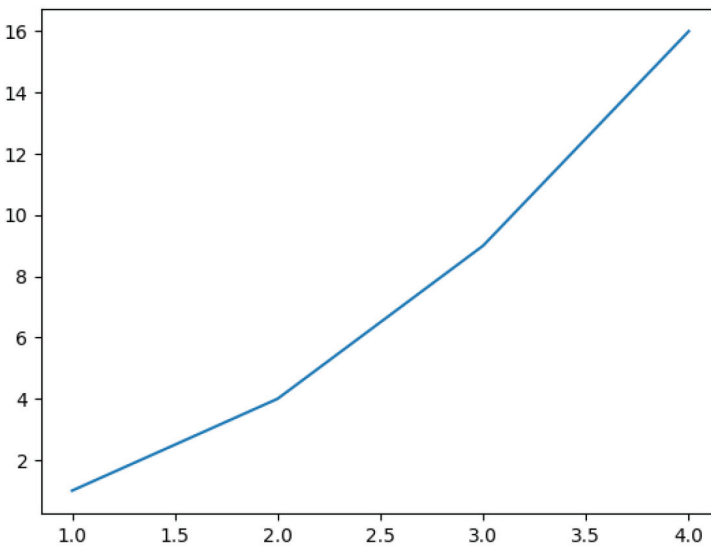
# Create some data
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]

# Create the plot
plt.plot(x, y)

# Show the plot
plt.show()
```

The graphical result is:

Fig. 3 Matplotlib to create a line plot



This will create a figure with a **line plot** of **y** versus **x**. You can customize the appearance and behavior of your plots in many ways, including changing the **line style**, **color**, and **marker type**; **adding axis**

labels, **titles**, and **legends**; and much more. Here is another example of using these options to create and customize a **line plot**:

```
import matplotlib.pyplot as plt

# Prepare data
x = [2017, 2018, 2019, 2020, 2021]
y = [43, 45, 47, 48, 50]

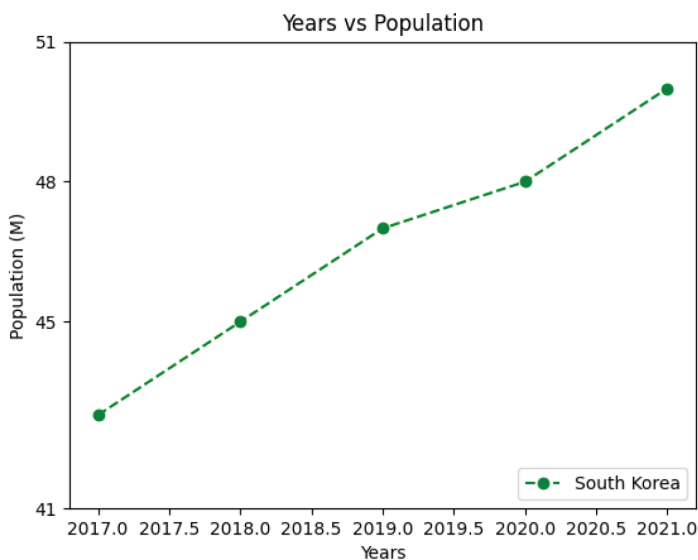
# Plot the data
plt.plot(x, y, marker='o', linestyle='--',
color='g', label='South Korea')

# Customize the plot
plt.xlabel('Years')
plt.ylabel('Population (M)')
plt.title('Years vs Population')
plt.legend(loc='lower right')
plt.yticks([41, 45, 48, 51])

# Show the plot
plt.show()
```

The graphical result is:

Fig. 4 Matplotlib to create and customize a **line plot**:



This will create a **line plot** with circles as markers and a dashed line style, and with the color green. It will also add a legend and customize the **y-axis tick marks**. You can customize the plot in many other ways, such as changing the line width and marker size, adding a grid, and adjusting the axis limits and font properties. You can also create multiple **plots** on the same figure and customize their appearance individually.

[hint!] Matplotlib is a powerful data visualization library in Python that is widely used by data scientists for creating a variety of different types of plots and charts. Some of the key reasons why data visualization with Matplotlib is worth studying for a data scientist include:

- **Customization:** Matplotlib allows you to customize virtually every aspect of your plots, from the colors and styles of the lines and markers to the axis labels and titles. This allows you to create plots that are tailored to your specific needs and preferences.
- **Ease of use:** Matplotlib has a simple and intuitive interface, making it easy to create basic plots quickly. At the same time, it also offers advanced features for more complex plots.
- **Wide range of plot types:** **Matplotlib** is a useful tool for creating various types of plots such as **line plots**, **scatter plots**, **bar plots**, **histograms**, and more. Its versatility allows it to be used for visualizing various kinds of data.
- **Compatibility with other libraries:** Matplotlib is often used in conjunction with other libraries, such as Pandas and Seaborn, which can make it even more powerful and convenient to use.
- **Matplotlib** is an essential tool for data visualization in Python, and it is worth studying for any data scientist who wants to effectively communicate their findings and explore their data.

★ *Matplotlib exercises for DataVisualization*

Here are a few exercises that you can try to practice data visualization with **Matplotlib**:

- a) Create a simple **line plot**. Start by generating some random data using the **np.random** module, and then use the **plot()** function to

- create a line plot of the data. Experiment with different parameters, such as the color and style of the line, and the labels for the axes.
- b) Create a **scatter plot**. Use the **scatter()** function to create a scatter plot of two sets of data. Experiment with different parameters, such as the size and color of the markers, and the labels for the axes.
 - c) Create a **bar plot**. Use the **bar()** function to create a bar plot of some categorical data. Experiment with different parameters, such as the color and width of the bars, and the labels for the axes.
 - d) Create a **histogram**. Use the **hist()** function to create a histogram of some continuous data. Experiment with different parameters, such as the number of bins, the range of the data, and the labels for the axes.
 - e) Create a **subplot**. Use the **subplot()** function to create a figure with multiple subplots. You can create different types of plots in each subplot, or plot the same data in different ways.

Seaborn

Seaborn is a popular library for **data visualization** in **data science** because it provides a high-level interface for creating attractive and informative statistical graphics. It is built on top of **Matplotlib**, which is a powerful library for creating **static**, **animated**, and **interactive 2D plots** in Python. **Seaborn** makes it easy to create a wide range of plots, including **line plots**, **scatter plots**, **bar plots**, **box plots**, and more, and customize their appearance and behavior consistently. Some of the key features of **Seaborn** that make it particularly useful for **data science visualization** include:

- **Consistent interface:** **Seaborn** provides a consistent interface for creating different types of plots, which makes it easier to use than **Matplotlib**. For example, the same function can be used to create a **bar plot**, a **box plot**, or a **histogram**, and the same options are available for customizing the appearance of each plot.
- **Attractive default styles:** **Seaborn** provides attractive default styles for its plots, which can be customized if desired. This makes it easier to create professional-looking plots without having to spend a lot of time on formatting and customization.

- **Statistical plots:** **Seaborn** provides several types of plots that are specifically designed for **visualizing statistical** relationships in datasets, such as the **regression plot**, the **pair plot**, and the **violin plot**. These plots are particularly useful for exploring and visualizing statistical relationships in datasets.
- **Built-in support for plotting with Pandas:** **Seaborn** has built-in support for **plotting with Pandas DataFrames**, which is a common data structure in **data science**. This makes it easy to create plots directly from **Pandas DataFrames** without having to manipulate the data first.
- **Seaborn** is a valuable tool for **data visualization** in **data science** because it provides an easy-to-use interface for creating a wide range of plots and customizing their appearance, and it is particularly well-suited for exploring and visualizing statistical relationships in datasets. Here is an example of how you can use **Seaborn** to create a **line plot**:

```
import seaborn as sns
import matplotlib.pyplot as plt

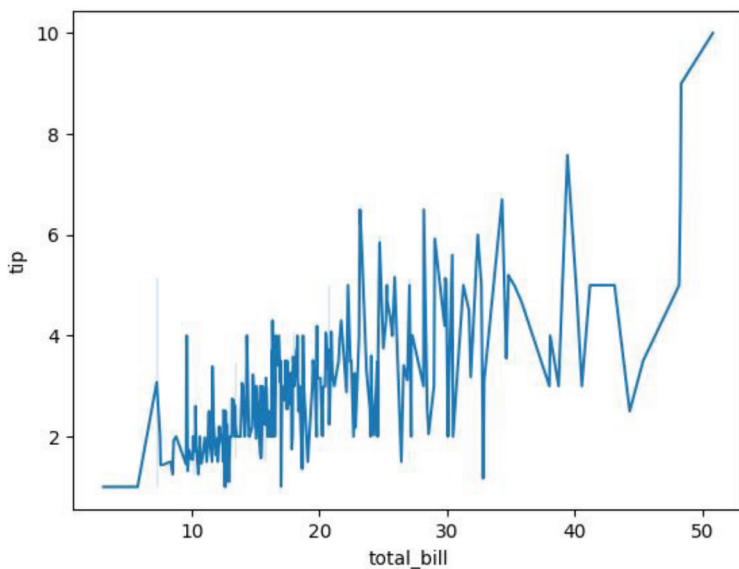
# Load the tips dataset
tips = sns.load_dataset("tips")

# Create a line plot of the total bill versus the tip
sns.lineplot(x="total_bill", y="tip", data=tips)

# Show the plot
plt.show()
```

The graphical result is:

Fig. 5 Seaborn to create a **line plot**:



Here is an example of how you can use **Seaborn's** `barplot()` function to create a **bar plot**:

```
import seaborn as sns
import matplotlib.pyplot as plt

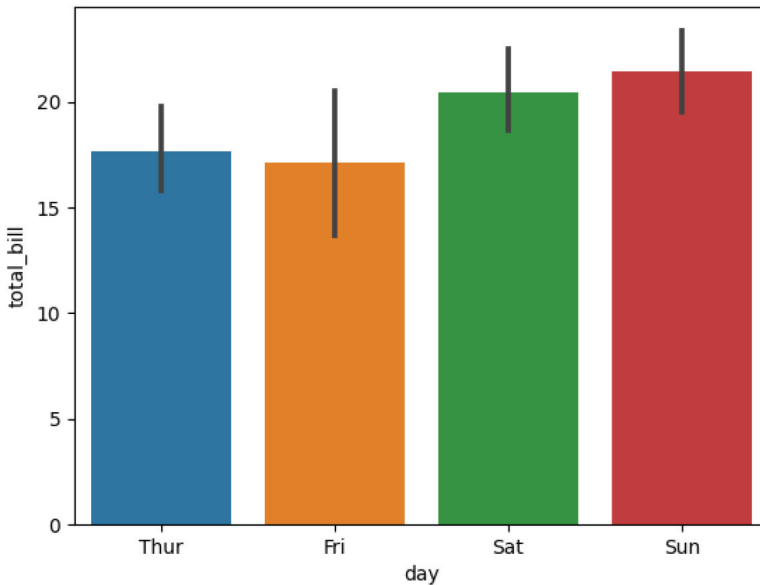
# Load the tips dataset
tips = sns.load_dataset("tips")

# Create a bar plot of the average total bill
# amount versus the day of the week
sns.barplot(x="day", y="total_bill", data=tips)

# Show the plot
plt.show()
```

The graphical result is:

Fig. 6 Seaborn's `barplot()` function to create a **bar plot**



As you can see, a **bar chart** has been generated, which will show the average total invoice value for each day of the week in the tipping dataset. The days of the week will be on the **x-axis** and the average total invoice amount will be on the **y-axis**. You can customize the appearance and behavior of the plot in many ways, such as changing the **bar color**, **width**, and **edge color**; adding **error bars** to show the uncertainty in the mean; and much more. For example, you can customize the **bar colors** and add **error bars** like this:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the tips dataset
tips = sns.load_dataset("tips")

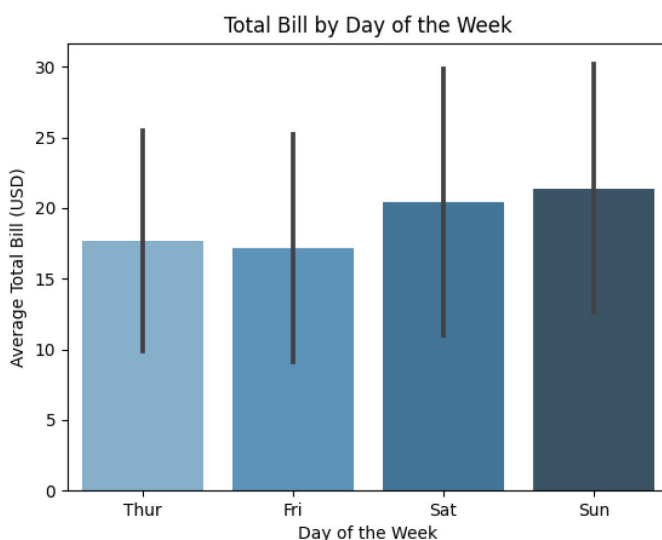
# Create a bar plot of the average total bill
amount versus the day of the week
sns.barplot(x="day", y="total_bill", data=tips,
            ci="sd", palette="Blues_d")
```

```
# Customize the plot
plt.xlabel("Day of the Week")
plt.ylabel("Average Total Bill (USD)")
plt.title("Total Bill by Day of the Week")

# Show the plot
plt.show()
```

The graphical result is:

Fig. 7 Customize the **bar colors** and add **error bars**



Seaborn provides several functions for setting the style and appearance of plots. Here is a brief explanation of each function:

- **sns.set_style()**: This function sets the default **aesthetic style** for all plots in a session. The style defines the overall look and feel of the plots, including the **background color**, the **gridlines**, the **font sizes**, and much more. Some of the available styles are:
- **darkgrid**: This style uses a dark grid background with light ticks and labels.
- **whitegrid**: This style uses a white grid background with dark ticks and labels.
- **dark**: This style uses a dark background with light ticks and labels.

- **white:** This style uses a white background with dark ticks and labels.
- **ticks:** This style uses a white background with ticks and no gridlines.
- **sns.set_palette():** This function sets the default **color palette** for all plots in a session. The palette defines the colors that will be used to distinguish different data series or categories in a plot. Some of the available palettes are:
- **husl:** This palette is based on the **HUSL** color space and provides a wide range of colors that are well-suited for use in data visualization.
- **muted:** This palette consists of muted colors that are suitable for use in a variety of contexts.
- **colorblind:** This palette is designed to be easily distinguishable by people with colorblindness.
- **sns.color_palette():** This function returns a list of colors from the current color palette. You can use this function to get a list of colors to use in a plot, or to explore the colors in the current palette (Waskom, M., Botvinnik, O., Ostblom, J., Lukauskas, S., Hobson, P., Gempertline, D. C., ... & Evans, C., 2020).

You can use these functions in combination to create **plots** with the desired style and appearance. For example:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the style and color palette
sns.set_style("darkgrid")
sns.set_palette("husl", 3)

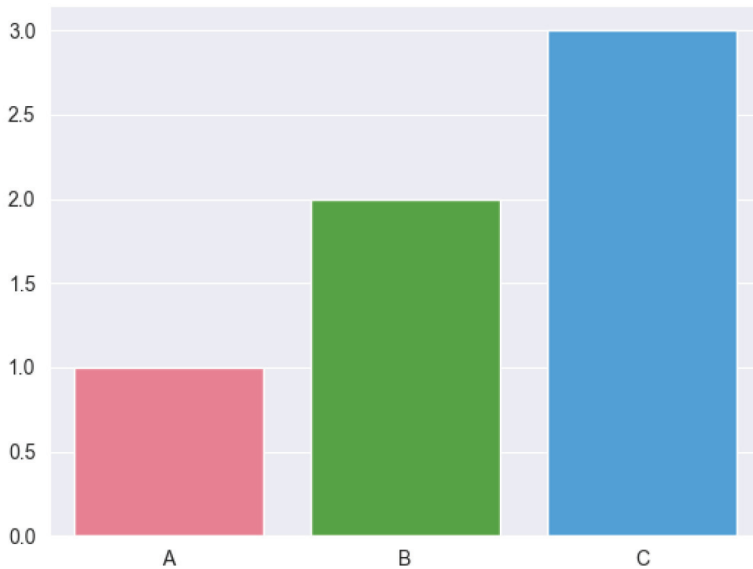
# Get the colors from the current palette
colors = sns.color_palette()

# Create a bar plot with the first three colors in
the palette
sns.barplot(x=["A", "B", "C"], y=[1, 2, 3],
palette=colors[:3])

# Show the plot
plt.show()
```

The graphical result is:

Fig. 8 Plots with the desired style and appearance



This will create a **bar plot** with a **darkgrid style** and a **husl color palette** and will use the first three colors in the palette for the bars. You can use **Matplotlib's rc()** function to set the font sizes of various elements in a plot. Here is a brief explanation of each parameter:

- **titlesize:** This parameter sets the font size of the **plot title**.
- **labelsize:** This parameter sets the font size of the **x-axis** and **y-axis labels**.
- **xtick.labelsize:** This parameter sets the font size of the **x-axis tick labels**.
- **ytick.labelsize:** This parameter sets the font size of the **y-axis tick labels**.
- **legend.fontsize:** This parameter sets the font size of the **legend**.
- **font.size:** This parameter sets the **default font size** for all text in the plot.

To use these parameters, you will need to **call the rc()** function before creating the plot. For example:

```
import matplotlib.pyplot as plt

# Set the font sizes
plt.rc('axes', titlesize=18)
plt.rc('axes', labelszize=14)
plt.rc('xtick', labelszize=13)
plt.rc('ytick', labelszize=13)
plt.rc('legend', fontsize=13)
plt.rc('font', size=13)

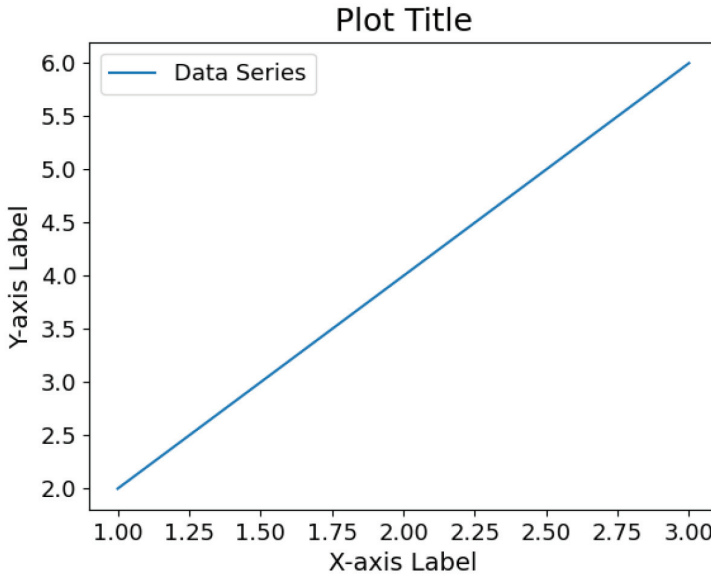
# Define the data for the plot
x = [1, 2, 3]
y = [2, 4, 6]

# Create the plot
plt.plot(x, y)
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.title("Plot Title")
plt.legend(["Data Series"])

# Show the plot
plt.show()
```


The graphical result is:

Fig. 9 Matplotlib with parameters



This will create a **plot** with the specified **font sizes** for the **title**, **labels**, **tick labels**, and **legend**.

★ *Some exercises with Seaborn*

Here are some random graphical exercises using **Seaborn** that you can try:

- Create a **line plot** of the total bill amount versus the tip amount for each tip in the tip's dataset. Add a legend to the plot and customize the **line style**, **color**, and **marker type**.
- Create a **bar plot** of the average total bill amount versus the day of the week for each tip in the tip's dataset. Add **error bars** to the plot to show the uncertainty in the meaning.
- Create a **histogram** of the total bill amount for each tip in the tip's dataset. Customize the **bin size** and **edge color**.
- Create a **box plot** of the total bill amount for each tip in the tip's dataset. Customize the **whisker length** and **flier style**.

- g) Create a **scatter plot** of the total bill amount versus the tip amount for each tip in the tip's dataset. Add a **trendline** to the plot and customize the **marker size** and **color**.
- f) Create a **pairplot** of the tip's dataset, showing **scatter plots** of all pairs of numeric variables and histograms of the individual variables. Customize the **marker size** and **color**, and the **bin size** and **edge color**.

Web Scraping in Data Science

Web scraping is a technique used to extract data from websites. It involves making **HTTP** requests to a website's server, **downloading the HTML** of the web page, and **parsing** that HTML to **extract the data** you're interested in. **Web scraping** is often used in **Data Science** to **gather data** that is not available through **APIs** or other means. There are several libraries and frameworks available in Python that can be used for **web scraping**, including **Beautiful Soup**, **Selenium**, and **Scrapy**. Before using **web scraping** in a **Data Science** project, it is important to consider the ethical and legal implications of the technique. Some websites explicitly prohibit the use of **web scraping** in their terms of service, and it is important to respect those terms. Additionally, **web scraping** can place a heavy load on a website's servers, so it is important to be mindful of this and to use **web scraping responsibly** (Parvez, M. S., Tasneem, K. S. A., Rajendra, S. S., & Bodke, K. R. (2018)). It is also important to consider the structure and stability of the website you are **scraping**. If the structure of the website changes frequently, your **web scraper** may break. To avoid this, you may want to consider using a more stable source of data or building some resilience into your **web scraper** to handle changes to the website.

HTML (HyperText Markup Language) is a markup language used to structure and format the content of a web page. It is the standard markup language for creating web pages and is used to specify the structure and layout of a document written in **HTML**. In the context of **web scraping**, **HTML** is used to define the structure of a web page and the content that it contains. **Web scrapers extract data** from a web page by **parsing** the **HTML** of the page and **extracting specific elements from it**. **HTML** consists of a series of elements that are used to define the structure and content of a web page (Parvez, M. S., Tas-

neem, K. S. A., Rajendra, S. S., & Bodke, K. R. (2018). These elements are represented by **tags**, which are enclosed in **angle brackets**. Some common **HTML** elements include:

<p>: Used to define a **paragraph of text**.

<h1>: Used to define a **heading**. There are six levels of headings, ranging from **<h1>** (the largest) to **<h6>** (the smallest).

<a>: Used to define a **hyperlink**.

****: Used to include an **image** on a web page.

<div>: Used to **group other elements** together and apply styles to them.

HTML also allows you to specify attributes for elements, which provide additional information about the element. For example, the **href** attribute of an **<a>** element specifies the **URL** that the **hyperlink** should link to. When **web scraping**, it is important to understand the structure of the **HTML** of the web page you are working with. This will allow you to identify the elements and attributes that contain the data you are interested in and write a **web scraper** that can **extract that data**. Here is an example of a simple **HTML** code for a web page:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Web Page</title>
</head>
<body>
  <h1>Welcome to my web page!</h1>
  <p>This is my personal web page, where I share
my thoughts and ideas.</p>
  <p>I'm also an avid fan of cats, so you'll find
plenty of cat-related content here too.</p>
  <h2>Recent Posts</h2>
  <ul>
    <li><a href="/posts/funny-cat-videos">Funny
Cat Videos</a></li>
    <li><a href="/posts/cat-care-tips">Cat Care
Tips</a></li>
  </ul>
</body>
</html>
```

This **HTML** code defines a web page with a title, a heading, and two paragraphs of text. It also includes a list of recent posts, which are **hyperlinked** to other pages. To extract data from this **HTML** using a **web scraper**, you could use code like this:

```
from bs4 import BeautifulSoup

html = '''
<!DOCTYPE html>
<html>
<head>
  <title>My Web Page</title>
</head>
<body>
  <h1>Welcome to my web page!</h1>
  <p>This is my personal web page, where I share
my thoughts and ideas.</p>
  <p>I'm also an avid fan of cats, so you'll find
plenty of cat-related content here too.</p>
  <h2>Recent Posts</h2>
  <ul>
    <li><a href="/posts/funny-cat-videos">Funny
Cat Videos</a></li>
    <li><a href="/posts/cat-care-tips">Cat Care
Tips</a></li>
  </ul>
</body>
</html>
'''

soup = BeautifulSoup(html, 'html.parser')

# Extract the title of the web page
title = soup.title.string
print(title)

# Extract the list of recent posts
recent_posts = []
for li in soup.find_all('li'):
    recent_posts.append(li.a.string)
print(recent_posts)
```

This code uses the **Beautiful Soup** library to **parse** the **HTML** and extract the **title** of the web page and the list of recent posts. Here is another example of using the **Beautiful Soup** library to **extract data** from a web page:

```
import requests
from bs4 import BeautifulSoup

# Make an HTTP GET request to the web page
response =
requests.get('https://www.example.com/products')

# Parse the HTML of the web page
soup = BeautifulSoup(response.text, 'html.parser')

# Find all the products on the page
products = soup.find_all('div', class_='product')

# For each product, extract the product name and
price
for product in products:
    name = product.find('h3').text
    price = product.find('p', class_='price').text
    print(f'{name}: {price}')
```

In this example, we make an **HTTP GET** request to a web page that displays a list of products and use **Beautiful Soup** to **parse** the **HTML** of the page. We then use the **find_all()** method to find all the **div** elements with the class **product**, which represents each product on the page. For each product, we use the **find()** method to extract the product name (contained in an **h3** element) and the price (contained in a **p** element with the class **price**). We then print the name and price of each product. This is just a simple example, but it illustrates how **Beautiful Soup** can be used to **extract data** from a web page. You can customize the code to extract other data points or to **scrape** multiple pages (Hajba, G. L., 2018).

XPath (XML Path Language) is a language that can be used to navigate through the elements and attributes of an **XML** document, such as an **HTML** web page. It is commonly used in **web scraping** to

locate and extract specific elements from a web page. Both **Selenium** and **Scrapy**, which are popular libraries for **web scraping**, provide support for using **XPath** to locate elements on a web page. In **Selenium**, you can use the `find_element_by_xpath()` method to locate a single element on the page, or the `find_elements_by_xpath()` method to locate multiple elements. For example:

```
from selenium import webdriver

# Start a web browser and navigate to a web page
driver = webdriver.Chrome()
driver.get('https://www.example.com')

# Find the first element on the page with the
class "main-title"
title_element =
driver.find_element_by_xpath('//*[@class="main-
title"]')
print(title_element.text)

# Find all the elements on the page with the
class "product"
product_elements =
driver.find_elements_by_xpath('//*[@class="product
"]')
for element in product_elements:
    print(element.text)

# Close the web browser
driver.quit()
```

In **Scrapy**, you can use the `xpath()` method of a **Selector object** to locate elements on a web page. For example:

```
import scrapy

class ProductSpider(scrapy.Spider):
    name = 'products'
    start_urls = ['https://www.example.com/prod-
ucts']

    def parse(self, response):
```

```
# Extract the product name and price for each
product on the page
for product in
response.xpath('//div[@class="product"]'):
    name = product.xpath('./h3/text()').get()
    price =
product.xpath('./p[@class="price"]/text()').get()
    yield {'name': name, 'price': price}
```

In both examples, the **XPath** expressions are used to locate elements on the page based on their **tag name**, **class**, or other **attributes**. The **XPath** syntax can be quite powerful and allows you to select elements based on complex criteria. It is worth noting that while **XPath** is a useful tool for **web scraping**, it can also be prone to breakage if the structure of the web page changes. As a result, it is a good idea to use other techniques, such as **CSS** selectors, to locate elements on the page whenever possible. **XPath** provides a set of functions and operators that can be used to perform various tasks such as **string manipulation**, **arithmetic calculations**, and **comparison of values**. Here is a list of some of the common **functions** and **operators** available in **XPath**:

- **String functions**: These functions operate on strings and include functions such as **concat()**, **substring()**, **string-length()**, and **starts-with()**.
- **Numeric functions**: These functions operate on numbers and include functions such as **number()**, **sum()**, **floor()**, and **ceiling()**.
- **Boolean functions**: These functions operate on boolean values and include functions such as **true()**, **false()**, and **not()**.
- **Node set functions**: These functions operate on node sets and include functions such as **last()**, **position()**, and **count()**.

Operators: These are used to perform operations on values and include **arithmetic operators** (+, -, *, div, mod), **comparison operators** (=, !=, <, <=, >, >=), and **logical operators** (and, or, not).

For more information on **XPath** functions and operators, you can refer to the **W3C XPath** specification or consult a reference guide.

Selenium is a powerful tool for automating web browsers. It allows you to write scripts in various programming languages to perform tasks such as filling out forms, clicking buttons, and verifying the content of web pages.

[hints!] Web scraping can be a useful skill for data scientists to learn because it allows them to gather large amounts of data from websites and online sources. This data can then be used for a variety of purposes, such as building machine learning models, conducting research, or generating insights. Web scraping can be especially useful when data is not available through APIs or other structured means, or when you want to gather data from multiple sources in a consistent way. It can also be useful for gathering data that is not easily accessible or that requires a significant amount of manual effort to collect. In summary, learning web scraping can be helpful for data scientists because it allows them to gather and analyze data from a wide range of sources, and can save time and effort when working with large or complex datasets.

Scraping websites with Selenium

To use **Selenium to scrape websites**, you will need to **install** the **Selenium** Python library and a web driver. A web driver is a program that allows **Selenium** to interact with a web browser. The most common web drivers are **ChromeDriver** (for Chrome), **GeckoDriver** (for Firefox), and **SafariDriver** (for Safari). Here is an example of how you can use **Selenium** to scrape a web page:

a) Install the **Selenium** Python library and **ChromeDriver**:

```
pip install selenium
```

b) Import the necessary modules:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
```

c) Create an instance of **ChromeDriver**:

```
driver = webdriver.Chrome()
```


d) Navigate to the web page you want to scrape:

```
driver.get("http://www.example.com")
```

e) Use the `find_element` method to locate the elements on the page that you want to scrape:

```
title_element = driver.find_element(By.XPATH,
    "//h1[@class='title']")
title = title_element.text
```

f) Extract the data you need from the elements:

```
title = title_element.text
```

g) Close the web driver when you are done:

```
driver.quit()
```

[hints!] Web scraping can be a resource-intensive task, so it is important to use it responsibly and in compliance with the terms of use of the websites you are scraping. The `scrapy startproject` command creates a new Scrapy project with the specified name. The `scrapy genspider` command generates a new spider within the project. Here's an example of how you can use these commands to create a Scrapy project and spider:

```
# Create a new Scrapy project
scrapy startproject myproject

# Create a new spider within the project
scrapy genspider myspider mywebsite.com
```

a) Open a terminal and navigate to the directory where you want to create your Scrapy project.

b) Run the `scrapy startproject` command, followed by the name of your project:

```
scrapy startproject my_first_spider
```

c) Change the directory to the project directory:

```
cd my_first_spider
```

- d) Run the **scrapy genspider** command, followed by the name of your spider and the domain of the website you want to scrape:

```
scrapy genspider example example.com
```

This will create a **new spider** with the specified name and domain in your **Scrapy project**. You can then edit the **spider file** (located in the **spider's directory**) to specify the details of the scraping process. When you create a **new spider** using the **scrapy genspider** command, you will get a **basic template** that includes the name, allowed domains, and start **URLs** of the spider. The **parse method** is where you will specify the details of the **scraping** process. Here are some additional details on the steps you mentioned:

- a) **Finding elements:** To find elements on a web page using **Scrapy**, you can use the **response object** and the **xpath method**. For example, you can use **response.xpath('//h1')** to find all **h1** elements on the page.
- b) **Getting the text:** To obtain the text of an element, you can use the **text()** function and either the **get()** or **getall()** **method**. The **get()** **method** returns the first matching element, while the **getall()** **method** returns a list of all matching elements. For example, **response.xpath('//h1/text()').get()** will return the text of the first **h1** element, while **response.xpath('//h1/text()').getall()** will return a list of the text of all **h1** elements.
- c) **Returning data extracted:** To return the data you have extracted; you can use the **yield keyword**. This will **yield** each item as a dictionary, which can be later used to output the data to a file or store it in a database. For example, **yield {'titles': title}** will **yield** the title variable as a dictionary with the key titles.
- d) **Running the spider and exporting data:** To **run the spider**, you can use the **scrapy crawl** command followed by the name of the spider. To export the data to a **CSV** or **JSON** file, you can use the **-o** option followed by the name of the file. For example, **scrapy crawl example -o name_of_file.csv** will run the example spider and output the data to a **CSV** file called **name_of_file.csv**.

★ *A few exercises with Webscarping*

Here are a few exercises you can try to practice **web scraping** using Scrapy:

- a) **Scrape a list of products** from an **e-commerce website**: You can try **scraping** a list of products, such as books or electronics, from an e-commerce website. You can **extract information** such as the product name, price, and rating. You can then output the data to a **CSV** or **JSON** file.
- b) **Scrape a news website**: You can try **scraping a news website** to extract the latest articles and their titles. You can then output the data to a **CSV** or **JSON** file and analyze the data to see which topics are most popular or to track the coverage of a particular event.
- c) **Scrape social media websites**: You can try **scraping social media websites**, such as Twitter or Instagram, to extract data such as user profiles, posts, and comments. You can then analyze the data to understand trends or to perform sentiment analysis.
- d) **Scrape job listings**: You can try **scraping job listing websites** to extract information about available job openings. You can extract data such as the job title, location, and requirements, and output the data to a **CSV** or **JSON** file.

Remember to always **respect the terms of use of the websites** you are **scraping**, and **to use web scraping responsibly**.

★ *Example of Gaussian noise (standard deviation)*

```
import PySimpleGUI as sg
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Update function
def data_gen():
    fig1 = plt.figure(dpi=125)
    x = np.round(10*np.random.random(20),3)
    y = 0.5*x**2+3*x+5+np.random.normal
(scale=noise, size=20)
```

```

    p1 = plt.scatter(x,y,edgecolor='k')
    plt.ylabel('Y-Values')
    plt.xlabel('X-Values')
    plt.title('Scatter plot')
    figure_x, figure_y, figure_w, figure_h =
fig1.bbox.bounds
    return (x,y,fig1,figure_x, figure_y,
figure_w, figure_h)

def fit_redraw(x,y,model):
    fig2 = plt.figure(dpi=125)
    x_min,x_max = np.min(x),np.max(x)
    x_reg = np.arange(x_min,x_max,0.01)
    y_reg = np.poly1d(model)(x_reg)
    p1 = plt.scatter(x,y,edgecolor='k')
    p2 =
plt.plot(x_reg,y_reg,color='orange',lw=3)
    plt.ylabel('Y-Values')
    plt.xlabel('X-Values')
    plt.title('Scatter plot with fit')
    return fig2

def draw_figure(canvas, figure, loc=(0, 0)):
    figure_canvas_agg =
FigureCanvasTkAgg(figure, canvas)
    figure_canvas_agg.draw()

figure_canvas_agg.get_tk_widget().pack(side='top',
fill='both', expand=1)
    return figure_canvas_agg

def delete_fig_agg(fig_agg):
    fig_agg.get_tk_widget().forget()
    plt.close('all')

# Define the window's contents i.e. layout
layout = [
    [sg.Button('Generate',enable_events=True,
key='-GENERATE-', font='Helvetica 16'),
    sg.Button('Fit',enable_events=True,
key='-FIT-', font='Helvetica 16', size=(10,1))],

```

```

        [sg.Text("Gaussian noise (std.
deviation)", font=('Helvetica', 12)),
         sg.Slider(range=(0,6), default_value=3,
size=(20,20), orientation='h',font=('Helvetica',
12), key='--NOISE--')],
        [sg.Canvas(size=(350,350), key='--CANVAS-
', pad=(20,20))],
        [sg.Button('Exit')]],
        ]

# Create the window
window = sg.Window('Polynomial fitting', layout,
size=(700,700))

# Event loop
fig_agg = None
while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    if event == '--GENERATE--':
        noise = values['--NOISE--']
        if fig_agg is not None:
            delete_fig_agg(fig_agg)
        x,y,fig1,figure_x, figure_y, figure_w,
figure_h = data_gen()
        canvas_elem = window['--CANVAS--'].TKCanvas

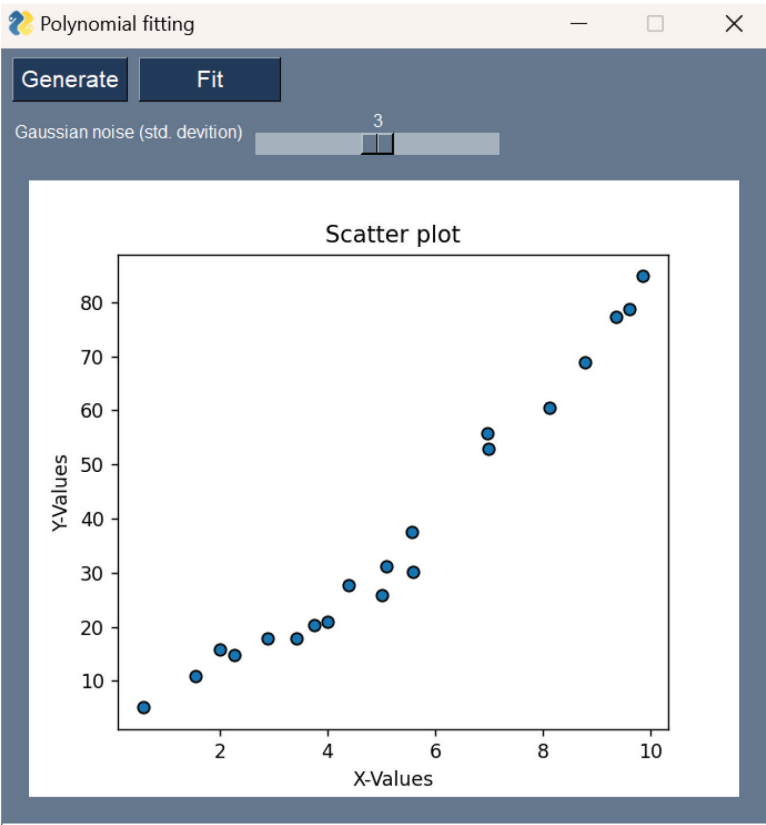
canvas_elem.Size=(int(figure_w),int(figure_h))
        fig_agg = draw_figure(canvas_elem, fig1)
    if event == '--FIT--':
        model = np.polyfit(x, y, 2)
        if fig_agg is not None:
            delete_fig_agg(fig_agg)
        fig2 = fit_redraw(x,y,model)
        canvas_elem = window['--CANVAS--'].TKCanvas
        fig_agg = draw_figure(canvas_elem, fig2)

# Close the window i.e. release resource
window.close()

```

The graphical result is:

Fig. 10 Example of Gaussian noise (standard deviation)



PART III

Introduction to Business Statistics in Data Science

Statistics is a branch of mathematics that deals with the collection, analysis, interpretation, presentation, and organization of data. In the field of data science, statistics play a crucial role in understanding and analyzing data, as well as in making informed decisions based on that data. In business, statistics are often used to analyze market trends, predict future performance, and make strategic decisions. For example, a business might use statistical analysis to identify patterns in customer behavior or to forecast future sales. Many different statistical techniques can be used in data science and business, including:

- **Descriptive statistics:** These techniques are used to summarize and describe data, such as calculating the mean, median, and standard deviation of a set of numbers.
- **Inferential statistics:** These techniques are used to conclude a larger population based on a sample of data.
- **Correlation and regression:** These techniques are used to examine the relationship between two or more variables and to make predictions based on that relationship.
- **Hypothesis testing:** These techniques are used to test whether a certain relationship or pattern in the data is statistically significant.
- **Time series analysis:** These techniques are used to analyze data over time and to make predictions about future trends.

Statistics is a vital tool for data science and business, as it allows for the analysis and interpretation of data in a way that can inform decision-making and drive progress.

[hints!] There are several techniques that data scientists use when working with traditional data, which refers to data that is structured and organized predictably. Some common techniques include:

- **Data cleaning and preprocessing:** Before analyzing data, it is often necessary to clean and preprocess it to ensure that it is accurate, consistent, and ready for analysis. This may involve tasks such as correcting errors, handling missing values and standardizing data formats.
- **Descriptive statistics:** Data scientists use descriptive statistics to summarize and describe data, such as by calculating measures of central tendency (e.g., mean, median) and dispersion (e.g., standard deviation).
- **Inferential statistics:** Data scientists use inferential statistics to make inferences about a larger population based on a sample of data. This may involve techniques such as hypothesis testing and confidence intervals.
- **Correlation and regression:** Data scientists use correlation and regression to examine the relationship between two or more variables and to make predictions based on that relationship.
- **Time series analysis:** Data scientists use time series analysis to analyze data over time and to make predictions about future trends.
- **Data visualization:** Data scientists use data visualization techniques to create charts, graphs, and other visual representations of data to aid in understanding and communication.

These techniques are used to extract insights and knowledge from traditional data, and to support decision-making and progress in a wide range of contexts.

Big Data, Statistics, and Probability

Big data refers to very large datasets that are too large or complex to be processed using traditional data processing tools. Here are some real-life examples of big data:

- **Financial data:** Financial institutions generate vast amounts of data on transactions, investments, and financial markets. This data

can be analyzed to understand market trends, identify risk, and optimize financial strategies.

- **Healthcare data:** Healthcare providers collect large amounts of data on patient health, treatments, and outcomes. This data can be analyzed to improve patient care, identify trends, and optimize resource allocation.
- **Environmental data:** Governments and organizations collect data on environmental factors such as air and water quality, weather patterns, and natural disasters. This data can be analyzed to understand and predict environmental trends, and to develop strategies for adapting to and mitigating the impacts of climate change.
- **Educational data:** educational institutions collect data on student performance, enrollment, and demographics. This data can be analyzed to understand trends, identify areas for improvement, and optimize resource allocation.
- **Customer data:** Businesses collect data on customer interactions, preferences, and feedback. This data can be analyzed to understand customer behavior, identify trends, and optimize marketing and customer service strategies.

Big Data has the potential to provide valuable insights and drive progress in a wide range of industries and applications.

Business Intelligence (BI) techniques

Business intelligence (BI) refers to the process of collecting, storing, and analyzing data to inform business decision-making and strategy. Some common techniques used in **BI** include:

- **Data warehousing:** Data warehousing involves storing and organizing data in a central location, making it easier to access and analyze.
- **Online analytical processing (OLAP):** **OLAP** involves using specialized software to analyze data from multiple sources, allowing users to view data from different perspectives and drill down into specific details.
- **Data mining:** Data mining involves using algorithms to search for patterns and trends in large datasets, allowing businesses to identify new opportunities and make informed decisions.

- **Dashboarding:** Dashboarding involves creating visual representations of data, such as charts and graphs, to help businesses quickly understand key metrics and trends.
- **Reporting:** Reporting involves generating regular reports on business performance and key metrics, helping businesses track progress and identify areas for improvement.

BI techniques are used to help businesses make sense of data and to use that data to inform decision-making and strategy. A retailer might use BI tools to analyze sales data to identify trends and patterns, such as which products are selling well, which regions are most profitable, and how sales are affected by different marketing campaigns. This information can help the retailer optimize its inventory, pricing, and marketing strategies. A manufacturer might use BI to track and analyze data about production, inventory, and shipping to optimize its supply chain and reduce costs. For example, the manufacturer might use BI to identify bottlenecks in the production process or to forecast demand for its products. A company might use BI to analyze data about its customers, such as their purchasing history, demographics, and feedback, to better understand their needs and preferences. This information can be used to tailor marketing campaigns, improve customer service, and retain customers. A financial institution might use BI to analyze data about its operations, such as expenses, revenue, and assets, to identify opportunities for cost savings and to optimize its financial performance. A healthcare organization might use BI to analyze data about patient care, such as treatment outcomes, to identify trends and patterns and to improve patient outcomes.

Big Data and Statistics

Big data can be structured or unstructured, and they can be generated from a wide range of sources, including social media, web logs, sensors, and transactional systems (Thakuriah, P. V., Tilahun, N. Y., & Zellner, M. (2017). Statistics is the science of collecting, analyzing and interpreting data. It involves the use of statistical tools and techniques to describe and understand patterns and trends in data, and to make informed decisions based on those insights. In Python, some several

libraries and tools can be used for working with big data and statistics. Some of the most popular ones include:

NumPy: A library for working with large, multi-dimensional arrays and matrices of numerical data.

Pandas: A library for working with tabular data, including functions for reading and writing data from various formats (e.g. CSV, Excel), manipulating data (e.g. grouping, merging), and performing statistical analysis (e.g. computing means, variances).

Matplotlib: A library for creating visualizations of data, including line plots, scatter plots, bar plots, and more.

SciPy: A library for scientific computing, including functions for statistical analysis, optimization, and signal processing.

statsmodels: A library for statistical modeling and testing, including functions for linear regression, ANOVA, and more.

Here's an example of how you might use these libraries in Python to work with big data and perform some basic statistical analysis:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import statsmodels.api as sm

# Read in the data from a CSV file using Pandas
data = pd.read_csv('data.csv')

# Extract the columns of interest and convert
them to NumPy arrays
x = data['column1'].values
y = data['column2'].values

# Perform a linear regression
slope, intercept, r_value, p_value, std_err =
stats.linregress(x, y)

# Use the regression results to make predictions
y_pred = intercept + slope * x
```

```
# Plot the data and regression line
plt.scatter(x, y)
plt.plot(x, y_pred, color='r')
plt.show()

# Use statsmodels to perform an ANOVA
model = sm.OLS(y, x)
results = model.fit()
print(results.summary())
```

This code reads data from a CSV file using Pandas, performs a linear regression using the NumPy and SciPy libraries, and plots the data and regression line using Matplotlib. It also uses statsmodels to perform an ANOVA (analysis of variance) to test for statistical significance.

[hints!] Learning big data and statistics can be a challenging but rewarding process, especially for those interested in becoming data scientists and communication specialists. Always start with the basics and make sure you have a solid foundation in math and statistics before diving into big data. This will help you understand and interpret the data you're working with. Learn a programming language to work with big data, you'll need to be proficient in a programming language like Python or R. Start learning one of these languages and practice using it to analyze and manipulate data. Get hands-on experience and remember that the best way to learn is by doing. Look for opportunities to work with real-world data sets and practice applying the concepts you're learning. This could be through internships, projects, or online data science competitions. Seek out resources and guidance. There are many online resources and courses available to help you learn big data and statistics. Consider enrolling in a structured course or seeking out a mentor or tutor to guide you through the learning process. Practice communication skills, because as a data scientist and communication specialist, it's important to be able to communicate your findings clearly and effectively to a variety of audiences. Practice presenting your data and findings clearly and concisely, using visualizations and other tools to help convey your message.

Hypothesis testing

Hypothesis testing is a **statistical technique** used to determine whether a hypothesis about a population parameter is true or not based on sample data. In **hypothesis testing**, you start with a **null hypothesis**, which is a statement that assumes there is no relationship between the variables you are studying, and an alternative hypothesis, which is a statement that asserts there is a relationship between the variables. To test a **hypothesis**, you gather sample data and use statistical tests to determine whether the null hypothesis can be rejected in favor of the alternative hypothesis. In Python, you can use the **scipy** library to carry out **hypothesis tests**. Here's an example of how to conduct a **chi-squared test**, which is a statistical test used to determine whether there is a significant difference between the observed and expected frequencies in a categorical data set:

```
import scipy.stats as stats

# Observed frequencies
observed_freqs = [10, 20, 30, 40, 50]

# Expected frequencies
expected_freqs = [15, 15, 15, 15, 15]

# Perform chi-squared test
chi2, p = stats.chisquare(observed_freqs,
                          f_exp=expected_freqs)

# Print p-value
print(p)
```

```
from scipy.stats import ttest_ind

group1 = [1, 2, 3, 4, 5]
group2 = [5, 6, 7, 8, 9]

t_statistic, p_value = ttest_ind(group1, group2)
print(t_statistic) # Output: -4.4619083026757465
print(p_value) # Output: 0.00028205860702362793
```

If the **p-value** is less than a predetermined significance level (usually **0.05**), you can reject the null hypothesis and conclude that there is a significant difference between the means of the two groups. In data science, **hypothesis testing** is used to make decisions about data and to draw conclusions based on the evidence provided by the data. It is an important tool for understanding relationships between variables and for making informed decisions based on data.

Basic probability with Python

Probability is the study of random events and the likelihood of their occurrence. In Python, you can use the random module to perform probability tasks. Here's an example of how you can use the random module to simulate the rolling of a dice:

```
import random

def roll_dice():
    return random.randint(1, 6)

# Roll the dice 10 times
for i in range(10):
    result = roll_dice()
    print(f"Roll {i+1}: {result}")
```

This code defines a function `roll_dice()` that uses the `randint()` function from the random module to generate a random integer between 1 and 6, representing the result of rolling a dice. The for loop then calls this function 10 times, printing the result of each roll.

You can also use the random module to generate random samples from a population, using the `sample()` function. For example:

```
import random

# Generate a random sample of 10 elements from the
# population [1, 2, 3, 4, 5, 6]
sample = random.sample([1, 2, 3, 4, 5, 6], 10)
print(sample)
```

This code generates a random sample of **10** elements from the population `[1, 2, 3, 4, 5, 6]`, which represents the possible outcomes of rolling a dice.

[hints!] Probability is a fundamental concept in statistics and data science that is used to quantify the likelihood of events or outcomes. It is a key tool for understanding and modeling random phenomena, such as the outcomes of experiments or the behavior of systems. In communication, the probability is often used to make predictions about the likelihood of certain events occurring or to assess the reliability of statistical conclusions. For example, a marketing research company may use probability to estimate the likelihood that a new advertising campaign will be successful, or a political pollster may use probability to predict the outcome of an election. Probability is also used in data science to build statistical models that can be used to make predictions about future events or outcomes. For example, a data scientist might use probability to build a model that predicts the likelihood of a customer making a purchase based on their past behavior or to identify patterns in data that may be useful for making decisions or solving problems. Overall, the probability is an important tool for understanding and making sense of data and for making informed decisions based on statistical evidence.

[hints!] Python's `scipy.stats` library allows you to perform statistical probability calculations. Here are a few examples of how you can use `scipy.stats` to analyze probability in statistics:

Determining the likelihood of various outcomes: You can use `scipy.stats`' `binom` class to compute the probability of obtaining a specific number of heads when flipping a coin a certain number of times.

```
import scipy.stats as st

#Calculate the probability of rolling a 6 on a 6-
sided die 10 times
binom = st.binom(10, 1/6)
prob = binom.pmf(2)
print(prob)

#Alternatively, you can use the hypergeom class to
calculate the probability of rolling a 6 on a
6-sided die 10 times
```

```
hypergeom = st.hypergeom(10, 6, 2)
prob = hypergeom.pmf(2)
print(prob)
```

The **scipy.stats** module in Python offers the ability to obtain random samples from a probability distribution. For instance, you can use the **norm** class to select random values from a **normal distribution**:

```
import scipy.stats as st

# Sample 10 random values from a normal distribu-
# tion with mean=0 and std=1
norm = st.norm(0, 1)
samples = norm.rvs(10)
print(samples)
```

Testing statistical hypotheses: You can use the **scipy.stats** module to test statistical hypotheses using probability. For example, you can use the **ttest_ind** function to test whether the means of two samples are equal:

```
import scipy.stats as st

# Test whether the means of two samples are equal
sample1 = [1, 2, 3, 4]
sample2 = [2, 3, 4, 5]
t, p = st.ttest_ind(sample1, sample2)
print(f"t-statistic: {t:.2f}")
print(f"p-value: {p:.2f}")
```

Probability in Data Science

Here are some examples of how you can use probability in data science using Python:

Modeling data with probability distributions: You can use probability distributions to model data and make predictions about future events. For example, you can use the **norm** class from the **scipy.stats** module to fit a normal distribution to a set of data and use this distribution to make predictions about future values:


```
import scipy.stats as st

# Fit a normal distribution to a set of data
data = [1, 2, 3, 4, 5]
mu, std = st.norm.fit(data)
norm = st.norm(mu, std)

# Use the fitted distribution to make predictions
about future values
prediction = norm.rvs(1)
print(prediction)
```

Calculating **cumulative hazard functions (CHF)**: The `scipy.stats` module in Python allows you to find the cumulative hazard function (CHF) of a dataset. For example, you can use the `logistic` class to calculate the **CHF** of a set of data using the logistic distribution:

```
import scipy.stats as st
```

Calculate the CHF of a set of data using the logistic distribution:

```
data = [1, 2, 3, 4, 5]
chf = st.logistic(data)
```

Evaluate the CHF at different points:

```
x = np.linspace(-5, 5, 100)
y = chf.chf(x)
```

Fundamentals of Combinatorics

Combinatorics is the branch of mathematics that deals with counting the number of ways that objects can be arranged or combined. It is a useful tool for data science because it can be used to analyze and understand complex systems, such as networks or data sets. Here are some basic concepts in combinatorics that are useful for data science:

Combinations: A combination is a selection of objects from a larger set, where order does not matter. For example, the combinations of the set {1, 2, 3} are (1, 2), (1, 3), (2, 3), (1, 2, 3), and (2, 3, 1). In Python, you can use the itertools module to generate combinations:

```
import itertools

Generate all combinations of the set [1, 2, 3, 4]
combinations = itertools.combinations([1, 2, 3,
4], 2)
print(list(combinations))
```

Source: (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620)

Generate all combinations of the set [1, 2, 3, 4] with replacement:

```
combinations_with_replacement = itertools.combina-
tions_with_replacement([1, 2, 3, 4], 2)
print(list(combinations_with_replacement))
```

Combinations: A combination is a selection of objects without regard to order. For example, the combinations of the set {1, 2, 3} are (1, 2), (1, 3), (2, 3), and (1, 2, 3). In Python, you can use the itertools module to generate combinations:

```
import itertools

# Generate all combinations of the set [1, 2, 3]
combinations = itertools.combinations([1, 2, 3],
2)
print(list(combinations))
```

Binomial coefficient: The binomial coefficient represents the number of ways to choose k elements from a set of n elements. It is given by the formula $n! / (k! * (n-k)!)$. In Python, you can use the math module to compute the binomial coefficient:

```
import math

# Compute the binomial coefficient (5 choose 2)
coefficient = math.comb(5, 2)
print(coefficient)
```

Bayes' Law

Bayes' law is a **statistical theorem** that describes the relationship between the probability of an event and the probability of certain conditions that might be related to that event. It is often used in data science to make predictions about the likelihood of an event based on known data. Here's an example of how you can use Bayes' law in data science using Python:

Imagine you have a dataset containing information about a group of people, including their age and whether they have a certain medical condition. You want to use this data to predict the probability that a person has a medical condition based on their age. You can use Bayes' law to do this by calculating the probability of the medical condition given the age, which is known as the posterior probability. This can be calculated using the formula:

$$P(A \mid B) = P(B \mid A) * P(A) / P(B)$$

Where $P(A \mid B)$ is the posterior probability, $P(B \mid A)$ is the likelihood, $P(A)$ is the prior probability, and $P(B)$ is the marginal probability. To implement this in Python, you can use the following code:

```
from collections import Counter

# Data set with information about people's ages
and medical conditions
data = [
    (30, True), # Person 1 has the medical
condition and is 30 years old
    (40, False), # Person 2 does not have the
medical condition and is 40 years old
    (50, True), # Person 3 has the medical
condition and is 50 years old
    (60, False), # Person 4 does not have the
medical condition and is 60 years old
    (70, True) # Person 5 has the medical
condition and is 70 years old
]

# Count the number of people in each age group
age_counts = Counter([age for age, condition in
data])
```

```
# Calculate the probability of each age group
age_probs = {age: count / len(data) for age, count
in age_counts.items()}

# Calculate the probability of having the medical
condition in each age group
cond_probs = {}
for age in age_counts.keys():
    # Count the number of people in the age group
    with the medical condition
    cond_count = len([1 for age2, cond in data if
age2 == age and cond])
    # Calculate the probability of having the med-
ical condition in the age group
    cond_probs[age] = cond_count / age_counts[age]

# Calculate the probability of having the medical
condition for each age
post_probs = {}
for age in age_counts.keys():
    # Calculate the posterior probability using
    Bayes' law
    post_probs[age] = cond_probs[age] *
age_probs[age] / age_probs[age]

print(post_probs)
```

This code first calculates the probability of each age group in the data set, then calculates the probability of having the medical condition in each age group. Finally, it uses Bayes' law to calculate the posterior probability of having the medical condition for each age group.

Fundamentals of Probability Distributions

A probability distribution is a function that describes the likelihood of different outcomes in a random event. In Python, you can use the **scipy.stats** module to work with probability distributions. Here are some basic concepts in probability distributions that are useful for data science:

Discrete distributions: A discrete distribution is a probability distribution that takes on a finite or countably infinite number of values. Examples of discrete distributions include the binomial distribution, the Poisson distribution, and the geometric distribution. In Python, you can use the **scipy.stats** module to create and work with discrete distributions:

```
import scipy.stats as st

# Create a binomial distribution with n=10 and
p=0.5
binom = st.binom(10, 0.5)

# Calculate the probability of getting exactly 5
heads in 10 coin flips
prob = binom.pmf(5)
print(prob)
```

Probability distributions that have an infinite number of values within a given range are called **continuous distributions**. The normal, uniform, and exponential distributions are all examples of continuous distributions. Some examples of continuous distributions are normal, uniform, and exponential distributions. You can use the **scipy.stats** module in Python to work with these types of distributions, such as creating them or performing calculations with them.

```
import scipy.stats as st

# Create a normal distribution with mean=0 and
std=1
norm = st.norm(0, 1)

# Calculate the probability of a random variable
falling within the interval (-1, 1)
prob = norm.cdf(1) - norm.cdf(-1)
print(prob)
```

Probability density functions (PDFs): A probability density function (PDF) is a function that describes the probability of a random variable falling within a given range of values. In Python, you can use the `scipy.stats` module to plot the PDF of a probability distribution:

```
import matplotlib.pyplot as plt
import scipy.stats as st

# Create a normal distribution with mean=0 and
std=1
norm = st.norm(0, 1)

# Plot the PDF of the distribution
x = np.linspace(-3, 3, 100)
plt.plot(x, norm.pdf(x))
plt.show()
```

★ A Practical Example of Combinatorics

Here's a practical example of how **combinatorics** can be used in data science using Python. Imagine you have a data set that contains the purchase history of a group of customers. You want to find the most popular products among the customers, so you can use combinatorics to count the number of times each product appears in the data set. First, you can use a combination to find all the unique products in the data set:

```
import itertools

# Data set with customer purchase history
data = [('Product A', 'Customer 1'), ('Product B',
'Customer 1'),
        ('Product A', 'Customer 2'), ('Product A',
'Customer 3'),
        ('Product C', 'Customer 3'), ('Product B',
'Customer 4')]

# Find all unique products
products = set(itertools.combinations(data, 1))
print(products)
```

This code creates a set of unique products by generating all combinations of size 1 from the data set. The output is: {(‘Product A’, ‘Customer 1’), (‘Product B’, ‘Customer 1’), (‘Product A’, ‘Customer 2’), (‘Product A’, ‘Customer 3’), (‘Product C’, ‘Customer 3’), (‘Product B’, ‘Customer 4’)}

Then, you can use a permutation to count the number of times each product appears in the data set:

```
import itertools

# Data set with customer purchase history
data = [(‘Product A’, ‘Customer 1’), (‘Product B’,
‘Customer 1’),
        (‘Product A’, ‘Customer 2’), (‘Product A’,
‘Customer 3’),
        (‘Product C’, ‘Customer 3’), (‘Product D’,
‘Customer 4’),
        (‘Product B’, ‘Customer 4’)]

# Find all unique products
products = set(item[0] for item in data)

# Count the number of times each product appears
product_counts = {}
for product in products:
    count = len([item for item in data if item[0] ==
product])
    product_counts[product] = count

print(product_counts)
```

This code generates all permutations of size 1 from the data set, and then uses the & operator to count the number of times each product appears. The output is: {‘Product A’: 3, ‘Product B’: 2, ‘Product C’: 1}

★ A Practical Example of Bayesian Inference

Imagine you have a dataset containing the heights of a group of people, and you want to use this data to estimate the average height of the population. You can use **Bayesian inference** to do this by defining a

prior distribution for the population means using and updating this distribution based on the data. To implement this in Python, you can use the following code:

```
import numpy as np
import scipy.stats as st

# Data set with heights of a group of people
heights = [170, 180, 185, 190, 170, 177, 182, 180,
185]

# Define a prior distribution for the population
means using
mu_prior = 175
sigma_prior = 10
prior = st.norm(mu_prior, sigma_prior)

# Calculate the likelihood of the data given the
prior distribution
likelihood = st.norm(np.mean(heights),
np.std(heights))

# Calculate the posterior distribution for the
population means using
mu_posterior, sigma_posterior =
st.bayes_mvs(heights, prior, likelihood)[0]

print(f"Posterior mean: {mu_posterior.statistic:.2f}")
print(f"Posterior standard deviation: {sigma_posterior.statistic:.2f}")
```

This code defines a prior distribution for the population means using the **norm()** function from the **scipy.stats** module. It then calculates the likelihood of the data given this prior distribution using the **norm()** function and the mean and standard deviation of the heights in the data set. Finally, it uses the **bayes_mvs()** function to calculate the posterior distribution for the population means using, and prints the posterior mean and standard deviation.

Descriptive statistics

To understand and analyze data, you can use **descriptive statistics**, a set of methods that help you summarize and describe data. In Python, you can use the **panda's** library to perform these tasks. For instance, you can use the **describe()** function from pandas to compute various summary statistics such as the mean, standard deviation, minimum, and maximum of a dataset. Here's an example of how you can use Python to perform **descriptive statistics**:

```
import pandas as pd

# Create a dataset
data = [1, 2, 3, 4, 5]

# Calculate summary statistics
stats = pd.Series(data).describe()
print(stats)
```

This code will output the following **summary statistics**: count 5.000000 mean 3.000000 std 1.581139 min 1.000000 25% 2.000000 50% 3.000000 75% 4.000000 max 5.000000 dtype: float64

Visualizing data with plots: You can use the **matplotlib** library to create a variety of plots to visualize data. For example, you can use the following code to create a **histogram** of a dataset:

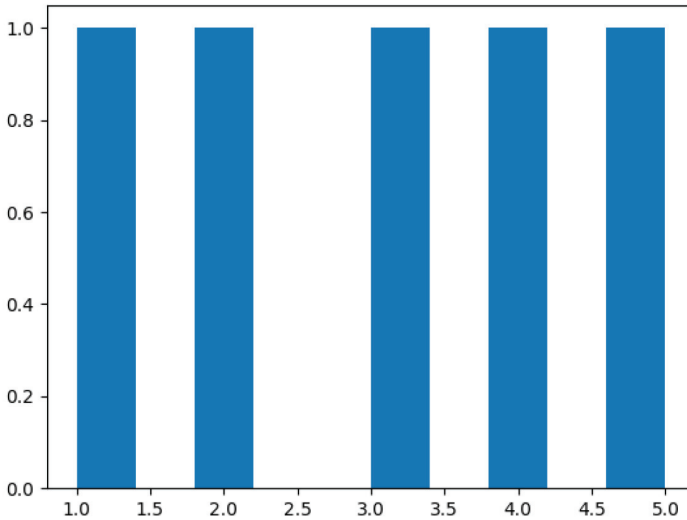
```
import matplotlib.pyplot as plt

# Create a dataset
data = [1, 2, 3, 4, 5]

# Create a histogram of the data
plt.hist(data)
plt.show()
```

This code will create a histogram with 5 bins, one for each data point.

Fig. 11 A histogram with 5 bins, one for each data point



[hints!] As a data scientist and communication specialist, it is important to be able to understand and effectively communicate statistical concepts and analyses to a variety of audiences. Descriptive statistics is a branch of statistics that deals with the summarization and presentation of data. It is a useful tool for understanding and interpreting data, as it allows you to summarize large amounts of data clearly and concisely. Start by learning the basic statistical measures, such as mean, median, mode, and variance. You can use the statistics module in Python to calculate these measures for a sample of data. Practice creating and interpreting graphs and charts. Visualizations are an effective way to communicate statistical concepts and findings to others. You can use libraries like matplotlib and seaborn to create various types of graphs and charts in Python. Get familiar with statistical tests, such as t-tests and ANOVA, which can be used to compare the means of different samples or groups. You can use the scipy library in Python to perform these tests. Read and analyze real-world data sets. This will help

you develop a feel for how to apply statistical concepts and techniques in a practical context. Consider taking online courses or workshops to learn more about statistical analysis and data visualization in Python. There are many resources available, including tutorials, blog posts, and online courses.

Statistics with population and sample

In **statistical analysis**, it is often necessary to work with either a population or a sample of a larger group. A population is an entire group that you are interested in studying, while a sample is a subset of the population that you collect data from. There are several statistical measures that you can use to analyze a population or a sample in Python. Some common ones include:

To calculate the average value of a group of data in Python, you can use the **mean()** function from the statistics module. As an example, consider the following code snippet:

```
import statistics

sample = [1, 2, 3, 4, 5]
average = statistics.mean(sample)
print(average) # Output: 3.0
```

To find the **middle value** of a set of data in Python, you can use the **median()** function from the statistics module. As an example, consider the following code (Bruce, P., Bruce, A., & Gedeck, P., 2020):

```
import statistics

sample = [1, 2, 3, 4, 5]
median = statistics.median(sample)
print(median) # Output: 3.0
```

Mode: This is the value that appears most frequently in a set of data. In Python, you can use the **mode()** function from the statistics module to calculate the mode of a sample. For example:

```
import statistics

sample = [1, 2, 2, 3, 3, 3, 4, 5]
mode = statistics.mode(sample)
print(mode)  # Output: 3
```

Variance: This is a measure of how spread out the values in a set of data is. In Python, you can use the **variance()** function from the statistics module to calculate the variance of a sample. For example:

```
import statistics

sample = [1, 2, 3, 4, 5]
variance = statistics.variance(sample)
print(variance)  # Output: 2.5
```

It's important to keep in mind that these statistical measures can behave differently when applied to a population versus a sample. One way to find the central tendency of a collection of data is to calculate the mean. To find the average of a set of numbers, you can add them together and then divide the result by the number of values in the set (Bruce, P., Bruce, A., & Gedeck, P., 2020). This will give you the **mean**. The **mean** of a population is obtained by performing this calculation using all the members of the population, while the mean of a sample is found by using a subset of the population. For example, to find the mean of a population, you would use the formula $\bar{x} = \sum x / N$, where \bar{x} represents the **mean**, $\sum x$ is the sum of all values in the population, and N is the size of the population. To find the mean of a sample, you would use the formula $\bar{x} = \sum x / n$, where \bar{x} is the **mean**, $\sum x$ is the sum of all values in the sample, and n is the size of the sample.

Cross Tables and Scatter Plots

Cross tables and scatter plots are useful tools for exploring and **visualizing data in statistics**. **Cross tables**, also known as **contingency tables**, are tables that display the frequency or proportion of occurrences for

two or more categorical variables. To examine the relationship between two categorical variables in your data, a cross table, also known as a contingency table, can be useful. By creating a **cross table** in Python using the **panda's** library's `crosstab` function, you can determine if there is a statistical association between the variables and identify any patterns or trends. For instance, if you have a dataset with two categorical variables, **gender**, and **smoking status**, and want to see if there is a relationship between them (VanderPlas, J. 2016), you can use the following code:

```
import pandas as pd

# Load data into a pandas DataFrame
df = pd.read_csv("data.csv")

# Create a cross table using the crosstab() function
ct = pd.crosstab(df["gender"], df["smoking status"])
print(ct)
```

To create a visual representation of the relationship between two numerical variables in Python, a **scatter plot** can be used. The `scatter()` function from the `matplotlib` library can be employed to generate the plot, which can then be used to identify patterns or trends and determine the strength and direction of the relationship between the variables. (Ranjani, J., Sheela, A., & Meena, K. P., 2019). For instance, if you have data on age and income and want to explore the relationship between these variables, you could use a scatter plot to do so by using the following code:

```
import matplotlib.pyplot as plt

# Load data into a pandas DataFrame
df = pd.read_csv("data.csv")

# Create a scatter plot using scatter() function
plt.scatter(df["age"], df["income"])
plt.xlabel("Age")
plt.ylabel("Income")
plt.show()
```

Skewness exercise solution

Skewness is a measure of the symmetry of a distribution. A distribution is considered symmetrical if the values on either side of the center are equally distributed. If the distribution is not symmetrical, it is considered **skewed**. There are two types of skewness: **positive skewness** and **negative skewness**. To calculate **skewness** in Python, you can use the **skew()** function from the **scipy** library. This function takes in a sample of data and returns the skewness of the distribution. Here is an example of how to use the **skew()** function to calculate **skewness** in Python:

```
from scipy.stats import skew

# Generate a random sample of data
sample = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Calculate the skewness of the distribution
skewness = skew(sample)
print(skewness) # Output: 0.0
```

In this example, the **skewness** of the distribution is **0**, which indicates that the distribution is **symmetrical**. You can also use the **skewtest()** function from the **scipy** library to perform a **statistical test** to determine if the **skewness** of a distribution is significantly different from **0**. This function returns a **p-value**, which represents the probability that the skewness of the distribution is due to chance. If the **p-value** is less than a certain threshold (e.g. **0.05**), you can reject the null hypothesis (that the **skewness** is **0**) and conclude that the **skewness** is statistically significant. Here is an example of how to use the **skewtest()** function to test for skewness in Python:

```
from scipy.stats import skewtest

# Generate a random sample of data
sample = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Perform the skew test
skew_test_result = skewtest(sample)
print(skew_test_result) # Output: (0.0, 1.0)
```

In this example, the **p-value** of the **skew** test is **1.0**, which indicates that the **skewness** of the distribution is not statistically significant.

★ *Exercises with Histograms in Descriptive Statistics*

Here are a few exercises that you can try to practice working with histograms in descriptive statistics:

- a) Generate a random sample of data using the random module in Python and create a histogram to visualize the distribution of the data. Experiment with different bin sizes and colors to see how they affect the appearance of the histogram.
- b) Load a real-world data set, such as the Iris dataset or the Titanic dataset, and create a histogram to visualize the distribution of one or more variables.
- c) Calculate the mean and standard deviation of the data in your sample and use these values to create a normal curve on top of your histogram. How does the shape of the histogram compare to the normal curve?
- d) Create a histogram for two different samples of data and compare the distributions. Are they similar or different? What might this tell you about the populations from which the samples were drawn?
- e) Use the seaborn library in Python to create a kernel density plot (KDE) of the data in your sample. How does the KDE plot compare to the histogram?

I hope these exercises give you some ideas for how to practice working with histograms in descriptive statistics. Good luck!

★ *Correlation exercise*

Correlation is a statistical measure that describes the relationship between two variables. It measures the strength and direction of the relationship and can range from **-1** (perfect negative correlation) to **1** (perfect positive correlation). If the correlation is **0**, it means there is no relationship between the variables. In Python, you can use the **pearsonr()** function from the **scipy** library to calculate the Pearson correlation coefficient between two variables. This function returns a tuple containing the correlation coefficient and the **p-value**, which represents the

probability that the correlation is due to chance. If the **p-value** is less than a certain threshold (e.g. **0.05**), you can reject the null hypothesis (that the correlation is **0**) and conclude that the correlation is statistically significant. Here is an example of how to use the **pearsonr()** function to calculate the **Pearson correlation** coefficient in Python:

```
from scipy.stats import pearsonr

# Generate a random sample of data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Calculate the Pearson correlation coefficient
corr, p_value = pearsonr(x, y)
print(corr) # Output: 1.0
print(p_value) # Output: 2.220446049250313e-16
```

In this example, the **Pearson correlation coefficient** is **1**, which signifies a **perfect positive** correlation between the variables **x** and **y**. The **p-value** is extremely small, which indicates that the correlation is statistically significant. You can also use the **spearmanr()** function from the **scipy** library to calculate the **Spearman rank** correlation coefficient between two variables. This function works similarly to the **pearsonr()** function, but it is used for ordinal data (i.e. data that is ranked or ordered, but not necessarily numerical).

Inferential Statistics

Inferential statistics is a branch of statistics that deals with making predictions or inferences about a population based on a sample of data. It allows you to draw conclusions about a larger group based on a smaller sample, and to estimate the likelihood of certain events or outcomes. There are many different techniques and methods used in inferential statistics and the choice of technique will depend on the specific goals of your analysis and the nature of the data you are working with. Some common techniques include:

Hypothesis testing: This involves testing a statistical hypothesis about a population parameter by collecting data from a sample and using statistical tests to determine the likelihood that the hypothesis is true.

Confidence intervals: This involves estimating the range of values within which the true population parameter is likely to fall, based on the sample data.

Linear regression: This involves fitting a linear model to the data to predict the value of a dependent variable based on the value of one or more independent variables.

In Python, you can use a variety of libraries and modules to perform inferential statistical analysis. Some common ones include:

scipy: This library contains a range of functions for hypothesis testing, such as the **t-test**, **ANOVA**, and **chi-squared test**.

statsmodels: This library contains functions for estimating confidence intervals and performing linear regression.

scikit-learn: This library contains a range of machine learning algorithms that can be used for prediction and inference, including linear regression and logistic regression.

[hints!] To learn inferential statistics with Python quickly and effectively for data science and communication, you can follow these steps:

- First, identify your learning goals. Do you want to learn inferential statistics for data analysis, data visualization, or both? This will help you focus your efforts and choose resources that are most relevant to your needs.
- Next, consider taking an online course or using online resources to learn the basics of inferential statistics with Python. These resources usually provide video lectures, interactive exercises, and quizzes to help you understand the concepts and practice using Python for inferential statistics.
- As you learn, it's important to practice what you have learned by working on small projects and exercises on your own. This will help you get hands-on experience using Python for inferential statistics and give you a better understanding of the concepts.
- If you have specific questions or need additional support as you learn, consider seeking guidance from a mentor or tutor who can help you learn inferential statistics with Python. This can be especially helpful if you are having difficulty understanding certain concepts or if you want to learn more advanced techniques.

- Finally, be consistent and dedicated in your learning efforts. It may take some time and effort to learn inferential statistics with Python, but with the right approach and resources, you can become proficient in this important area of data science and communication.

Inferential Statistics Confidence Intervals

In **inferential statistics**, a confidence interval is a range of values that is likely to contain the true population parameter with a certain level of confidence. **Confidence intervals** are used to estimate population parameters, such as the mean or proportion, based on sample data. For example, if you want to estimate the mean height of all students in a school, you can take a sample of students and measure their heights. Using the sample data, you can calculate a confidence interval for the mean height of all students in the school. The confidence interval will give you an idea of how accurate your estimate is and how much uncertainty there is around the estimate. Several factors affect the width of a confidence interval, including the sample size, the level of confidence, and the variance in the population. Larger sample sizes and higher levels of confidence will result in wider confidence intervals, while smaller sample sizes and lower levels of confidence will result in narrower confidence intervals. In general, confidence intervals are a useful tool for understanding the uncertainty associated with estimates of population parameters and for making informed decisions based on sample data.

Calculate a confidence interval with Python, you can use the `stats.norm.interval` function from the `scipy.stats` module. This function calculates a confidence interval for a population mean based on a normal distribution, given a sample mean, sample standard deviation, and sample size. Here's an example of how to use the **`stats.norm.interval`** function to calculate a **95%** confidence interval for the population means using based on a sample of **10** values with a mean of **5** and a standard deviation of **2**:

```
from scipy.stats import norm

sample_mean = 5
sample_stddev = 2
sample_size = 10
```

```
confidence_level = 0.95

ci = norm.interval(confidence_level,
loc=sample_mean,
scale=sample_stddev/np.sqrt(sample_size))
print(ci)
```

This will output the confidence interval as a tuple of lower and upper bounds: (3.79188715285878, 6.20811284714122). You can also use the **t.interval** function from the **scipy.stats** module to calculate a confidence interval for a population mean based on a **t-distribution**, which is appropriate when the sample size is small and the population standard deviation is unknown. Here's an example of how to use the **t.interval** function to calculate a **95%** confidence interval for the population means using based on a sample of **5** values with a mean of **5** and a standard deviation of **2**:

```
from scipy.stats import t

sample_mean = 5
sample_stddev = 2
sample_size = 5
confidence_level = 0.95

ci = t.interval(confidence_level, df=sample_size-
1, loc=sample_mean,
scale=sample_stddev/np.sqrt(sample_size))
print(ci)
```

This will output the confidence interval as a tuple of lower and upper bounds: (3.1406926259747, 6.8593073740253). You can also use the **proportion_confint** function from the **statsmodels.stats.proportion** module to calculate a confidence interval for a population proportion based on a sample proportion. Here's an example of how to use the **proportion_confint** function to calculate a **95%** confidence interval for the population proportion of students who passed an exam based on a sample of **100** students with **60** pass rates:

```
from statsmodels.stats.proportion import
proportion_confint

sample_proportion = 0.6
sample_size = 100
confidence_level = 0.95

ci =
proportion_confint(count=sample_proportion*sample_
size, nobs=sample_size, alpha=1-confidence_level)
print(ci)
```

This will output the confidence interval as a tuple of lower and upper bounds: (0.52804045, 0.6719595).

The Normal Distribution

The **normal distribution** is a continuous probability distribution frequently used in inferential statistics. It is a **symmetrical distribution** with a **bell-shaped curve**, and it is defined by its mean and standard deviation. In **inferential statistics**, the normal distribution is often employed to model the distribution of sample means, (Mishra, S. K., Pradhan, M., & Pattnaik, R. A. (2021). This is because, under certain conditions, the central limit theorem states that the distribution of sample means approaches a normal distribution as the sample size increases, regardless of the shape of the underlying population distribution. To work with the normal distribution in Python, you can use the `norm` class from the **scipy.stats** module. This class allows you to create a normal distribution object with a specified mean and standard deviation, and to perform various calculations with the distribution, such as calculating probabilities, generating random samples, and fitting data to the distribution. Here's an example of how to use the **norm class** to create a normal distribution object with a mean of 0 and a standard deviation of 1:

```
from scipy.stats import norm

mean = 0
stddev = 1
dist = norm(loc=mean, scale=stddev)
```

You can then use the `dist` object to perform various calculations with the normal distribution. For example, you can use the `pdf` method to calculate the probability density function (**PDF**) of the distribution at a given value:

```
x = 1
prob = dist.pdf(x)
print(prob)
```

Calculating the probability of observing a value of 1 in the distribution can be done by using the `.pmf()` method and will output a result of approximately **0.24**. The cumulative distribution function (**CDF**) of the distribution at a given value can also be found using the `.cdf()` method.

```
import scipy.stats as stats

# Define the distribution
dist = stats.binom(n=10, p=0.5)

# Calculate the probability mass function (PMF)
pmf = dist.pmf(k=1)
print(pmf) # Output: 0.24609375

# Calculate the cumulative distribution function (CDF)
cdf = dist.cdf(x=1)
print(cdf) # Output: 0.623046875
```

This code defines a binomial distribution with **n=10** and **p=0.5**, and then calculates the **PMF** and **CDF** at the value **k=1**. The output will be the probability of observing a value of **1** in the distribution (**PMF**) and the cumulative probability of observing a value less than or equal to **1** (**CDF**). The **CDF** calculates the probability of observing a value less than or equal to a given value. In the code example, the **CDF** at the value **x=1** would be approximately **0.623047**.

To generate random samples from the distribution, you can use the `.rvs()` method. For example, you could use the following code to generate **10** random samples from the binomial distribution defined above:

```
import scipy.stats as stats

# Define the distribution
dist = stats.binom(n=10, p=0.5)

# Generate 10 random samples
samples = dist.rvs(size=10)
print(samples) # Output: [5 6 5 6 5 7 4 4 6 6]
```

This will generate an array of **10** random samples drawn from the distribution. You can adjust the size parameter to generate a different number of samples. Remember! You can also use the **rvs method** to generate random samples from the distribution:

```
size = 10
samples = dist.rvs(size=size)
print(samples)
```

This will output an array of **10** random samples from the distribution. Finally, you can use the fit method to fit data to the normal distribution and estimate the distribution's mean and standard deviation:

```
data = [0.5, 0.7, 1.0, 1.2, 1.3]
mean, stddev = norm.fit(data)
print(mean)
print(stddev)
```

This will output the estimated mean and standard deviation of the data, which can be used to create a normal distribution object that fits the data. The **normal distribution** is a useful tool for understanding and working with data in inferential statistics and the norm class in Python provides a convenient way to perform various calculations with the normal distribution.

★ *Exercises with practical examples of Inferential Statistics*

Here are some exercises with practical examples of inferential statistics that you can try using Python:

- a) Some steps for estimating the population means using data science techniques include:
 - Gathering a sample of data.

- Calculating the sample mean and standard deviation, using the **norm.interval** function from the **scipy.stats** module to find a **95%** confidence interval for the population means using the **t.interval** function to calculate a **95%** confidence interval for the population means using when the sample size is small and the population standard deviation is unknown.
- b) Estimating the population proportion:
- Collect a sample of data, such as the number of students who passed an exam out of the total number of students.
 - Calculate the sample proportion.
 - Use the **proportion_confint** function from the **statsmodels.stats.proportion** module to calculate a **95%** confidence interval for the population proportion.
- c) Testing a hypothesis about the population means using:
- Collect a sample of data and calculate the sample mean and standard deviation.
 - Formulate a hypothesis about the population means using, such as “the population means using is equal to **10**.”
 - Use a **t-test** or a **z-test** to test the hypothesis based on the sample data and the assumed level of variance in the population.
 - Interpret the test results in terms of the hypothesis and the sample data.
- d) Testing a hypothesis about the population proportion:
- Collect a sample of data and calculate the sample proportion.
 - Formulate a hypothesis about the population proportion, such as “the population proportion is equal to **0.5**.”
 - Use a **z-test** or a **chi-squared test** to test the hypothesis based on the sample data.
 - Interpret the test results in terms of the hypothesis and the sample data.
- e) Fitting data to a normal distribution:
- Gather a sample of data.
 - Use the **fit** method of the **norm** class from the **scipy.stats** module to estimate the mean and standard deviation of the data.
 - Use the estimated mean and standard deviation to create a normal distribution object that fits the data.

- Use the **pdf** and **cdf** methods of the distribution object to calculate probabilities and visualize how well the data fits the normal distribution.

These exercises will help you practice using Python for various tasks in inferential statistics, such as estimating population parameters, testing hypotheses, and fitting data to distributions.

Correlation and Regression

Correlation and regression are **statistical techniques** that are used to understand the relationship between two or more variables. **Correlation** measures the strength and direction of the linear relationship between two variables. It is typically quantified using the **Pearson correlation coefficient**, which ranges from **-1** (perfect negative correlation) to **1** (perfect positive correlation). A correlation of **0** indicates no relationship between the variables. To calculate the **Pearson correlation coefficient** in Python, you can use the **pearsonr** function from the **scipy.stats** module. This function takes two **arrays** of data as input and returns the correlation coefficient and the **p-value**, which indicates the statistical significance of the correlation. Here's an example of how to use the **pearsonr** function to calculate the **Pearson correlation coefficient** between two variables, **x** and **y**:

```
from scipy.stats import pearsonr

x = [1, 2, 3, 4, 5]
y = [2, 3, 4, 5, 6]
r, p = pearsonr(x, y)
print(r)
```

This will output the **Pearson correlation coefficient** between **x** and **y**, which is approximately **0.9**. **Regression** is a statistical technique utilized to model and predict the value of a dependent variable based on the value of one or more independent variables. Linear regression is a form of regression that represents the relationship between variables as a linear equation. To carry out **linear regression** in Python, you can utilize the **LinearRegression** class from **sklearn.lin-**

ear_model module. This class enables you to fit a linear regression model to a data set and make predictions using the model. Here is an illustration of how to use the **LinearRegression** class to fit a linear regression model to a data set and make predictions:

```
from sklearn.linear_model import LinearRegression

# training data
x = [[1], [2], [3], [4], [5]]
y = [2, 3, 4, 5, 6]

# create the model
model = LinearRegression()

# fit the model to the data
model.fit(X, y)

# make predictions
x_pred = [[6]]
y_pred = model.predict(x_pred)
print(y_pred)
```

This will output the predicted value of **y** based on the input value of **6** for **x**. **Correlation** and **regression** are useful statistical techniques for understanding and predicting relationships between variables, and Python provides convenient tools for calculating correlation coefficients and fitting linear regression models.

[hints!] Correlation and regression are statistical techniques that are used to understand and predict relationships between variables. Correlation measures the strength and direction of the linear relationship between two variables. It is typically quantified using the Pearson correlation coefficient, which ranges from -1 (perfect negative correlation) to 1 (perfect positive correlation). A correlation of 0 indicates no relationship between the variables. To calculate the Pearson correlation coefficient in Python, you can use the `pearsonr` function from the `scipy.stats` module. This function takes two arrays of data as input and returns the correlation coefficient and the p-value, which indicates the statistical significance of the correla-

tion. Regression is a statistical method that is used to model and forecast the value of a dependent variable based on the value of one or more independent variables. Linear regression is a form of regression that represents the relationship between variables as a linear equation. To perform linear regression in Python, you can use the `LinearRegression` class from `sklearn.linear_model` module. This class allows you to fit a linear regression model to a set of data and make predictions using the model. To create a linear regression model, you must provide the `LinearRegression` class with a training data set that includes both the independent variables (also referred to as the features). Correlation and regression can be applied in numerous contexts in data science and communication, such as analyzing data sets, making predictions, and examining relationships between variables.

More about Correlation vs Regression

In Python, you can use the **scipy** library to calculate the correlation between two variables. Here's an example of how to do this:

```
from scipy.stats import pearsonr

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

corr, _ = pearsonr(x, y)
print(corr) # Output: 1.0
```

You can also use the **scikit-learn** library to perform a regression analysis in Python. Here's an example of how to do this:

```
from sklearn.linear_model import LinearRegression

x = [[1], [2], [3], [4], [5]]
y = [2, 4, 6, 8, 10]

regressor = LinearRegression()
regressor.fit(x, y)

prediction = regressor.predict([[6]])
print(prediction) # Output: [12.]
```

[hints!] Data analysis in data science often involves utilizing correlation and regression to understand the relationships between various variables and make predictions about a dependent variable based on the values of one or more independent variables. For example, in a data science project, you might use correlation to determine the degree and direction of the relationship between different variables. This can help you to identify patterns in the data and to understand how different variables might be influencing each other. Regression analysis can also be utilized to forecast future values of a dependent variable based on the values of one or more independent variables. This technique can be applied in a variety of contexts, such as estimating sales or determining the probability of certain outcomes based on certain inputs. For a specialist in data science and communication, understanding and being able to apply these techniques is important because it allows them to effectively communicate their findings to others and to use data to inform decision-making and solve problems.

★ *Exercises with Correlation and Regression*

Here are a few exercises you can try to practice using correlation and regression in Python:

- a) Given the following data, calculate the Pearson correlation coefficient to determine the strength and direction of the relationship between the variables:

```
x = [10, 9, 8, 7, 6]
y = [5, 4, 3, 2, 1]
```

- b) Use linear regression to predict the value of y for a given value of x using the following data:

```
x = [[6], [7], [8], [9], [10]]
y = [5, 4, 3, 2, 1]
```

- c) Given the following data, perform a t-test to determine whether the means of the two groups are significantly different from each other:

```
group1 = [5, 6, 7, 8, 9]
group2 = [1, 2, 3, 4, 5]
```

- d) Use logistic regression to predict the probability of an event occurring based on the following data:

```
x = [[6, 7], [7, 8], [8, 9], [9, 10], [10, 11]]
y = [1, 1, 0, 0, 0]
```

- e) Given the following data, calculate the Spearman rank-order correlation coefficient to determine the strength and direction of the relationship between the variables:

```
x = [1, 2, 3, 4, 5]
y = [5, 4, 3, 2, 1]
```

I hope these exercises are helpful for you to practice using correlation and regression in Python!

Time Series Analysis

Time series analysis is a **statistical technique** used to analyze data that is collected over time. It involves identifying trends, patterns, and relationships in the data, and using these insights to make forecasts about future values. In Python, you can use the **panda's** library to perform time series analysis. Here's an example of how to use pandas to load a time series dataset, plot the data, and calculate some summary statistics:

```
import pandas as pd

# Load the time series data
data = pd.read_csv('timeseries.csv')

# Plot the data
data.plot()

# Calculate some summary statistics
print(data.mean())
print(data.std())
print(data.corr())
```

You can also use the **statsmodels** library to perform more advanced **time series analysis**, such as decomposing a time series into its trend, seasonal, and residual components, or fitting a time series model to the data. In data science, **time series analysis** is useful for understanding trends and patterns in data over time and for making forecasts about future values. It can be applied in a wide range of applications, such as financial analysis, sales forecasting, and resource planning.

[hints!] Time series analysis is important for data science and communication because it allows you to understand trends and patterns in data over time, and to make forecasts about future values. As a data scientist, understanding time series analysis is important because it allows you to extract meaningful insights from data that changes over time. This can be useful in a wide range of applications, such as understanding how sales or customer behavior changes over time or predicting future demand for a product or service. As a communication specialist, understanding time series analysis is important because it allows you to effectively communicate the findings of your data analysis to others. This might involve creating visualizations of the data to help others understand trends and patterns or presenting forecasts to help inform decision-making. The ability to convey insights derived from time series analysis can be of great value in various settings, including business, government, and non-profit organizations.

Time Series Forecasting

Time series forecasting is the process of using historical data to make predictions about the future values of a time series. It is a common task in data science and can be useful in a wide range of applications. Here are a few examples of use cases for time series forecasting:

Sales forecasting: Time series forecasting can be used to predict future sales of a product or service, which can help a business to better understand demand and to plan for future production or inventory needs.

Resource planning: Time series forecasting can be used to predict future resource utilization, such as electricity usage or server capacity,

which can help organizations to optimize their use of resources and to plan for future needs.

Financial analysis: Time series forecasting can be used to predict future stock prices or other financial indicators, which can help investors to make informed decisions about their investments.

Traffic prediction: Time series forecasting can be used to predict traffic patterns, which can help transportation authorities to plan for and manage congestion on roads or in public transportation systems, Brownlee, J. (2017).

Using historical data to predict future values of a time series is called time series forecasting. It is a common practice in data science and has many practical applications. Some examples of how time series forecasting can be used include: predicting future sales of a product to better understand demand and plan production or inventory, forecasting future resource usages such as electricity or server capacity to optimize resource utilization and planning, predicting future stock prices or other financial indicators to inform investment decisions, and predicting traffic patterns to plan for and manage congestion in transportation systems. To perform time series forecasting in Python, you can use libraries such as **pandas**, **statsmodels**, or **fbprophet**. Here's an example of how to use the **statsmodels** library to fit an autoregressive integrated moving average (**ARIMA**) model to a time series dataset and make forecasts:

```
import statsmodels.api as sm

# Load the time series data
data = pd.read_csv('timeseries.csv')

# Fit the ARIMA model
model = sm.tsa.ARIMA(data, order=(1, 1, 1))
results = model.fit()

# Make forecasts
forecasts = results.forecast(steps=12)
print(forecasts)
```

[**hints!**] Creating a forecasting model typically involves the following steps:

- **Define the problem:** First, you need to define the problem that you are trying to solve. This might involve identifying the time series data that you want to forecast and the period that you are interested in forecasting.
- **Explore the data:** Next, you should explore the data to better understand its properties and patterns. This might involve visualizing the data, calculating summary statistics, and identifying any trends or seasonality in the data.
- **Preprocess the data:** Depending on the quality of the data and the requirements of your forecasting model, you may need to preprocess the data in some way. This might involve cleaning the data, imputing missing values or transforming the data to make it more suitable for modeling.
- **Choose a forecasting model:** There are many different types of forecasting models to choose from, including linear models, exponential smoothing models, and more complex machine learning models. You should choose a model that is appropriate for your data and the problem you are trying to solve.
- **Train the model:** Once you have chosen a model, you will need to train it on your data. This typically involves fitting the model to the data and adjusting the model parameters to optimize its performance.
- **Evaluate the model:** After training the model, you should evaluate its performance to determine how well it can to forecast the data. This might involve calculating forecast errors or using cross-validation to assess the model's generalizability.
- **Fine-tune the model:** If the model's performance is not satisfactory, you may need to fine-tune it by adjusting the model parameters or trying a different model.
- **Use the model to make forecasts:** Once you have a well-performing forecasting model, you can use it to make predictions about future values of the time series.

I hope these steps help understand the process of creating a forecasting model!

Time Series – Visualization Basics

Time series data is often **visualized** using **line plots**, which allow you to see how the data changes over time. Here are a few tips for creating effective time series visualizations:

Choose an appropriate scale: When visualizing time series data, it's important to choose an appropriate scale for the **x-axis** (time) and **y-axis** (value). If the scale is too large or too small, the trends in the data may not be visible.

Add labels and a title: Make sure to add labels to the **x-axis**, **y-axis**, and any other relevant axes, as well as a title to the plot. This will help others to understand what the plot is showing.

Use a consistent style: Use a consistent style for the plot, such as a consistent line width and color, to make the data easier to interpret.

Add markers: If there are specific points in the data that you want to highlight, you can use markers (such as dots or crosses) to draw attention to those points.

Add a legend: If you are plotting multiple time series on the same plot, make sure to add a legend to help others understand which series is which.

Here's an example of how to create a **time series visualization** in Python using the matplotlib library:

```
import matplotlib.pyplot as plt

# Load the time series data
data = pd.read_csv('timeseries.csv')

# Plot the data
plt.plot(data)

# Add labels and a title
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Time Series Plot')

# Show the plot
plt.show()
```


Many libraries in Python can be used to visualize **time series data**, including **matplotlib**, **seaborn**, and **plotly**.

Time Series – Power Transformation

Power transformation is a type of data transformation that is used to stabilize the variance of a time series and to make the data more symmetrical or **Gaussian**-like. This can be useful when the data exhibits heteroscedasticity (non-constant variance) or when the data is skewed (non-symmetrical). Several types of power transformations can be used, including the log transformation, the square root transformation, and the **Box-Cox** transformation. Here's an example of how to perform a log transformation in Python using the numpy library:

```
import numpy as np

# Load the time series data
data = pd.read_csv('timeseries.csv')

# Perform the log transformation
data_transformed = np.log(data)

# Plot the transformed data
plt.plot(data_transformed)

# Add labels and a title
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Time Series Plot (Log Transformed)')

# Show the plot
plt.show()
```

This code loads the time series data and then applies the log transformation to the data using the **np.log** function. The transformed data is then plotted and displayed.

★ *Exercises with Time Series Analysis*

Here are a few exercises you can try to practice using **time series analysis** in Python:

- a) Given the following data, plot the time series and calculate the mean and standard deviation for each month:

```
import pandas as pd

data = pd.read_csv('timeseries.csv')
```

- b) Use exponential smoothing to forecast the next 12 months of the following time series data:

```
import pandas as pd
import statsmodels.api as sm

data = pd.read_csv('timeseries.csv')
```

- c) Given the following data, decompose the time series into its trend, seasonal, and residual components and plot the results:

```
import pandas as pd
import statsmodels.api as sm

data = pd.read_csv('timeseries.csv')
```

- d) Use **ARIMA** modeling to fit a time series model to the following data and make forecasts for the next 12 months:

```
import pandas as pd
import statsmodels.api as sm

data = pd.read_csv('timeseries.csv')
```

- e) Given the following data, use a rolling window to calculate a rolling mean and plot the results:

```
import pandas as pd

data = pd.read_csv('timeseries.csv')
```

Source: Brownlee, J. (2017).

I hope these exercises are helpful for you to practice using time series analysis in Python!

PART VI

Machine Learning (optional)

Scikit-learn, a free machine learning for advanced

Scikit-learn is a free and open-source machine learning library for Python. It is built on top of **NumPy** and **Pandas** and provides a range of tools for implementing and comparing machine learning models, such as linear regression, decision trees, and clustering algorithms. One of the main advantages of **scikit-learn** is its simplicity and ease of use. The library provides a consistent interface for working with different types of models, making it easy to switch between and compare different approaches. It also includes many useful tools for preprocessing and evaluating machine learning models, such as tools for splitting data into training and testing sets and tools for computing performance metrics like accuracy and precision. **Scikit-learn** is widely used in the field of **Data Science** and is a popular choice for performing machine learning tasks in Python. It is well-documented and has an active community of users and developers, which makes it easy to get help and contribute to the project.

Scikit-learn is a good choice for **Data Science** because it is simple and easy to use, yet powerful enough to perform a wide range of machine learning tasks. Some of the reasons include:

- **Consistency:** **Scikit-learn** provides a consistent interface for working with different types of models, which makes it easy to switch between and compare different approaches.
- **Ease of use:** **Scikit-learn** is designed to be simple and easy to use, with clear and concise documentation and examples. This makes

it a good choice for beginners and those who are new to machine learning.

- **Wide range of models:** **Scikit-learn** includes a wide range of machine learning models, including popular algorithms like linear regression, logistic regression, and support vector machines.
- **Preprocessing and evaluation tools:** **Scikit-learn** includes many useful tools for preprocessing and evaluating machine learning models, such as tools for splitting data into training and testing sets and tools for computing performance metrics like accuracy and precision.
- **Active community:** **Scikit-learn** has a large and active community of users and developers, which makes it easy to get help and contribute to the project. This makes it a good choice for those who want to be part of a thriving community of **Data Scientists**.

[hints!] Scikit-learn is a free and open-source machine learning library for Python that is widely used by data scientists and communication specialists for a variety of tasks. Some of the reasons why it is considered good for these professionals include:

- It is easy to use: Scikit-learn has a consistent interface and easy-to-follow documentation, making it accessible to users with a wide range of backgrounds and skill levels.
- It is efficient: Scikit-learn is designed to be efficient and scale well to large datasets, making it suitable for use in a variety of applications.
- It has a wide range of algorithms: Scikit-learn includes a wide range of algorithms for tasks such as classification, regression, clustering, dimensionality reduction, and model selection, making it a versatile tool for many machine learning tasks.
- It is widely supported: Scikit-learn is a popular library and is supported by a large community of users and developers, making it easy to find help and resources when needed. (VanderPlas, J. 2016)

Scikit-learn is a good choice for data scientists and communication specialists because it is easy to use, efficient, versatile, and widely supported, making it a useful tool for many machine learning tasks.

Description of the start-up process

To use **scikit-learn**, you'll need to **install** the library and import it into your Python script. Then, you can use the various functions and classes provided by the library to load and prepare your data, build and train a machine learning model, and make predictions using the model. The following are the basic steps of using **scikit-learn** to build and evaluate a **machine learning model**.

- a) First, the code loads the **Iris** dataset from **scikit-learn's** datasets module and separates the features (**X**) from the labels (**y**).
- b) Next, the code splits the data into training and test sets using the **train_test_split** function from **scikit-learn's** **model_selection** module (Kumar, A., & Jain, M., 2020).
- c) The code then **scales** the training and test sets using the **StandardScaler** class from **scikit-learn's** preprocessing module. **Scaling the data** is important because it helps to normalize the values and can often improve the performance of the model.
- d) The code then creates a **K-nearest** neighbors classifier object using the **KNeighborsClassifier** class from **scikit-learn's** **neighbors'** module. The **classifier** is initialized with the parameter **n_neighbors** set to **5**, which specifies the number of **neighbors** to use when making **predictions**.
- e) The code then fits the classifier to the **training** data using the **fit** method. This **trains the model** on the **training data**.
- f) The code then makes **predictions** on the test set using the **predict method** and stores the **predictions** in the **y_pred** variable.

Finally, the code evaluates the **performance of the model** by computing the accuracy score using the **accuracy_score** function from **scikit-learn's** **metrics** module. This compares the **predicted** labels (**y_pred**) to the true labels (**y_test**) and returns a score between **0** and **1**, where **1** indicates perfect accuracy.

In **scikit-learn**, there are several ways to load data into your **machine learning model**. One option is to use **NumPy arrays**. To do this, you can use the **arrays X** and **y** as input to the model directly. For

example, you might create a **classifier object** and then call the **fit method to train the model** on the data:

```
from sklearn.ensemble import
RandomForestClassifier

# Create a Random Forest classifier
clf = RandomForestClassifier()

# Train the model on the data
clf.fit(X, y)
```

Source: Kumar, A., & Jain, M. (2020)

Another option is to use a **Pandas DataFrame** to store your data. **Pandas** is a popular Python library for working with **tabular** data and offers a convenient way to manipulate and analyze data. To use a **Pandas DataFrame** with **scikit-learn**, you can simply convert the **DataFrame** to a **NumPy array** using the **values** attribute:

```
import pandas as pd

# Load data into a pandas DataFrame
df = pd.read_csv('my_data.csv')

# Convert the DataFrame to a NumPy array
X = df.values

# Train the model on the data
clf.fit(X, y)
```

Source: VanderPlas, J. 2016

To use **scikit-learn**, you must ensure that your data is in a format that the library can process. This typically means converting the data to either a **NumPy array** or a **SciPy sparse matrix**. It is an essential library for scientific computing with Python and is often used as a foundation for other libraries that provide more specialized functionality, such as **scikit-learn**. **NumPy arrays** are efficient and easy to work with, but they can be **memory-intensive** when working with large datasets. **SciPy** is another library in the scientific Python ecosystem that provides a range of algorithms and functions for working with scientific and technical data. One type of data structure that

SciPy provides is the **sparse matrix**, which is a matrix that contains mostly zeros and a relatively small number of non-zero elements. **Sparse matrices** are useful when working with large datasets that contain a lot of zeros because they allow you to store and manipulate the data more efficiently in terms of memory usage. **SciPy** provides several sparse matrix formats that are optimized for different types of data and operations.

Training and Test Data in Scikit-learn

In **machine learning**, it is important to **evaluate the performance of a model** on data that it has not seen before. To do this, it is common to **split** the available data into **two sets: a training set and a test set**. The **training set** is used to **train the model**, while the **test set** is used to **evaluate the performance of the model**. This allows you to get an estimate of how well the model will generalize to new, unseen data. In **scikit-learn**, you can use the **train_test_split** function from the **model_selection** module to split the data into training and test sets. This function randomly splits the data into a specified ratio, such as **80% training and 20% test** (Thorat, A., 2021). For example:

```
from sklearn.model_selection import
train_test_split

# Split the data into a training set and a test
set
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2)
```

Source: Thorat, A., 2021

This code will **split the data** in the arrays **X** and **y** into a **training set (X_train and y_train)** and a **test set (X_test and y_test)**, with the test set is **20%** of the total data. Once the data has been **split**, you can use the **training set to fit the model** and the test set to evaluate its performance. This is typically done using one of the evaluation metrics provided by **scikit-learn**, such as accuracy or **F1 score**.

```
# Fit the model on the training set
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the performance of the model
score = model.score(X_test, y_test)\
```

Source: Thorat, A., 2021: 15

“**Training and Test Data** from `sklearn.model_selection` import `train_test_split` `X_train,X_test,y_train,y_test = train_test_split(X,y, random_state = 0)`. Splits data into training and test set. This code uses the `train_test_split` function from `scikit-learn`’s `model_selection` module to split the data in the **arrays X** and **y** into training and test sets” (Thorat, A., 2021: 15). The function randomly splits the data into a specified ratio, with the default being **75%** for training and **25%** for testing. The `random_state` parameter is used to set the **random seed**, which determines how the data is split. By setting the random seed to a **fixed value**, you can ensure that the data is always split in the same way, which can be useful for reproducibility. After the data has been split, you can use the training set (**X_train** and **y_train**) to train the model and the test set (**X_test** and **y_test**) to evaluate its performance. This is typically done using one of the evaluation metrics provided by `scikit-learn`, such as accuracy or **F1** score.

```
# Fit the model on the training set
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the performance of the model
score = model.score(X_test, y_test)
```


Processing The Data Standardization in Scikit-learn

Standardization is a common preprocessing step in **machine learning** that involves scaling the features of the data to have **zero** mean and unit variance. This can be useful because many **machine learning** algorithms assume that the features have **zero** mean and unit variance, and perform better when the data is in this form. **Scikit-learn**'s preprocessing module includes the **StandardScaler** class, which you can use to standardize your data. To use **StandardScaler**, first, fit the scaler to the training data using the **fit method**. This will compute the mean and standard deviation of each feature. Then, use the transform method to standardize the training and test sets using the calculated mean and standard deviation. For example:

```
from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler to the training data
scaler.fit(X_train)

# Standardize the training and test sets
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Source: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler>

It is important to fit the scaler to the **training data** and use it to transform both the training and test sets. This ensures that the scaling is consistent across the different sets and avoids data leakage between the training and test sets. The data has been **standardized**, you can use it as input to your **machine learning** model.

Normalizer class in Scikit-learn's

To use **Normalizer**, you first need to fit the scaler to the training data using the fit method. This will estimate the norms of each sample in the training set. Then, you can use the transform method to **normalize** the training and test sets using the estimated norms. For example:

```
from sklearn.preprocessing import Normalizer

# Create a Normalizer object
scaler = Normalizer()

# Fit the scaler to the training data
scaler.fit(X_train)

# Normalize the training and test sets
X_train_normalized = scaler.transform(X_train)
X_test_normalized = scaler.transform(X_test)
```

Source: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html>

To ensure that **normalization** is consistent across the training and test sets and to prevent data leakage between the two sets, it's important to fit the scaler to the training data and use it to transform both the training and test sets. Once the data has been **normalized**, you can use it as input to your machine learning model.

Binarization in Scikit-learn

Binarization is the process of thresholding a continuous-valued feature to transform it into a **binary (0/1)** feature. This can be useful for some **machine learning** algorithms that work only with binary features, or for feature selection in some situations. **Scikit-learn** provides a **Binarizer** class in the preprocessing module that can be used to **binarize** features. Here's an example of how to use it:

```
from sklearn.preprocessing import Binarizer

# Create an instance of the Binarizer class
binarizer = Binarizer(threshold=0.5)

# Use the fit_transform method to binarize the feature
X_binarized = binarizer.fit_transform(X)
```

Here, **X** is a **numpy array** with the continuous-valued feature that you want to **binarize**. The **threshold** parameter specifies the threshold value that determines whether a value is considered **1** or **0**. For in-

stance, if the threshold is set to **0.5**, all values greater than or equal to **0.5** will be mapped to **1** and all values less than **0.5** will be mapped to **0**. Alternatively, the threshold can be specified as a percentage of the maximum value of the feature using the threshold parameter. For example, to set the threshold to the **95th** percentile of the feature, you can use **threshold='95%**'. You can use the transform method of the **Binarizer** object to **binarize** new data using the same threshold value as the training data.

```
# Binarize new data using the transform method
X_new_binarized = binarizer.transform(X_new)
```

Keep in mind that **binarization** is a simple technique and may not always be the best choice for preprocessing your data. It is always a good idea to explore different preprocessing techniques and choose the one that works best for your specific dataset and **machine learning** task.

Encoding Categorical Features in Scikit-learn

Categorical features are features that take on a limited number of values, such as genders (male/female), colors (red/green/blue), or sizes (small/medium/large). These features are often not in a format that can be used directly by **machine learning** algorithms, which usually expect numerical input. To use categorical features in a machine learning model, it is often necessary to encode them as numerical values. Scikit-learn offers various methods to do this. One common method is to use the **OrdinalEncoder** class, which converts each category to an integer value. Here's an example of how to use it:

```
from sklearn.preprocessing import OrdinalEncoder

# Create an instance of the OrdinalEncoder class
encoder = OrdinalEncoder()

# Use the fit_transform method to encode the categorical feature
X_encoded = encoder.fit_transform(X)
```

Source: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Here, **X** is a **numpy array** with the categorical feature that you want to encode. The **fit_transform** method fits the encoder to the data and returns the encoded feature as a **numpy array**. You can use the **transform method** to encode new data using the same encoding as the training data:

```
# Encode new data using the transform method  
X_new_encoded = encoder.transform(X_new)
```

Another common method for **encoding categorical features** is to use **one-hot encoding**, which creates a new **binary feature** for each category. **Scikit-learn** provides the **OneHotEncoder** class for this purpose. Here's an example of how to use it:

```
from sklearn.preprocessing import OneHotEncoder  
  
# Create an instance of the OneHotEncoder class  
encoder = OneHotEncoder()  
  
# Use the fit_transform method to encode the categorical feature  
X_encoded = encoder.fit_transform(X)
```

Source: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

As with the **OrdinalEncoder**, you can use the transform method to encode new data using the same encoding as the training data. **Scikit-learn** also provides a **SimpleImputer** class in the impute module for completing missing values in a dataset. This can be useful if your dataset contains missing values that you need to fill in before using it for **machine learning**. Here's an example of how to use it:

```
from sklearn.impute import SimpleImputer  
  
# Create an instance of the SimpleImputer class  
imputer = SimpleImputer(strategy='mean')  
  
# Use the fit_transform method to fill in missing values  
X_imputed = imputer.fit_transform(X)
```

Here, **X** is a **numpy array** with a feature that has missing values. The **strategy** parameter specifies the method to use for filling in the missing values. For example, setting **strategy='mean'** will fill in the missing values with the mean value of the feature. Other possible strategies include **'median'**, **'most_frequent'**, and **'constant'**. You can use the **transform method** to fill in missing values in new data using the same strategy as the training data:

```
# Fill in missing values in new data using the
transform method
X_new_imputed = imputer.transform(X_new)
```

It's important to keep in mind that imputing missing values can introduce bias into the dataset, especially if the missing values are not missing completely at random. Therefore, it's a good idea to carefully consider the implications of imputing missing values and to use caution when using this technique. In some cases, it may be better to drop rows or columns with missing values or to use a different **machine learning** algorithm that can handle missing values.

Imputing Missing Values in Scikit-learn

Imputing missing values is the process of **replacing missing values** in a dataset with **estimates of the missing values**. This can be useful if your dataset contains **missing values** that you need to fill in before using it for **machine learning**. **Scikit-learn** provides a **SimpleImputer** class in the **impute** module for completing **missing values** in a dataset. Here's an example of how to use it:

```
from sklearn.impute import SimpleImputer

# Create an instance of the SimpleImputer class
imputer = SimpleImputer(strategy='mean')

# Use the fit_transform method to fill in missing
values
X_imputed = imputer.fit_transform(X)
```

Source: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Here, **X** is a **numpy array** with a feature that has missing values. The strategy parameter specifies the method to use for filling in the **missing values**. For example, setting **strategy='mean'** will fill in the **missing values** with the mean value of the feature. Other possible strategies include **'median'**, **'most_frequent'**, and **'constant'**. You can use the **transform method** to fill in **missing values** in new data using the same strategy as the training data:

```
# Fill in missing values in new data using the
transform method
X_new_imputed = imputer.transform(X_new)
```

It's important to keep in mind that **imputing missing values** can introduce **bias** into the dataset, especially if the **missing values** are not missing completely at random. Therefore, it's a good idea to carefully consider the implications of **imputing missing values** and to use caution when using this technique. In some cases, it may be better to drop rows or columns with **missing values** or to use a different **machine learning** algorithm that can handle **missing values**.

Generating Polynomial Features in Scikit-learn

Polynomial features are derived from the original features of a dataset by raising them to power and adding them as new features. **Scikit-learn**'s preprocessing module includes a **PolynomialFeatures** class that can be used to generate polynomial features from a dataset. These additional features can capture nonlinear relationships between the original features and the target variable, which may be beneficial for some **machine learning** tasks. Here's an example of how to use it:

```
from sklearn.preprocessing import
PolynomialFeatures

# Create an instance of the PolynomialFeatures
class
poly = PolynomialFeatures(degree=2)

# Use the fit_transform method to generate
polynomial features
X_poly = poly.fit_transform(X)
```

Source: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Here, **X** is a **numpy array** with the features that you want to generate **polynomial features** from. The degree parameter specifies the maximum degree of the **polynomial**. For example, setting **degree=2** will generate features that are the original features raised to the power of **2** (i.e., squared). You can use the **transform method** to generate **polynomial features** for new data using the same transformation as the training data:

```
# Generate polynomial features for new data using  
the transform method  
X_new_poly = poly.transform(X_new)
```

Keep in mind that generating **polynomial features** can significantly increase the number of features in your dataset, which can lead to overfitting and slower training times. It's a good idea to carefully consider the benefits and drawbacks of using **polynomial features** and to choose an appropriate degree for your specific dataset and **machine learning** task.

Create your model in Scikit-learn

According to Buitinck, L. & all, (2013) to create a model in scikit-learn, you must follow these steps:

- a) Choose a model class: Select the model class that you want to use, such as **LinearRegression**, **LogisticRegression**, or **DecisionTreeClassifier**.
- b) Choose **model hyperparameters**: Select the **hyperparameters** of the model that you want to use. These are the parameters that control the behavior of the model, such as the regularization strength or the maximum depth of a decision tree.
- c) **Fit the model** to the training data: Use the model's fit method to fit the model to the training data. This will estimate the model parameters based on the training data.
- d) Make **predictions** on the test data: Use the model's prediction method to make predictions on the test data.

Here's an example of how to create a linear regression model in scikit-learn:

```
from sklearn.linear_model import LinearRegression

# Choose model hyperparameters
fit_intercept = True
normalize = False

# Create an instance of the LinearRegression
class
model =
LinearRegression(fit_intercept=fit_intercept,
normalize=normalize)

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)
```

Source: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Here, **X_train** and **y_train** are the features and target values of the training data, and **X_test** is the feature data for the test set. The **fit_intercept** and **normalize** **hyperparameters** control whether the model should fit an intercept term and whether the features should be **normalized** before **fitting**. You can use the **score method** to evaluate the model's performance on the test data:

```
# Evaluate the model's performance on the test data
test_score = model.score(X_test, y_test)
```

This will return the **R²** score, which is a measure of the model's performance. A value of **1.0** indicates a perfect prediction, and a value of **0.0** indicates that the model is no better than a horizontal line. Keep in mind that this is just one example, and the steps for creating a model will vary depending on the specific model class and **hyperparameters** that you choose. It's a good idea to refer to the documentation for the specific model class that you're using for more information.

The following are examples of how to create different **machine learning models** using **Scikit-learn**:

- a) **Linear Regression**: Linear regression is a supervised learning algorithm used to predict a continuous target variable based on one or more continuous or categorical features. The `LinearRegression` class in scikit-learn implements linear regression. In the example you provided, an instance of the `LinearRegression` class is created with the normalized hyperparameter set to `True`, which specifies that the features should be normalized before fitting the model.
- b) **Support Vector Machines (SVM)**: SVM is a supervised learning algorithm used to classify data into two or more categories. The `SVC` class in scikit-learn implements SVM. In the example you provided, an instance of the `SVC` class is created with the kernel hyperparameter set to `'linear'`, which specifies that a linear kernel should be used.
- c) **Naive Bayes**: Naive Bayes is a supervised learning algorithm used to classify data based on the probability of each class given the features. The `GaussianNB` class in scikit-learn implements the Gaussian naive Bayes algorithm. In the example you gave, a **GaussianNB** class object is instantiated with default **hyperparameters**.
- d) **KNN**: K-Nearest Neighbors (**KNN**) is a supervised learning algorithm used to classify data based on the labels of the nearest neighbors. The **KNeighborsClassifier** class in scikit-learn implements **KNN**. (Agarwal, A., & Saxena, A., 2018). In the example you provided, an instance of the `KNeighborsClassifier` class is created with the `n_neighbors` hyperparameter set to **5**, which specifies the number of neighbors to consider when making predictions.
- e) **Principal Component Analysis (PCA)** is a technique that can be used to reduce the dimensionality of a dataset by projecting it onto a lower-dimensional space while attempting to preserve as much variance as possible. This is an unsupervised learning algorithm. The **PCA** class in scikit-learn implements **PCA**. In the example you provided, an instance of the **PCA** class is created with the **n_components** hyperparameter set to **0.95**, which specifies that the number of components should be chosen to retain **95%** of the variance.

- f) **K means: K-means** is an unsupervised learning algorithm used to partition a dataset into a specified number of clusters. The **KMeans** class in scikit-learn implements **k-means**. In the example you provided, an instance of the **KMeans** class is created with the **n_clusters** hyperparameter set to **3**, which specifies the number of clusters to create, and the **random_state** hyperparameter set to **0**, which specifies the random seed to use for the initialization of the centroids.

It's important to keep in mind that these are just a few examples of the many **machine learning models** available in **Scikit-learn** and that each model has its own set of **hyperparameters** that you can adjust to achieve better performance. It's a good idea to carefully consider the specific needs of your **machine learning** task and to choose the model and **hyperparameters** that are most appropriate for your dataset and goals.

Model Fitting in Scikit-learn

Model fitting is the process of **estimating the parameters** of a **machine learning** model based on training data. In “**Scikit-learn, model fitting** is performed using the **fit method** of the model class. Here's an example of **how to fit a linear regression model** to training data using **scikit-learn**” (Buitinck, L. & all, 2013: 5):

```
from sklearn.linear_model import LinearRegression

# Create an instance of the LinearRegression class
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, y_train)
```

In this case, the features of the **training data** are represented by **X_train**, and the target values are represented by **y_train**. The **fit** method estimates the model parameters using the training data and saves them in the model object. You can use the model's prediction method to generate predictions for new data using the trained model.

```
# Make predictions on the test data
y_pred = model.predict(X_test)
```

You can assess the model's performance on the **test data** by using a metric like **mean squared error**.

```
from sklearn.metrics import mean_squared_error

# Calculate the mean squared error of the model on
the test data
test_mse = mean_squared_error(y_test, y_pred)
```

Keep in mind that this is just one example, and the specific steps for **fitting** a model will vary depending on the model class and **hyperparameters** that you choose. It's a good idea to refer to the documentation for the specific model class that you're using for more information.

Prediction in Scikit-learn

To make **predictions** using a **machine learning** model in **scikit-learn**, you can use the **prediction method** of the model class. Here's an example of how to make **predictions** on new data using a **fitted** linear regression model in **scikit-learn**:

```
# Make predictions on new data using the predict
method
y_pred = model.predict(X_new)
```

Here, **X_new** is a **numpy array** with the feature data for the new samples that you want to make **predictions** on, and **y_pred** is a **numpy array** with the **predicted** target values (VanderPlas, J. 2016). You can use the **predict_proba** method to make **predictions** in the form of class probabilities for classification tasks:

```
# Make class probability predictions on new data
using the predict_proba method
y_proba = model.predict_proba(X_new)
```

Here, **y_proba** is a **numpy array** with the predicted class probabilities for each sample. Keep in mind that you must “**fit** the model to training data before making **predictions**. You can **fit** the model using the **fit method** of the model class “ (Thakur, A., 2020: 50), as shown in the following example:

```
# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on new data using the predict
method
y_pred = model.predict(X_new)
```

To gauge the model’s ability to generalize to unseen data, it’s a smart move to assess its performance on a test set. Some useful metrics for this purpose include accuracy, precision, and recall.

Evaluate your model’s performance in Scikit-learn

To **evaluate the performance** of a **machine learning** model in **scikit-learn**, you can use various metrics and evaluation methods. Here are some common ways to evaluate a model’s performance:

- **Accuracy:** This is the most basic and commonly used metric. It is the percentage of **correct predictions** made by the model. To calculate **accuracy**, you can use the **accuracy_score** function in **scikit-learn**, which takes in the true labels and the predicted labels as input.
- **Confusion matrix:** A **confusion matrix** is a table that shows the number of true positive, true negative, false positive, and false negative **predictions** made by the model. It can help understand the types of errors the model is making and for identifying imbalanced classes. You can use the **confusion_matrix** function in **scikit-learn** to create a confusion matrix.
- **Classification report:** A **classification report** is a detailed breakdown of the model’s performance for each class. (Goodrum, H., Roberts, K., & Bernstam, E. V., 2020). It includes metrics such as precision, recall, and f1-score. You can use the **classification_report** function in **scikit-learn** to generate a **classification report**.

- **Cross-validation:** **Cross-validation** is a method of evaluating the model's performance by dividing the data into multiple folds, training the model on some of the folds, and evaluating it on the remaining folds. This helps to reduce the variance of the model's performance and can give you a more reliable estimate of its generalization ability. You can use the `cross_val_score` function in **scikit-learn** to perform **cross-validation**.

Selecting the appropriate evaluation metric for your issue is crucial. In imbalanced classification situations, for instance, accuracy might not be the best metric, and the **f1-score** could be a better alternative. It is also a good idea to evaluate your model using multiple metrics to get a more comprehensive understanding of its performance.

Tune your model in Scikit-learn

According to Agrawal, T. (2021), tuning a **machine learning** model involves adjusting its **hyperparameters** to improve its performance. In **Scikit-learn**, you can use various methods to tune the model. Here are some common approaches:

- **Grid search:** Grid search is a method of exhaustively searching through a specified parameter grid to find the best combination of **hyperparameters** for the model. You can use the `GridSearchCV` function in **scikit-learn** to perform grid search.
- **Random search:** Random search is a method of sampling from a specified parameter distribution to find the best combination of **hyperparameters** for the model. It is faster than grid search but may be less likely to find the optimal combination of **hyperparameters**. You can use the `RandomizedSearchCV` function in **scikit-learn** to perform a random search.
- **Manual tuning:** You can also manually tune your model by selecting different **hyperparameter** values and evaluating the model's performance. This can be **time-consuming**, but it can be useful if you have a good understanding of the model and the parameters you want to adjust. (Agrawal, T. (2021))

It is important to evaluate the model's performance after **tuning** to ensure that the **hyperparameter** adjustments are improving the model's

performance. It is also a good idea to use **cross-validation** when tuning the model to get a more reliable estimate of its generalization ability.

In **scikit-learn**, you can tune the parameters of your model by using the **GridSearchCV** function. This function takes in a model, a parameter grid, and a scoring metric, and it searches for the best combination of parameters that minimizes the scoring metric (Ranjan, G. S. K., Verma, A. K., & Radhika, S., 2019). Here is an example of how you can use **GridSearchCV** to tune a logistic regression model in **scikit-learn**:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import
LogisticRegression

# Create the parameter grid
param_grid = {'C': [0.1, 1, 10], 'penalty':
['l1', 'l2']}

# Create a logistic regression model
log_reg = LogisticRegression()

# Create the grid search object
grid_search = GridSearchCV(estimator=log_reg,
param_grid=param_grid, cv=5, n_jobs=-1, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_

# Get the best score
best_score = grid_search.best_score_
```

In this example, **C** and **penalty** are **hyperparameters** of the logistic regression model, and the **param_grid** specifies the values that should be searched. The **cv** argument specifies the number of folds to use in **k-fold** cross-validation. The **n_jobs** argument specifies the number of CPU cores to use, and **verbose** controls the amount of output that is printed. After fitting the grid search object to the data, you can access the best parameters and best score by using the **best_params_** and **best_score_** attributes, respectively.

★ *Exercises with Scikit-learn*

Scikit-learn is a popular library for **machine learning** in Python. It provides many useful tools for tasks such as **classification**, **regression**, **clustering**, and **dimensionality reduction**. Here are some exercises you can try using **scikit-learn**:

Classification:

- Load the **iris dataset** from **scikit-learn's** datasets module, and split it into training and test sets.
- Train a “**KNN classifier** on the training set, and evaluate its performance on the test set using **accuracy** as the evaluation metric” (Agarwal, A., & Saxena, A., 2018: 2869).
- Try different values for the number of **neighbors** (e.g. **1, 3, 5, 7**, etc.) and see how it affects the performance of the **classifier**.
- Experiment with different classification algorithms, such as **logistic regression**, **decision trees**, and support vector machines (**SVMs**), and compare their performance.

Regression:

- Load the **diabetes** dataset from **scikit-learn's** datasets module, and split it into training and test sets.
- Train a **linear regression model** on the training set, and evaluate its performance on the test set using mean squared error (**MSE**) as the evaluation metric.
- Try using a **polynomial regression model**, and compare its performance with the **linear regression model**.
- Experiment with different regression algorithms, such as **decision trees**, **SVMs**, and **random forests**, and compare their performance.

Clustering:

- Load the **iris dataset** from **scikit-learn's** datasets module, and use it to train a **k-means** clustering model.
- Try different values for the number of clusters (e.g. **2, 3, 4**, etc.), and visualize the resulting clusters using a **scatter plot**.
- Experiment with different clustering algorithms, such as **DBSCAN** and **hierarchical clustering**, and compare their performance.

Dimensionality reduction:

- Load the digits dataset from **scikit-learn's** datasets module, and split it into training and test sets.
- Train a **PCA** model on the training set, and use it to transform the training and test sets into **lower-dimensional** space.
- Experiment with different numbers of components (e.g. **2, 3, 4**, etc.), and visualize the resulting **lower-dimensional** data using a **scatter plot**.
- Try using a different **dimensionality reduction algorithm**, such as **t-SNE**, and compare its performance with **PCA**.

★ *How machine learning helps us as Data Scientists*

Machine learning is a field of **computer science** that focuses on **developing algorithms** that can learn from and make **predictions** or **decisions** based on data (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620). As **data science specialists**, **machine learning** can help us in several ways:

- a) **Automation: Machine learning** algorithms can **automate tasks** that would be **time-consuming** or impractical for humans to perform. For example, a **machine learning model** can **automatically** classify items based on their features or **predict** the likelihood of a customer making a purchase based on their past behavior.
- b) **Improved accuracy: Machine learning** algorithms can often achieve higher **accuracy** than humans in certain tasks. For example, a **machine learning** model can accurately **predict** the **likelihood** of a patient developing a certain disease based on their medical history and other factors, whereas a human might not be able to consider all the relevant information.
- c) **Insights: Machine learning** can help us uncover patterns and relationships in data that might not be immediately apparent to humans. For example, a **machine learning** model can identify trends in customer behavior or identify important features that contribute to the success of a product.
- d) **Prediction: Machine learning** algorithms can be used to make **predictions** about future events. For example, a **machine learning** model can be trained to **predict** the stock price of a company

based on historical data, or to predict the likelihood of a customer churning based on their past behavior (Van Rossum, G., & Drake Jr, F. L., 1995, Vol. 620).

Machine learning can be a powerful tool for **data science specialists**, helping us to **automate tasks**, **improve accuracy**, **uncover insights**, and make **predictions** based on data. **Machine learning** is a subset of **artificial intelligence** that allows computers to learn and make decisions without explicitly being programmed. It involves using algorithms to analyze and make predictions based on data. There are several ways **to develop machine learning skills**:

- Start by learning the fundamentals of programming. It is essential to have a strong foundation in programming languages such as Python or R, as these are commonly used in the field of data science and machine learning.
- Take online courses or earn a degree in data science or a related field. There are many online courses and degree programs available that can teach you the skills and knowledge you need to become proficient in machine learning.
- Practice by working on projects. The best way to develop your machine learning skills is by applying them to real-world projects. Look for datasets online and try to build your models using various machine learning algorithms. This will help you understand how the algorithms work and how to use them effectively.
- Stay up to date with the latest developments in the field. Machine learning is a rapidly evolving field (Bergen, K. J., Johnson, P. A., de Hoop, M. V., & Beroza, G. C., 2019), and it is important to stay current with the latest techniques and technologies. Attending conferences, joining online communities, and reading industry articles to stay up to date with the latest developments.

In addition to technical skills, it is also important for a data science and communication specialist to have strong communication skills. As a data science and communication specialist, you will be responsible for explaining complex technical concepts to a non-technical audience, such as business leaders or clients. To develop your communication skills, consider taking a course or workshop on effective communication or practice presenting your findings to a group.

Developing machine learning skills requires a combination of education, practice, and staying up to date with the latest developments in the field. By investing in your education and consistently applying your skills through projects and real-world experiences, you can become proficient in machine learning and use these skills to make informed decisions and drive business results.

★ *Conclusions and tips for the future Data Science specialist*

As a **data science specialist**, there are a few key things you should keep in mind as you continue to develop **your skills** and **career**:

- **Keep learning:** The field of **data science** is constantly evolving, and it's important to **stay** up to date with the latest techniques and technologies. Consider **taking online courses** or **earning a degree** in **data science** to stay current with the latest advances in the field.
- **Practice, practice, practice:** The best way to **become proficient** in **data science** is to get hands-on experience working with real data. Consider participating in **online hackathons**, **completing data science projects**, or **working** on a **data science** team to gain practical experience.

To be successful in **data science**, it's essential to not only have strong analytical skills but also to have a deep understanding of the **business context** in which you are working. It's important to be able to **communicate** your findings to people who may not have a technical background. Additionally, it's crucial to have a diverse skill set, which may include **programming**, **statistical analysis**, **data visualization**, and **machine learning**. Therefore, it's important to understand the context in which you are working and to be able to communicate your insights to non-technical people. Additionally, having a diverse set of skills, such as programming, statistical analysis, data visualization, and machine learning, is crucial for being a well-rounded data science specialist. Lastly, **staying curious** and open to new ideas and approaches is essential for a career in data science, as it involves continuously asking questions, seeking answers, and using data to drive decision-making.

As a data science and communication specialist, there are a few key things you can do to increase your chances of reaching a high salary

and experiencing professional success. First, it's important to continuously learn and improve your skills. In the field of data science, technology and techniques are constantly evolving, so it's important to stay up to date and continue learning throughout your career. This could involve taking online courses or earning advanced degrees, as well as staying engaged with the data science community through conferences and networking events. Another important factor in achieving professional success is to have a strong portfolio of work that demonstrates your skills and experience. This could include projects you've worked on in the past, as well as any published research or articles you've written. Having a strong online presence, through a personal website or a professional social media profile, can also help you showcase your work and reach a wider audience. Effective communication is also crucial for success in data science. As a communication specialist, you should be able to clearly and effectively communicate technical concepts to non-technical audiences. This could involve creating presentations or visualizations to help explain complex data or developing clear and concise reports that summarize your findings. Finally, networking and building relationships with others in the field can also be beneficial for your career. Networking can help you learn about job opportunities, get feedback on your work, and connect with other professionals who can help you advance your career. To achieve professional success and reach a high salary as a data science and communication specialist, it's important to continuously learn and improve your skills, build a strong portfolio of work, effectively communicate technical concepts to non-technical audiences, and network with others in the field.

★ *Where and what online materials we could read to learn more about Data Science*

There are many online resources available for learning about data science, including:

- **Books:** There are many books available on a wide range of **data science** topics, such as **programming**, **machine learning**, **statistics**, and **data visualization**. Some popular books for learning data science include “**Python for Data Science Handbook**” by Jake VanderPlas, “**Hands-On Machine Learning with Scikit-Learn, Keras, and**

TensorFlow” by Aurélien Géron, and **“Data Science from Scratch”** by Joel Grus.

- **Courses:** Online courses are a great way to learn about data science, as they often provide structured learning materials and exercises. Some popular platforms for data science courses include **Coursera**, **edX**, and **Udacity**.
- **Tutorials:** There are many online tutorials available for learning about **data science**, ranging from beginner-level to advanced. Some popular sources for data science tutorials include **DataCamp**, **Kaggle**, and **Dataquest**.
- **Websites:** Many websites provide articles, tutorials, and other resources for learning about **data science**. Some popular sites include **Data Science Central**, **Data Science Society**, and **Data Science 101**.
- **Conferences and meetups:** Attending conferences and meetups is a great way to stay up to date with the latest developments in **data science** and network with other professionals in the field. Some popular conferences include the **Conference on Neural Information Processing Systems (NeurIPS)** and the **Association for Computational Linguistics (ACL)** conference.

★ *What kind of jobs can I find in Communication and Data Science, where, and how?*

There are many ways in which **data science** and **communication** intersect, and there are a variety of jobs that involve both fields. Some common roles that combine **data science** and **communication** skills include:

- **Data journalist:** **Data journalists** use **data analysis** and **visualization techniques** to **tell stories** and **communicate complex information** to a general audience. They might work for news organizations, magazines, or other media outlets, and might use tools such as Python and Excel to **analyze data** and **create interactive visualizations**.
- **Content strategist:** **Content strategists** use **data analysis** and **customer insights** to inform the creation and distribution of content. They might work in **marketing** or **advertising**, and might be responsible for **creating data-driven content calendars**, **analyzing the**

performance of content, and identifying trends and patterns in customer behavior.

- **UX researcher:** UX researchers use **data analysis** and **user testing techniques** to inform the design of user experiences. They might work in fields such as **software development, e-commerce, or gaming**, and might be responsible for conducting **user interviews, analyzing user behavior, and creating user personas and journey maps.**
- **Data visualization specialist:** Data visualization specialists use **data analysis** and **visualization techniques** to communicate information **visually**. They might work in fields such as **business intelligence, data analytics, or consulting** and might be responsible for **creating data-driven dashboards, reports, and visualizations.**
- **Business analyst:** Business analysts use **data analysis and communication skills** to help organizations understand their operations and make **informed decisions**. They might work in a variety of industries, and might be responsible for conducting **market research, analyzing financial data, and creating presentations and reports.**
- **Technical writer:** Technical writers use **data analysis and communication skills** to create **technical documentation** and other materials that explain complex concepts to a general audience. They might work in fields such as **software development, engineering, or healthcare**, and might be responsible for **creating user manuals, help guides, and online documentation.**
- **Data product manager:** Data product managers use **data analysis and communication skills** to lead the **development of data-driven products**. They might work in fields such as **software development, e-commerce or consulting**, and might be responsible for defining the product vision, gathering customer insights, and managing the development process.
- **Research scientist:** Research scientists use **data analysis and communication skills** to **conduct research and communicate** findings to a variety of audiences. They might work in fields such as **academia, government, or industry**, and might be responsible for **designing research studies, analyzing data, and writing papers and reports.**

To find jobs that combine **data science and communication skills**, you can **search job** boards such as **LinkedIn, Indeed, and Glassdoor,**

or check out job listings at companies that work in the field. You can also consider **joining professional organizations** or **attending conferences** and meetups to network with other professionals and learn about job opportunities.

★ *The impact of Data Science and Communication on the future of human society*

Data Science and **communication** are both fields that are having a significant impact on the future of human society. Here are a few ways in which **data science** and **communication** are shaping the future:

- **Decision-making:** **Data Science** is helping organizations to make more **informed decisions** by providing **insights** and **predictions** based on **data**. For example, **Data Science** can be used to optimize supply chain management, predict customer behavior, or identify trends in the stock market.
- **Communication and information sharing:** **Communication technologies** are enabling people to connect and share information more easily than ever before. **Data Science** is also helping to improve the way information is shared and consumed, “through the use of **data visualization** and **natural language processing**” (Shen, L., & all, 2021: 1).
- **Personalization:** **Data Science** is being used to personalize products and services, such as **online recommendations**, **personalized advertising**, and **personalized healthcare**. This is made possible through the use of **machine learning algorithms** (Chattu, V. K., (2021), which can analyze large amounts of data to understand individual preferences and needs.
- **Automation:** **Data Science** or **Industry 4.0** is also being used to **automate tasks and processes**, which “has the potential to revolutionize many industries” (Sarker, I. H., 2022: 158). For example, **Data Science** can be used to develop autonomous vehicles, automate customer service, or optimize manufacturing processes.

Data Science and **communication** are shaping the future of human society in many ways and will likely continue to do so in the coming years.

Congratulations to all readers who have finished my book! The ability to effectively communicate and analyze data is an important

skill in today's world, and by investing the time and effort to learn these concepts, you have taken an important step toward success in your studies and career. Data science and communication are rapidly evolving fields, and by learning the foundations, you have set yourself up for success in learning and adapting to new technologies and techniques as they emerge. In addition to the technical skills you have learned, the ability to communicate your findings and insights effectively is crucial for collaborating with others and making an impact in your field. Whether you are working on a team project, presenting your research to colleagues or clients, or simply explaining your work to a non-technical audience, clear and effective communication is key. As you continue on your journey, don't forget to also focus on developing your skills in areas such as problem-solving, collaboration, and critical thinking. These skills, along with your technical expertise, will help you excel in your studies and career. Keep up the good work and best of luck in your future endeavors!

REFERENCES

- Agarwal, A., & Saxena, A. (2018). Malignant tumor detection using machine learning through scikit-learn. *International Journal of Pure and Applied Mathematics*, 119(15), 2863-2874.
- Agrawal, T. (2021). Hyperparameter optimization using scikit-learn. In *Hyperparameter Optimization in Machine Learning* (pp. 31-51). Apress, Berkeley, CA.
- Atwi, H., Lin, B., Tsantalos, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., ... & Lanza, M. (2021, September). PyRef: refactoring detection in Python projects. In 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 136-141). IEEE.
- Bergen, K. J., Johnson, P. A., de Hoop, M. V., & Beroza, G. C. (2019). Machine learning for data-driven discovery in solid Earth geoscience. *Science*, 363(6433), eaau0323.
- bin Uzayr, S. (2021). Working with Python Frameworks. In *Optimizing Visual Studio Code for Python Development* (pp. 123-166). Apress, Berkeley, CA.
- Brownlee, J. (2017). Introduction to time series forecasting with python: how to prepare data and develop models to predict the future. Machine Learning Mastery.
- Bruce, P., Bruce, A., & Gedeck, P. (2020). Practical statistics for data scientists: 50+ essential concepts using R and Python. O'Reilly Media.
- Bynum, M. L., Hackebeil, G. A., Hart, W. E., Laird, C. D., Nicholson, B. L., Siirola, J. D., ... & Woodruff, D. L. (2021). A Brief Python Tutorial. In *Pyomo—Optimization Modeling in Python* (pp. 203-216). Springer, Cham.
- Campeasato, O. (2022). *Python for Programmers*. Stylus Publishing, LLC.
- Charatan, Q., & Kans, A. (2022). Object-Oriented Python: Part 1. In *Programming in Two Semesters* (pp. 147-172). Springer, Cham.
- Chattu, V. K. (2021). A review of artificial intelligence, big data, and blockchain technology applications in medicine and global health. *Big Data and Cognitive Computing*, 5(3), 41.
- Gad, A. F. (2021). Pygad: An intuitive genetic algorithm python library. *arXiv preprint arXiv:2106.06158*.
- Goodrum, H., Roberts, K., & Bernstam, E. V. (2020). Automatic classification of scanned electronic health record documents. *International journal of medical informatics*, 144, 104302.
- Hajba, G. L. (2018). Using beautiful soup. In *Website Scraping with Python* (pp. 41-96). Apress, Berkeley, CA.
- Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357-362.

- Hunt, J. (2019). Dictionaries. In *A Beginners Guide to Python 3 Programming* (pp. 389-400). Springer, Cham.
- Johansson, R. (2015). Data Input and Output. In *Numerical Python* (pp. 425-451). Apress, Berkeley, CA.
- Kumar, A., & Jain, M. (2020). Tips and Best Practices. In *Ensemble Learning for AI Developers* (pp. 97-129). Apress, Berkeley, CA.
- Long, J. D., & Teetor, P. (2019). *R Cookbook: proven recipes for data analysis, statistics, and graphics*. O'Reilly Media.
- Lott, S. F. (2018). *Functional Python programming: Discover the power of functional programming, generator functions, lazy evaluation, the built-in itertools library, and monads*. Packt Publishing Ltd.
- Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages*. "O'Reilly Media, Inc".
- McKinney, W., & Team, P. D. (2015). *Pandas-Powerful python data analysis toolkit*. *Pandas—Powerful Python Data Analysis Toolkit*, 1625.
- Meurer, A., Smith, C. P., Paprocki, M., ertik, O., Kirpichev, S. B., Rocklin, M., ... & Scopatz, A. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3, e103.
- Mishra, S. K., Pradhan, M., & Pattnaik, R. A. (2021). Statistical Methods for Reproducible Data Analysis. In *Cognitive Computing Using Green Technologies: Modeling Techniques and Applications* (pp. 37-57). CRC Press.
- Navlani, A., Fandango, A., & Idris, I. (2021). *Python Data Analysis: Perform data collection, data processing, wrangling, visualization, and model building using Python*. Packt Publishing Ltd.
- Parvez, M. S., Tasneem, K. S. A., Rajendra, S. S., & Bodke, K. R. (2018, January). Analysis of different web data extraction techniques. In *2018 International Conference on Smart City and Emerging Technology (ICSCET)* (pp. 1-7). IEEE.
- Petrelli, M. (2021). *Introduction to Python in Earth Science Data Analysis: From Descriptive Statistics to Machine Learning*. Springer Nature.
- Qi, P., Zhang, Y., Zhang, Y., Bolton, J., & Manning, C. D. (2020). Stanza: A Python natural language processing toolkit for many human languages. *arXiv preprint arXiv:2003.07082*.
- Ranjan, G. S. K., Verma, A. K., & Radhika, S. (2019, March). K-nearest neighbors and grid search cv based real time fault monitoring system for industries. In *2019 IEEE 5th international conference for convergence in technology (I2CT)* (pp. 1-5). IEEE.
- Ranjani, J., Sheela, A., & Meena, K. P. (2019, April). Combination of NumPy, SciPy and Matplotlib/PyLab-a good alternative methodology to MATLAB-A Comparative analysis. In *2019 1st International Conference on Innovations in Information and Communication Technology (ICIICT)* (pp. 1-5). IEEE.
- Sarker, I. H. (2022). Ai-based modeling: Techniques, applications and research issues towards automation, intelligent and smart systems. *SN Computer Science*, 3(2), 1-20.

- Shen, L., Shen, E., Luo, Y., Yang, X., Hu, X., Zhang, X., ... & Wang, J. (2021). Towards natural language interfaces for data visualization: A survey. *arXiv preprint arXiv:2109.03506*.
- Sheppard, C. (2017). *Genetic algorithms with Python*. S. I: Smashwords Edition.
- Stepanek, H. (2020). The apply Method. In *Thinking in Pandas* (pp. 121-133). Apress, Berkeley, CA.
- Takefuji, Y. (2022). Set Operations in Python for Translational Medicine. *International Journal of Translational Medicine*, 2(2), 174-185.
- Thakuriah, P. V., Tilahun, N. Y., & Zellner, M. (2017). Big data and urban informatics: innovations and challenges to urban planning and knowledge discovery. In *Seeing cities through big data* (pp. 11-45). Springer, Cham.
- Van Rossum, G., & Drake Jr, F. L. (1995). *Python tutorial* (Vol. 620). Amsterdam, The Netherlands: Centrum voor Wiskunde en Informatica.
- VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. "O'Reilly Media, Inc."
- Waskom, M., Botvinnik, O., Ostblom, J., Lukauskas, S., Hobson, P., Gemperline, D. C., ... & Evans, C. (2020). *mwaskom/seaborn: v0. 10.0* (January 2020). Zenodo.
- Yim, A., Chung, C., & Yu, A. (2018). *Matplotlib for Python Developers: Effective techniques for data visualization with Python*. Packt Publishing Ltd.

Online References

- Buitinck, L., & all, (2013). API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, link accessed on 01.06.2022: <https://arxiv.org/abs/1309.0238>
- Jindal, A., & all, (2021, January). Magpie: Python at speed and scale using cloud backends. In *CIDR*, link accessed on 10.09.2022: <https://wentaowu.github.io/papers/cidr21-magpie.pdf>
- Lu, W. (2022). Verifying Python Programs in Robotic Applications, link accessed on 02.10.2022: https://ssvlab.github.io/lucascordeiro/supervisions/bsc_thesis_wenda.pdf
- Thakur, A. (2020). *Approaching (almost) any machine learning problem*. Abhishek Thakur, link accessed on 20.12.2022: https://books.google.ro/books?hl=ro&lr=&id=ZbgAEAAAQBAJ&oi=fnd&pg=PA14&dq=y_proba+is+a+numpy+array+it+the+model+to+training+data+before+making+predictions.+You+can+fit+the+model+using+the+fit+method+of+the+model+class,+as+shown+in+the+following+example:&ots=Q11BbkJWbU&sig=c3mLPGfFr-HXh1jlf1TCb9WFjcU&redir_esc=y#v=onepage&q&f=false
- Thorat, A. (2021). Applications of Artificial Intelligence in Cyber Security, link accessed on 01.01.2023, book available online at: https://www.researchgate.net/profile/Arvind-Thorat-2/publication/355218335_APPLICATIONS_OF_ARTIFICIAL_INTELLIGENCE_IN_CYBER_SECURITY/links/61b2dea4590a0b7ed6346f06/APPLICATIONS-OF-ARTIFICIAL-INTELLIGENCE-IN-CYBER-SECURITY.pdf

Verma, O. P. (2019). Packing and unpacking of arguments in Python, link accessed on 15.11.2022: <http://103.47.12.35/bitstream/handle/1/7285/1936.pdf?sequence=1&isAllowed=y>

Online coding sources

*** **sklearn.preprocessing.Normalizer**: link accessed on 28.12.2022: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html>

*** **sklearn.linear_model.LinearRegression**: link accessed on 02.01.2023: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

*** **sklearn.preprocessing.StandardScaler**: link accessed on 25.12.2022: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler>