Floyd Bax

PYTHON DATA ANALYTICS

First edition

This book was professionally typeset on Reedsy
Find out more at reedsy.com

Contents

1

Introduction

In today's discussions, data is a topic that dominates conversations. Chances are, you encounter the term "data" numerous times in a single day. Data, as a concept, is incredibly expansive. There exists a depth to data that may elude complete comprehension, at least within our lifetimes. However, one undeniable aspect of data is its capacity to narrate a story, whether it involves explaining an event or forecasting the future.

Data stands as the cornerstone of the future. Businesses, governments, organizations, and even malevolent actors all seek data for various purposes. Entities are investing in diverse data strategies to gain insights into their current circumstances and equip themselves for the unknown. The world of technology is advancing toward an open-source paradigm, facilitating the free exchange of ideas. This represents the initial stride toward dismantling monopolies and decentralizing innovative concepts. Consequently, the tools, techniques, and data utilized in analysis are readily accessible for anyone seeking to decipher datasets and derive meaningful interpretations.

Numerous tools are available for conducting data analysis, making the ultimate choice a daunting task for most individuals. To set yourself on the right trajectory, the initial step involves selecting the programming language you wish to acquire and then building upon that foundation. Novice programmers often grapple with this decision, as elucidated in

prior volumes of this series. Nevertheless, as a proficient data analyst, your path is likely well-established. Nonetheless, there is no harm in embracing novelty, as the world of technology often reveals unexpected utility.

For the majority, Python has supplanted older languages like C as the primary language of choice. Python's surging popularity can be attributed to its user-friendly nature, simplicity, and its classification as a high-level programming language. Being high-level means it closely resembles human languages. Your familiarity with Python's syntax and functions over the years likely underscores your appreciation for this aspect. Furthermore, a burgeoning community of developers, data scientists, and experts continually collaborates to enhance Python and provide mutual support.

Python finds extensive application across various domains, with a particular emphasis on data analysis. Data scientists have increasingly embraced Python due to its effectiveness in exploring and comprehending extensive datasets. Consequently, experts have developed specialized libraries tailored to data manipulation and analysis within Python. These libraries offer an array of powerful tools for data processing and analysis. Such is the growth of data science that tech giants like Microsoft and Google are heavily invested in supporting open-source projects and initiatives in this domain.

A pivotal concept in data analysis, particularly in Python, is simplicity. Python stands out among programming languages for its simplicity, ensuring clarity in code definitions. Other developers who encounter your work should not struggle to decipher it, simplifying its integration into

their projects. Your code should be easily comprehensible to anyone perusing it.

Thanks to Python's simplicity and streamlined code flow, the focus typically shifts towards efficient memory utilization rather than script performance. This further streamlines the data analysis process. Utilizing Python for data analysis necessitates access to an array of tools specifically designed for scientific, numerical, and visual computations and representations, as these constitute the crux of data comprehension.

As an adept data analyst, your mastery of Python libraries proves invaluable on various occasions. NumPy, for instance, facilitates tasks involving linear algebra, vectors, random variables, and matrices. Matplotlib enables diverse data visualization methods, enhancing the data's accessibility and understanding. Pandas offers reliable, fast, and easily comprehensible data structures crucial for data manipulation and computations.

To simplify your work, IPython notebooks within the Anaconda environment provide an exceptional platform for Python code execution without the need for extensive manual coding. The notebooks incorporate Python code within visual elements, allowing for instant visualization of results. These tools collectively empower your journey in data analysis.

Data analysis and data science are inherently evolutionary fields where each new skill acquired contributes to something greater. Beginning with the fundamentals of data analysis, you can progress into machine learning. In fact, Python-based data analysis serves as the foundation for venturing into machine learning. Proficiency in logistic and linear regression and

familiarity with the Scikit-learn library in Python represent initial steps toward advancing into machine learning and predictive science.

One essential lesson gleaned from experience in utilizing Python for data analysis is that analytics rarely exists in isolation. Consequently, you must acquire proficiency in other programming languages. The advantage is that Python knowledge is transferable to numerous programming environments.

Python remains the optimal choice for anyone captivated by the world of data. Whether it involves data retrieval, web scraping, data processing, or data analysis, Python simplifies these tasks. It stands as an accessible language equipped with a plethora of tools, offering limitless possibilities for data exploration and utilization.

Conceptual Approach to Data Analysis

Data surrounds us constantly, and we engage with it throughout our daily routines. Every individual and organization leaves traces of data on their frequently used devices. Both parties rely on this data for making informed decisions. So, how do they transform raw data into valuable insights that can guide credible business choices? This is where the process of data analysis comes into play.

Data analysis is a comprehensive procedure wherein analysts employ statistical and analytical tools to draw meaningful deductions from a given dataset. Various analytical techniques are at the disposal of data analysts for this purpose, including data visualization, business intelligence, and data mining.

Methods Employed in Data Analysis

As mentioned earlier and in previous works within this series, data analysis is a multifaceted process. The following offers an overview of some of the techniques one encounters in data analysis:

Visualization of Data

Data visualization is primarily concerned with presentation. You are likely familiar with many tools used in data visualization, such as pivot tables, pie charts, and other statistical instruments. Beyond enhancing the presentation, data visualization simplifies the comprehension of extensive datasets. Rather than deciphering tables, for instance, you can quickly grasp the information when it's presented as a color-coded pie chart.

Humans are inherently visual beings, and visual representations tend to linger longer in our memories compared to textual information. At a glance, one can discern the essence of the information. Summarizing data through visualization is swifter and more accessible than sifting through raw data. An inherent strength of data visualization is its ability to expedite the decision-making process.

Business Intelligence

Business intelligence encompasses the process of converting data into actionable information aligned with the strategic objectives of end users. While raw data may often appear daunting, business intelligence molds it into a coherent narrative. Techniques within business intelligence aid in identifying trends, analyzing them, and extracting valuable insights.

Many companies employ these techniques to inform decisions related to pricing strategies, product placements, and the exploration of new markets.

Such data also contributes to assessing the sustainability of these markets. Ultimately, this information enables companies to devise specific strategies for thriving within each market segment.

Data Mining

Data mining involves the scrutiny of extensive datasets to identify recurring patterns. These patterns enable analysts to recognize trends and base decisions on their findings. Data mining methods include machine learning, artificial intelligence, database utilization, and statistical computations.

The outcome of data mining is the transformation of raw data into reliable information suitable for informed business decisions. Beyond decision-making, data mining is valuable for uncovering dependencies or anomalies across different datasets. It is also instrumental in cluster analysis, where analysts study specific data sets to identify distinct data groups.

Data mining can be combined with machine learning to gain insights into consumer behavior, which is inherently dynamic, particularly in the context of the ever-evolving e-commerce landscape. Through data mining, analysts collect extensive information about consumer actions on websites, facilitating the accurate or nearly accurate prediction of purchase behaviors and frequencies. Such insights prove invaluable to marketing departments and allied sectors, aiding them in creating targeted promotional content to attract and retain customers.

Similar to how marketing experts often create niches within broader market demographics, data mining can identify previously unidentified data groups. Analyzing such data groups is crucial, as it allows analysts to

experiment with undefined stimuli, potentially uncovering new opportunities for marketing strategies.

In addition to previously unidentified data, data mining is also effective when dealing with well-defined datasets. This often involves elements of machine learning, exemplified by modern email systems. Each email provider employs systems that classify messages as spam or non-spam, effectively filtering them into the appropriate inboxes.

Text Analysis

Text analysis, frequently considered a subset of other data analysis methods, involves the examination of text messages to extract useful information from their content. Beyond reading the text, the information undergoes processing using specific algorithms to support decision-making.

The nature and process of text analysis vary based on organizational needs. Information is extracted from diverse databases or file systems and subjected to linguistic analysis. This approach facilitates the identification of patterns by examining keyword frequencies. Pattern recognition algorithms target specific elements, such as email addresses, street names, geographic locations, or phone numbers.

Text analysis finds widespread application in marketing, as companies analyze competitors' websites to gain insights into their business operations. This involves searching for specific target words that shed light on why a competitor may outperform or underperform. Such analysis can yield competitor keywords and phrases, aiding analysts in devising counter-strategies for their own companies.

Data Analysis Procedure

While the data analysis methods discussed above may differ in their approaches, their ultimate goal remains largely consistent: supporting decision-making within organizations at various levels. The following steps outline the data analysis process:

1. Define the Objectives

Clearly delineate the objectives of your study, as they form the bedrock of your analysis. The subsequent steps depend on the clarity of these objectives, guiding your data collection efforts and determining the purpose of the gathered data.

2. Pose the Right Questions

To fulfill the previously outlined objectives, seek answers to specific questions. This focused approach ensures that your analysis centers on relevant matters, preventing the collection of extraneous data. An efficient data collection process is vital to avoid amassing irrelevant information.

3. Collect Data

Establish suitable data collection points, selecting appropriate statistical methods or data collection techniques. Data can take various forms, particularly when dealing with raw data. Once obtained, the data refinement process commences, eliminating inaccuracies or irrelevant entries. Employ appropriate tools for importing and analyzing the data.

4. Analyze Data

Aggregate and cleanse the data using various tools. This stage enables the study of data to identify patterns and trends, providing answers to the questions posed earlier. “What if” analyses are often conducted at this juncture.

5. Interpretation and Predictive Analysis

With the essential insights garnered from your analysis, the final step involves drawing conclusions from the data. Predictive analysis entails making informed decisions based on the analyzed data and integrating it with other supporting information. Quantitative data is not the sole consideration; qualitative elements also play a role. For instance, you may possess the requisite numbers, but if market sentiment toward your business is negative, predictions should encompass this qualitative dimension.

This stage also prompts a review of the initial objectives. Does the collected data adequately address the posed questions? Are there potential objections that the data can convincingly counter? Have any intentional omissions or limitations affected the conclusions? And how does the introduction of external factors impact the outcomes?

Approaches Utilized in Data Analysis

In today’s world, access to an abundance of data is commonplace. What truly matters is the manner in which you employ your data. Data analysts routinely grapple with copious amounts of data, the challenge lying in discerning the significance within. To tackle this task, various tools and techniques, primarily in statistical data analysis, come into play.

In a landscape where big data has reached its zenith, numerous tools can alleviate your workload while concurrently enhancing the efficiency and

reliability of your data. The methods expounded upon here constitute the bedrock of data analysis. Proficiency in these techniques serves as a stepping stone toward more advanced methods and strategies:

1. Standard Deviation

Standard deviation quantifies the extent to which data deviates from the arithmetic mean. In data analysis, it signifies the dispersion of data points from the mean. A high standard deviation denotes a wide divergence from the mean, while a low value indicates that the majority of the data closely aligns with the mean.

It is essential to employ standard deviation in conjunction with other techniques to derive conclusive findings from your study, particularly when dealing with datasets containing numerous outliers, as it may not be a reliable determinant on its own.

2. Averages

Averages, often referred to as arithmetic means, are calculated by dividing the sum of (n) items in a list by the total number of (n) items. Averages offer insights into the general trends within a specific dataset. Computing averages is straightforward, and from this information, one can glean valuable insights about the dataset at a glance.

While using averages, caution is necessary to avoid relying on them in isolation. Independent of other methods, averages can sometimes be misconstrued and may not provide accurate information, especially when dealing with data exhibiting skewed distributions.

3. Regression Analysis

Regression analysis revolves around identifying relationships among various variables. From these relationships, the dependency between the variables is established. This analysis aids in determining whether the relationships between variables are weak or strong.

Regression analysis proves particularly valuable when forecasting decision-making processes. It allows for the examination of numerous variables that impact a business in various ways. The dependent variable in your study represents the variable of interest, while the independent variables can be diverse, representing factors under scrutiny that may influence the dependent variable.

4. Hypothesis Testing

Also known as t testing, hypothesis testing aims to validate or refute a specific assertion within your study population. This method is widely used in fields reliant on data, such as economics, scientific research, and business analysis.

Awareness of potential errors is crucial for successful hypothesis testing. The Hawthorne effect, or observer effect, is a common pitfall wherein study results are skewed due to participants being aware of observation, rendering the outcomes unreliable.

Hypothesis testing aids in decision-making by comparing data against hypothetical scenarios related to your operations, revealing correlations between variables.

5. Determining Sample Sizes

Selecting an appropriate sample size is imperative for effective study conduct, as it is impractical to collect data from an entire study population. Careful sample size selection ensures accurate and unbiased results.

One challenge in sample size determination is achieving accuracy. Although you won't study the entire population of interest, your sample must be randomly selected to yield precise results.

Types of Data Analysis

Before delving into data analytics, it is essential to grasp fundamental concepts that you will frequently encounter. The terminology employed in data analytics varies depending on the type of analysis. The sheer volume and unpredictability of data from diverse sources necessitate data refinement into an understandable and actionable format. Here are some types of data analysis you will encounter:

1. Descriptive Analysis

Descriptive analysis focuses on summarizing data. It aims to provide concise answers to critical questions about organizations, events, or activities. Tools employed in descriptive analysis encompass generated narratives, pie charts, bar charts, and line graphs, offering a quick overview of presented information.

2. Diagnostic Analysis

Diagnostic analysis resembles consulting a doctor for a health diagnosis. While you may experience symptoms, the goal is to use data to elucidate the unknown. In marketing campaigns on social media, for instance, diagnostic

analysis delves deeper into engagement metrics, revealing the meaning behind numbers.

3. Predictive Analysis

Predictive analysis, a common approach in organizations, leverages statistical algorithms and machine learning to extrapolate future possibilities from historical data. Accuracy in predictions hinges on the quality of historical data, as flawed data can lead to erroneous forecasts.

Predictive analysis centers on future planning, employing present and past data to anticipate outcomes when altering controllable variables and creating predictive models.

4. Exploratory Analysis

Exploratory analysis seeks to uncover data trends and elucidate features that may elude detection through other analytical methods. It emphasizes identifying outliers, understanding their occurrence, and comprehending their impact on decision-making.

5. Prescriptive Analysis

Unlike general analyses, prescriptive analysis provides precise, specific answers. It operates akin to prescribing medication, where a doctor recommends particular drugs to be taken under specific instructions. In data analysis, it narrows down factors contributing to specific outcomes, such as road accidents caused by drunk driving, inadequate road signage, vehicle roadworthiness, or reckless driving.

Tools Utilized in Data Analysis

To excel in a data analyst career, familiarity with various tools is essential. At a foundational level, you should possess proficiency in web development, SQL, mathematics, and Microsoft Excel. Additionally, competence in PHP, HTML, JavaScript, and basic programming commands, libraries, and syntax is beneficial.

For advanced users, expertise in the following areas is advantageous:

1. R Programming

Data analysts often grapple with the choice of programming languages. Learning multiple languages is prudent, as project demands can vary. While you may not master all languages, having working knowledge is valuable.

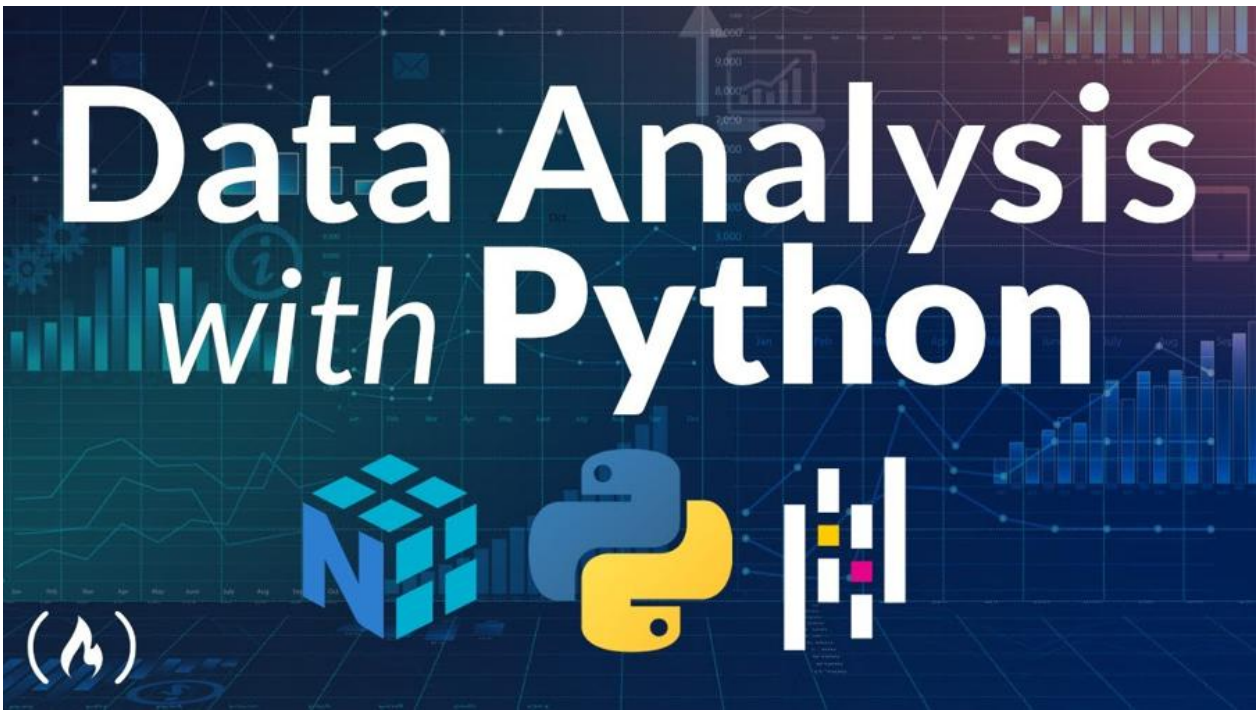
Among programming languages, R programming is crucial for any data analyst. It stands out for its versatility, especially in handling statistical data. As an open-source platform, R offers access to a community of data analysts for assistance. R boasts recursive functions, loops, conditionals, I/O support, storage capabilities, and an effective GUI for data display.

2. Python

Python, an open-source programming language, is renowned for its simplicity and robustness. It enjoys widespread use among programmers and developers due to its extensive library support for data management, manipulation, and analysis.

Python’s ease of learning, especially for those with experience in other languages, makes it a preferred choice. Countless projects can be undertaken

in Python, with its potential continually expanding through new projects, such as YouTube, which was developed using Python.



Database Management

Working with substantial volumes of data requires proficiency in data management. You must hone your skills in this area. Mastery of essential tools such as MySQL, MongoDB, MS Access, and SQL Server is imperative for data collection, processing, and storage. Equally important is the comprehension of commands like order by, having, group by, where, from, and select.

MatLab

MatLab stands as another versatile, potent, and user-friendly programming language crucial for data analysis. With MatLab, you can manipulate and scrutinize data using its built-in libraries. Familiarity with programming

languages like C++ and C can expedite your progress in MatLab, given their syntactical similarities.

Over time, data analysis has gained prominence across diverse settings. Companies and organizations employ data analysis to gain insights into their business performance by examining customer interactions with their brands at various data collection points. Now that we've covered the basics of data analysis, let's delve into data analysis using one of the most remarkable programming languages: Python.

Advantages of Data Analysis in Python Compared to Excel

By now, you may have experimented with various tools and applications for data analysis. Many analysts begin with Excel before transitioning to Python and other languages. In the business realm, Microsoft Excel is a pivotal program, particularly for data collection. While Excel can be used for data analysis, it presents challenges that may necessitate a shift to Python for data analysis.

While Excel is a valuable tool, it poses unique challenges that Python can overcome. Learning some Python programming can significantly enhance your data analysis capabilities in the field of data science.

Expert Data Handling

Python sets itself apart from Excel and other basic data analysis tools with its extensive data handling capabilities. This encompasses everything from data importation to manipulation. In Python, you can effortlessly import various data file formats, a capability Excel lacks. Excel often struggles with certain data formats, hindering your work. Python grants you greater control

over data handling, enabling you to scrape data from diverse databases, analyze it, and draw conclusions.

While Excel can perform numerous data tasks, it may impose limitations. These restrictions do not apply in Python, where you can perform a wide range of data manipulations, including recording, merging, and cleaning. Python libraries like Pandas facilitate data viewing and cleaning, ensuring suitability for your analysis purpose. Achieving the same tasks in Excel would require excessive time and might not yield optimal results. Therefore, Python not only offers enhanced utility but also optimizes time efficiency.

Automated Data Management

Excel excels in data management with its user-friendly graphical user interface (GUI). Nevertheless, when it comes to automating processes, Excel falls short. Automating tasks or conducting analytical processes across multiple Excel sheets often proves cumbersome. In contrast, Python simplifies automation. To analyze recurring data, you only need to write a script that imports new data, parses it, and generates analytical reports promptly. In Excel, you would need to create new files manually, input desired formulas and functions, and then proceed with the analysis.

Furthermore, Python allows you to save output files in various database file formats, eliminating the need for time-consuming file conversions, a common hindrance when working with Excel.

Economies of Scale

Excel organizes data into tabs and sheets, a feature that works well for tasks centered around Excel. However, this structure becomes problematic when dealing with extensive databases. Processing such files in Excel can be

time-consuming and may lead to system crashes. Python, on the other hand, is specifically designed to handle such challenges. It processes large files faster and more efficiently than Excel, reducing the risk of system crashes.

Data Regeneration Capability

As a data analyst, you must explain your work to various stakeholders. After completing an analysis, you may need to prepare a report for another department's use or present your findings to a panel. To meet these objectives, your data must be reproducible. Excel poses challenges in this regard since it is challenging to provide a comprehensive illustration of the analysis process. Replicating your work can be cumbersome, especially if you completed it hastily.

Python simplifies the sharing of your work. In some cases, you can execute the analysis with a single button press, enabling others to replicate your results easily. You can also explain each step, allowing your audience to follow along, execute code, and observe immediate results.

Debugging

Identifying errors in Excel can be challenging, as you must manually search for them within extensive datasets. In contrast, Python simplifies debugging. It promptly notifies you of syntax errors, offering a chance to rectify them. Python's code comments further assist in tracing and resolving issues. While some errors may still require time to identify and resolve, Python provides a more efficient debugging process than Excel.

Open-Source Programming

Excel relies on Microsoft for updates and bug fixes, limiting your control over the software's features and performance. Python offers the advantage of open-source programming, granting access to a vast community of programmers willing to assist with any concerns. You can modify code and share improvements with the Python community, resulting in enhanced functionality and visualizations.

Advanced Operation Support

Excel falls short in supporting machine learning and related features, requiring advanced programming languages like Python to fill the gap. Python allows you to create unique machine learning models and seamlessly integrate them into your code using popular frameworks like TensorFlow and Scikit-Learn.

Data Visualization

Effective data analysis often requires visualization. While Excel offers basic visualization features, Python provides a broader range of visualization options, especially for advanced visualizations. In business presentations, compelling visuals can significantly enhance your ability to convey information to non-technical audiences. Python's visualization capabilities can make your presentations more engaging and comprehensible.

In summary, while Excel remains a valuable tool, Python offers numerous advantages for data analysis, including enhanced data handling, automation, scalability, data regeneration, debugging, open-source support, advanced operations, and data visualization. As you progress in your data analysis career, consider expanding your skills into Python programming to conduct more accurate and complex analyses without the limitations of Excel.

Potential Drawbacks of Analyzing Data in Python

It’s widely acknowledged that Python is a favored choice for programming and data analysis among many individuals. Python stands out as one of the most accessible programming languages to learn, thanks to its concise code structure, which alleviates the need for extensive lines of code, making it particularly beloved by programmers. An exceptional feature of Python programming is its highly readable syntax, simplifying programming compared to languages like C. Moreover, Python boasts a dynamic library that allows programmers to execute various tasks, such as managing system interfaces and handling string operations, without the need for excessive code writing.

The mention of data analysis often sparks enthusiasm, as it plays a pivotal role in today’s business landscape. Virtually all companies, in some capacity, require access to your data to comprehend your behavior and enhance their service delivery. Beyond profit motives, businesses aim to ensure your satisfaction by deciphering the factors influencing your purchase decisions and tailoring their offerings to meet your needs effectively.

However, amidst the excitement surrounding data analysis, it may come as a surprise that this practice comes with unique challenges that hinder expected outcomes. One significant challenge data analysts encounter is the reliance on user-level data, which introduces room for errors, ultimately affecting data credibility and derived reports. Whether in marketing or other data-dependent business functions, the unpredictability of user-level data necessitates a careful balance between using, discarding, or persistently updating the data.

While Python offers numerous advantages, it’s crucial to acknowledge the potential challenges and limitations one might face while working with it.

Being aware of these issues allows you to prepare adequately. Below, we'll delve into some of the challenges that data analysts encounter when working with Python for data analysis.

1. Input Bias:

Data analysts often grapple with concerns regarding the reliability of the data they access, especially data collected at various touchpoints like online ads. Unfortunately, this input data doesn't always provide an accurate representation of customer interactions with a brand. Even methods like tracking cookies, while informative, raise questions about data accuracy due to the fragmented nature of user interactions across multiple devices.

2. Speed:

Python is widely known for its slower execution speed compared to many other programming languages, such as C++. When dealing with time-sensitive data analysis, speed becomes a critical factor. Recognizing potential speed challenges in Python programming is vital for planning and setting realistic project goals.

3. Version Compatibility:

Python's version compatibility can be a seemingly mundane but significant challenge. Choosing the right Python version, especially for novice data analysts, can be daunting. Python 2 and Python 3 are common choices, with some libraries and frameworks supporting one version over the other. Navigating version compatibility issues can be tricky when executing code and computations.

4. Porting Applications:

Python, like other high-level programming languages, relies on interpreters to translate code into instructions for the operating system. Installing the correct interpreter version for different platforms can be problematic, and the porting process often encounters obstacles.

5. Lack of Independence:

Python depends on third-party libraries, packages, and frameworks to facilitate effective data analysis. Unlike some programming languages that come with bundled features, Python requires the inclusion of additional libraries and frameworks. This reliance on open-source dependencies can increase project costs and resource consumption.

6. Algorithm-Based Analysis:

Data analysts typically employ two methods for data interpretation: sample-based analysis and algorithm-based analysis. While both methods offer actionable insights, they may not provide the contextual understanding needed to answer questions about customer behavior comprehensively.

7. Runtime Errors:

Python’s flexibility allows for coding without stringent variable definitions, which can lead to runtime errors. Identifying and rectifying these errors during code compilation can be time-consuming and resource-intensive.

8. Outlier Risks:

Data analysts often encounter outliers, which can cast doubt on data credibility, particularly when dealing with raw user data. Handling outliers effectively and determining their significance can be challenging, impacting the validity of analysis results.

9. Data Transfer Restrictions:

Stringent data protection laws, such as GDPR, require organizations to safeguard data. Consequently, data sharing among analysts and peers can be restricted, limiting collaborative efforts and hindering second opinions on data analysis procedures and assumptions.

In summary, while Python offers numerous advantages for data analysis, it’s essential to be mindful of these potential challenges and limitations to ensure a more informed and prepared approach to data-driven projects.

Data Analysis in Python

Why should you opt for Python when it comes to data analysis? Many individuals rely on Microsoft Excel as their primary analytical tool, but Python offers a more extensive array of capabilities. Python, introduced in 1991, has rapidly become one of the most widely used programming languages globally, thanks to its ease of learning.

Python boasts robust library support, with Pandas being a standout among these libraries. Python doesn't just provide a means of data analysis; it excels in data manipulation and cleaning, surpassing the capabilities of Excel. For applications reliant on data, Python stands out as the optimal choice, particularly given its versatility as a multipurpose language.

Python Libraries for Data Analysis

Before diving in, you must select the ideal development environment to work in. Most individuals choose from the following three options:

iPython notebook
IDLE
Terminal

While the choice of development environment largely hinges on personal preference, many developers favor iPython due to its exceptional built-in features that simplify work. With iPython, you can execute your code in blocks, eliminating the need to run each line individually during testing. Before delving deep into data analysis, let's revisit some crucial Python concepts. Most of these have been thoroughly covered in previous books in this series, serving as a refresher.

In Python, lists are enclosed in square brackets, with each item separated by a comma. For instance, here's a list of square numbers:

```
squares_list = [0, 1, 4, 9, 16, 25, 36]
```

Strings in Python are always defined within inverted commas. Here's an example of a string:

```
text = "'Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."'
print(text)
```

Lists and strings play a pivotal role in Python, particularly in data analysis. Assume you're tasked with performing mathematical operations or creating graphs from a given dataset in Python. You'd typically need to write code

explicitly for each task, which can be challenging for many individuals and may lead to waning enthusiasm for Python.

Instead of this arduous process, Python offers unique libraries with predefined instructions and functions that you can import into your development environment to address your tasks efficiently. Python libraries are invaluable in this regard.

In previous books in this series, you've been introduced to the fundamentals of Python programming. Now, let's shift our focus to the Python libraries used in data analysis. To expedite your learning, we'll reinforce key concepts learned in earlier books when necessary.

Among the myriad reasons why data scientists prefer Python over most programming languages is its user-friendly nature and open-source availability. Python is also a high-performance language, greatly simplifying development for object-oriented projects. However, the standout feature that makes Python immensely popular is its extensive library ecosystem. Each library is unique, yet comprehensive, enabling programmers to tackle a wide range of data-related challenges daily.

Here are some of the top libraries commonly employed in data science:

NumPy

For numerical computations, NumPy (Numerical Python) is indispensable. Considered the bedrock of numerical computations in Python, NumPy is a versatile array processor that leverages N-dimensional array objects. It excels in efficiency, particularly when working with multidimensional arrays, eliminating the sluggishness often associated with numerical computations. NumPy functions are precompiled, allowing for faster completion of

numerical routines compared to other libraries. NumPy's approach facilitates swift and efficient computations, especially when working with vectors. It serves as the foundation for libraries like Scikit-learn and SciPy and can also replace MATLAB when working with Matplotlib and SciPy.

TensorFlow

For high-performance computational projects, TensorFlow reigns supreme. With a vast community of contributors, TensorFlow serves as a valuable resource when grappling with challenges. Data scientists can define and execute computations with tensors, which are computational objects that yield values. TensorFlow offers high-quality graphical visualizations, enhancing project presentations. In the realm of neural machine learning, TensorFlow is a preferred choice, reducing computational errors by up to 60% and facilitating parallel computing. Google's support further bolsters TensorFlow, streamlining library management and ensuring access to the latest features through frequent updates. TensorFlow proves invaluable for projects involving video detection, time series analysis, text applications, and image or speech recognition.

Matplotlib

Matplotlib is the go-to library for data visualizations in data science, delivering exceptional results. It stands as the premier plotting library in Python, offering a wide array of plots and graphs. Matplotlib also features an object-oriented API for seamlessly integrating visualizations into various applications. If you've worked with MATLAB in the past, Matplotlib serves as an excellent open-source alternative. It comes with a wealth of expertise from a thriving community, providing ample support. Matplotlib is platform-agnostic, accommodating various output types and backends, enabling you to create visualizations in your preferred format. Its efficient memory utilization ensures a smooth runtime experience. Matplotlib aids in analyzing

correlations between variables, presenting each variable uniquely to facilitate the identification of similarities and differences. It's also instrumental in detecting outliers in scatter plots and highlighting data distribution nuances, offering deeper insights into the data under examination.

Pandas

Python Data Analysis (Pandas) is another indispensable library in data science, working hand-in-hand with Matplotlib and NumPy, especially for data cleaning. Pandas offers flexible and efficient data structures, simplifying structured data programming. When it comes to data cleaning or wrangling, Pandas excels, particularly with CSV files. It provides exceptional support for CSV files, allowing you to access data frames and execute transformations like extraction, transformation, and loading on datasets. Pandas boasts an elaborate syntax with powerful functions, enabling you to achieve remarkable results even with partially incomplete datasets. Data scientists across commercial, financial, and academic fields find Pandas invaluable, particularly in statistical data analysis. It's also a valuable tool for financial computations and has recently made inroads into neuroscience.

SciPy

For advanced computations in data science, Scientific Python (SciPy) is indispensable. This open-source library boasts a large contributor community. An extension of NumPy, SciPy offers the same efficiency for technical and scientific computations. Its functions and algorithms, an extension of NumPy, make scientific calculations more user-friendly. SciPy proves particularly advantageous when tackling differential problems, thanks to its integrated functions, further enhanced by the ndimage submodule for expedited multidimensional image processing. Speed is another hallmark of SciPy, rendering it a dependable library for data visualization and manipulation. Data scientists often turn to SciPy when dealing with linear algebra,

optimization algorithms, Fourier transforms, differential equations, and operations involving multidimensional images.

These libraries constitute the essential toolkit for data analysis. Most operating systems come pre-installed with Python, but it's advisable to verify you have the correct version before proceeding.

Windows Installation Guide

If you're operating on a Windows system, the installation process should be straightforward. Windows provides an installer package for you to download, and an installation wizard will guide you through the process until the final step. In this example, we'll walk you through the installation of NumPy, but the same procedure applies to other Python libraries as well.

Step 1:

Begin by going online and downloading the Windows installer package that matches your system setup. Here are some links to libraries:

- NumPy: https://pypi.org/project/numpy/
- SciPy: https://scipy.org/scipylib/
- Matplotlib: https://matplotlib.org
- IPython: https://ipython.org

Once you have the installer package (installer.exe) on your device, double-click it and follow the prompts of the installer wizard. If you already have Python installed, the wizard will detect it and provide guidance accordingly.

Linux Installation Guide

The method for installing on Linux depends on the specific Linux distribution you are using. Most Linux distributions come with NumPy preinstalled.

Installing IPython

This installation assumes you are using version 6.0 or higher. For a quick installation, enter the following command if you already have pip installed:

```
$ pip install ipython
```

This command will not only install IPython but also any necessary dependencies for future use. The most convenient way to install IPython along with most dependencies is to use pip. To ensure that you get the correct packages, consider using conda or pip.

While it is possible to install IPython by itself without additional dependencies, it is not recommended due to the lengthy process and potential issues that could impact your productivity. For a streamlined installation, follow these steps:

```
$ pip install ipython
```

Before installing any Python packages, always verify that you have the correct Python version and are running it from the command line. You can check the available version by running the following command:

```
python —version
```

The output should resemble this:

```
Python 3.6.0
```

If you do not have Python installed, you can find the latest version at www.python.org.

Building Python Libraries from Source

Installing Python libraries from source is an option, especially if you need to use the latest available version. While this process is generally straightforward, you may encounter some challenges specific to your operating system.

The following steps explain how to obtain NumPy from GitHub, and you can apply similar commands for other libraries:

```
$ git clone git://github.com/numpy/numpy.git numpy
```

To unpack the installer package, use the following command:

```
$ tar -xzf ipython.tar.gz
```

To install the library to your preferred destination, execute these commands:

```
$ python setup.py build
$ sudo python setup.py install —prefix=/usr/local
```

If you are using pip, you can install the libraries with these commands:

```
$ pip install numpy
$ pip install scipy
$ pip install matplotlib
$ pip install ipython
```

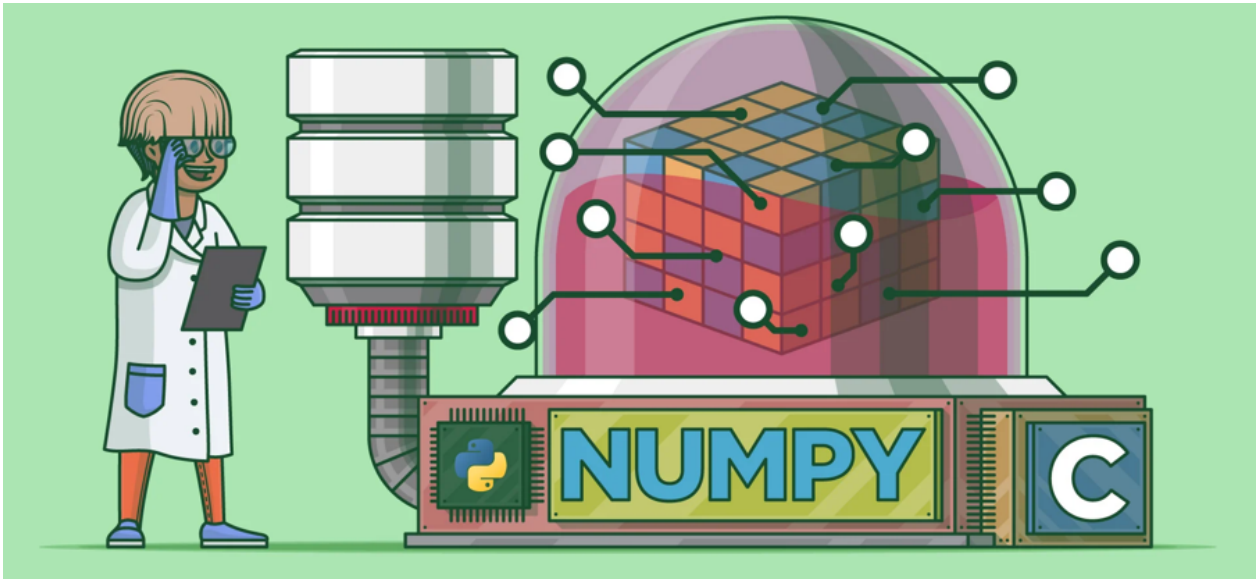
For setuptools users, the installation commands are as follows:

```
$ easy_install numpy
$ easy_install scipy
$ easy_install matplotlib
$ easy_install ipython
```

If you lack administrator rights on your device, prepend “sudo” to the commands mentioned above to install the libraries with super-user privileges.

4

Statistics in Python - NumPy



In the realm of Python programming, you'll encounter a plethora of libraries, and one such essential library is NumPy. Your comprehension of NumPy carries significant weight in the realm of scientific computation, particularly in the context of data analysis. Proficiency in this library marks a fundamental milestone in mastering the art of data analysis. Once you've grasped the intricacies of NumPy, you can seamlessly delve into other libraries like Pandas.

After mastering the fundamentals of NumPy, you can progress into the realm of data analytics, leveraging concepts from linear algebra and various statistical methodologies to dissect and scrutinize data. These two mathematical facets stand as pivotal knowledge areas for any data analyst. Throughout the data analysis journey, you'll frequently find yourself tasked with making predictions based on raw data at your disposal, such as calculating standard deviations or arithmetic means.

In the domain of linear algebra, the focus shifts towards employing linear equations for problem-solving, facilitated by NumPy and SciPy. A firm grasp of NumPy's fundamentals equips you to build upon your accrued knowledge and tackle intricate operations within the Python environment.

Within the NumPy realm, it's crucial to bear in mind the significance of file Input/Output (I/O). All the data you manipulate is sourced from files, underscoring the importance of acquiring basic proficiency in reading and writing data to and from these files. In the example below, we'll generate an identity matrix and save its contents to a file.

One of the distinctive advantages of harnessing the NumPy library lies in its inherent property where all elements within an array share the same data type. This property simplifies the determination of storage requirements for the array.

Your Python distribution typically includes NumPy as a core component. Nevertheless, if it's not already installed, you can install it using the following commands:

For Linux systems:

```bash
sudo apt-get install python-numpy
```

For Windows systems, ensure Anaconda is up and running, then execute the following command:

```bash
conda install numpy
```

Once NumPy is successfully installed, you can import the NumPy package into a new Python session as demonstrated below:

```python
>>> import numpy as np
```

As you delve deeper into NumPy, you'll soon realize that a substantial portion of your work revolves around N-dimensional arrays, commonly known as ndarrays. An ndarray represents a multidimensional array capable of accommodating a defined number of elements. Furthermore, ndarrays exhibit homogeneity, signifying that all elements within an array share the same size and data type.

Each object within an array is uniquely characterized by its data type (dtype). Consequently, each ndarray is intrinsically linked with a specific dtype. Arrays house a specific number of elements, and these elements are distributed across various dimensions. These dimensions and the elements within them collectively determine the array's shape, which is often referred to as its axes, and as these axes accumulate, they form a rank.

When initiating a new array, you can employ the array() function to incorporate all the elements from a Python list, as exemplified below:

```python
>>> x = np.array([5, 7, 9])
>>> x
array([5, 7, 9])
```

To verify that the object you've created is indeed an ndarray, you can utilize the type() function:

```python
>>> type(x)


```

The dtype associated with the ndarray can also be identified using the following function:

```python
>>> x.dtype
dtype('int32')
```

The aforementioned array possesses a single axis, resulting in a rank of 1. Its shape is denoted as (3,1). To derive these values from the array, you can employ attributes such as ndim (for the number of axes), size (for the array's length), and shape (for the array's shape), as illustrated below:

```python
>>> x.ndim
1
>>> x.size
3
>>> x.shape
(3L,)
```

In the preceding examples, we exclusively worked with one-dimensional arrays. As you progress in your data analysis journey, you'll encounter arrays with multiple dimensions. To elucidate this further, consider an example with two dimensions:

```python
>>> y = np.array([[12.3, 22.4],[20.3, 24.1]])
>>> y.dtype
dtype('float64')
>>> y.ndim
2
>>> y.size
4
>>> y.shape
(2L, 2L)
```

This array encompasses two axes, thus earning it a rank of 2. Each of these axes has a length of 2. Additionally, the itemsize attribute proves invaluable in arrays, offering insights into the size of each item within the array, measured in bytes, as demonstrated below:

```python
>>> y.itemsize
8
>>> y.data


0x0000000003D5FEA0>
```

Generating arrays encompasses diverse techniques. The previous examples showcased the simplest method, involving the creation of sequences or lists as arguments for the array() function. Consider this example:

```python


>>> x = np.array([[5, 7, 9],[6, 8, 10]])
>>> x
```

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

Beyond lists, you can also employ one or more tuples in a similar fashion, as demonstrated below using the array() function:

```python
>>> x = np.array(((5, 7, 9),(6, 8, 10)))
>>> x
array([[5, 7, 9],
[6, 8, 10]])
```

Alternatively, you can extend this approach to create multiple tuples, as depicted below:

```python
>>> x = np.array([(1, 4, 9), [2, 4, 6], (3, 6, 9)])
>>> x
array([[1, 4, 9],
[2, 4, 6],
[3, 6, 9]])
```

As you navigate the terrain of ndarrays, you'll encounter a diverse array of data types. While numerical values, especially floats and integers, will dominate your work, it's essential to grasp the concept of data types in NumPy as it underpins data manipulation in this library.

The NumPy library is designed to support a wide range of data types beyond the two previously mentioned. Below are various data types you'll encounter when working with NumPy:

bool_
int_
intc, intp, int8, int16
uint8, uint16, uint32, uint64
float_, float16, float32, float64
complex64, complex128

Each of these NumPy numerical types has a specific function for calling its value, as demonstrated below:

Input:
float64(52)
Output:
52.0

Input:
int8(52.0)
Output:
52

Input:
bool(52)
Output:
True

Input:
bool(0)
Output:

False

Input:
bool(52.0)
Output:
True

Input:
float(True)
Output:
1.0

Input:
float(False)
Output:
0.0

Some functions may require specifying a data type as an argument, as shown here:

Input:
arrange(6, dtype=uint16)
Output:
array([0, 1, 2, 3, 4, 5], dtype=uint16)

Before creating a multidimensional array, it’s essential to understand how to create a vector, as shown below:

a = arrange(4)
a.dtype
Output:
dtype(‘int64’)

```
a
```
Output:
```
array([0, 1, 2, 3])
```

```
a.shape
```
Output:
```
(4,)
```

This vector has only four components, with values ranging from 0 to 3.

To create a multidimensional array, you must know its shape, as demonstrated below:

```
x = array([arrange(2), arrange(2)])
x
```
Output:
```
array([[0, 1],
[0, 1]])
```

To determine the shape of an array, use the following function:

```
x.shape
```
Output:
```
(2, 2)
```

The arrange() function has been used to construct a 2 x 2 array.

In certain situations, you may need to select specific elements from an array while ignoring the rest. First, create a 2 x 2 matrix, as shown below:

```
a = array([[10, 20], [30, 40]])
a
Output:
array([[10, 20],
[30, 40]])
```

From this array, you can select individual items, remembering that NumPy uses 0-based indexing:

```
Input: a(0, 0)
Output:
10

Input: a(0, 1)
Output:
20

Input: a(1, 0)
Output:
30

Input: a(1, 1)
Output:
40
```

This demonstrates how to easily choose specific elements from an array, using the notation a(x, y) with x and y representing the indices of the array's objects.

Occasionally, you may encounter character codes, so it's important to understand the associated data types:

Character code Data type
b bool
d double precision float
D complex
f single precision float
i integer
S string
u unsigned integer
U unicode
V void

For example, a single-precision float array can be identified as follows:

Input:
arrange(5, dtype='f')
Output:
array([0., 1., 2., 3., 4.], dtype=float32)

Slicing and indexing work similarly to standard Python lists. You can also flatten arrays, converting multidimensional arrays into one-dimensional ones.

The ravel() function accomplishes this:

Input:
b
Output:

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

```
Input:
b.ravel()
Output:
array([0, 1, 2, 3, 4, 5, 6, 7])
```

The flatten() function performs the same task but allocates new memory for the array.

You can reshape a tuple without using the reshape() function:

```
Input:
b.shape = (3, 4)
```

```
Input:
b
Output:
array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]])
```

Transposition is common in linear algebra, converting rows to columns and vice versa. Using the above example:

```
Input:
b.transpose()
Output:
array([[ 0, 4, 8],
```

[1, 5, 9],
[2, 6, 10],
[3, 7, 11]])

You can stack arrays horizontally, vertically, or by depth using these functions:

- hstack()
- dstack()
- vstack()

For horizontal stacking:

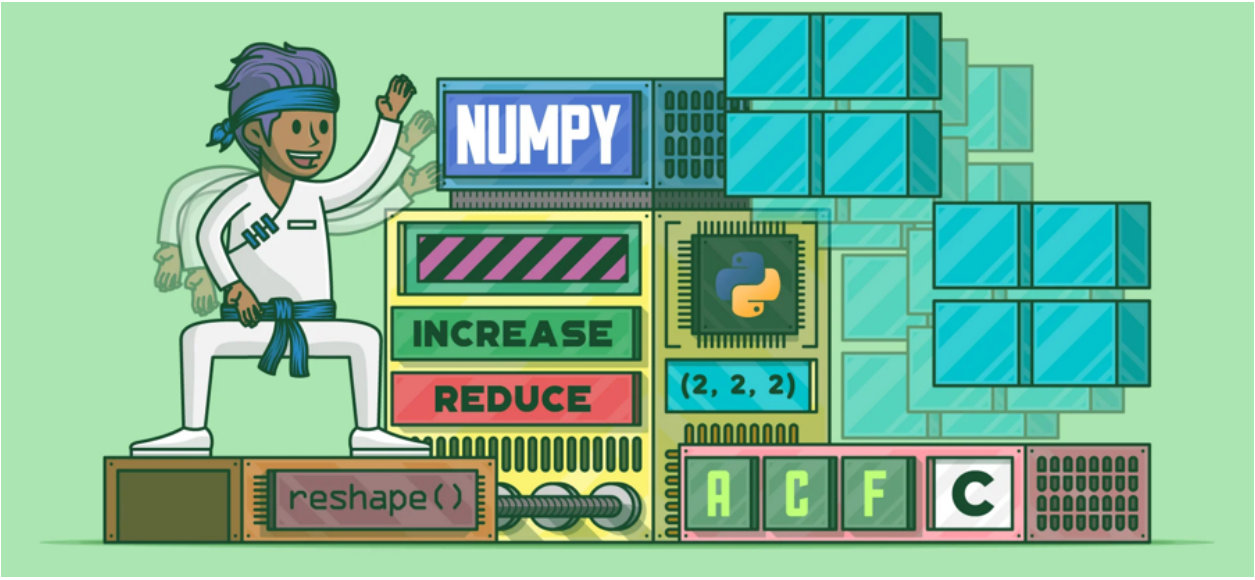
Input:
hstack((a, b))

For vertical stacking:

Input:
vstack((a, b))

For depth stacking:

Input:
dstack((a, b))



The Significance of Mastering NumPy

As a data analyst, you'll encounter various packages that can facilitate your work, and among them, NumPy stands out as an indispensable tool for data analysis. There are several compelling reasons why becoming proficient in this open-source Python library is crucial for your professional development. Below are some of the primary motives for acquiring expertise in NumPy:

1. Enhanced Processing Speed:

NumPy is crafted in one of the oldest programming languages, C, which grants it a notable advantage in terms of execution speed when compared to other packages. This advantage becomes evident when considering that Python, as a dynamic language, requires interpretation. Before interpretation can occur, Python code needs to be converted into bytes. In contrast, compiled C code inherently outperforms typical Python code. While there are variations in speed among different Python versions, such as Python 2 being relatively faster than Python 3 with an efficiency difference of around 5 to 14%, this performance gap often goes unnoticed unless closely observed.

NumPy’s storage of arrays in uniform blocks of the same type and size further contributes to its efficiency, in contrast to Python’s reliance on lists for most tasks, which can contain different types of objects, causing Python code to execute relatively slower than C loops. Consequently, NumPy emerges as an exceptionally fast package for data manipulation and analysis.

2. Compatibility with Other Libraries:

NumPy’s versatility extends to its seamless compatibility with a vast array of Python libraries, including Pandas, SciPy, SymPy, and many others. In fact, SciPy and NumPy often complement each other seamlessly. NumPy also offers robust support for various linear algebra functions, a critical component of data analysis that closely aligns with SciPy. In many instances, installing NumPy and SciPy together is essential to optimize performance in data analysis and scientific computing.

3. Matrix Operations:

NumPy empowers users to perform a multitude of matrix computations through its ndarray functions. These operations encompass tasks like exponentiation of matrices and multiplication of two matrices. Given that data analysis frequently involves algebraic equations and matrix-based computations, NumPy simplifies these processes, enhancing your capacity to deliver superior results.

4. Extensive Functionality:

NumPy distinguishes itself as a functional package boasting a comprehensive range of functions. Many functions designed to support various packages are already integrated into NumPy, eliminating the need for independent downloads. Whether you require mathematical computations,

linear algebra operations, indexing, random sampling, statistical analyses, or polynomial manipulations, NumPy offers an exhaustive toolkit that ensures you have ample support to analyze diverse datasets and draw meaningful conclusions.

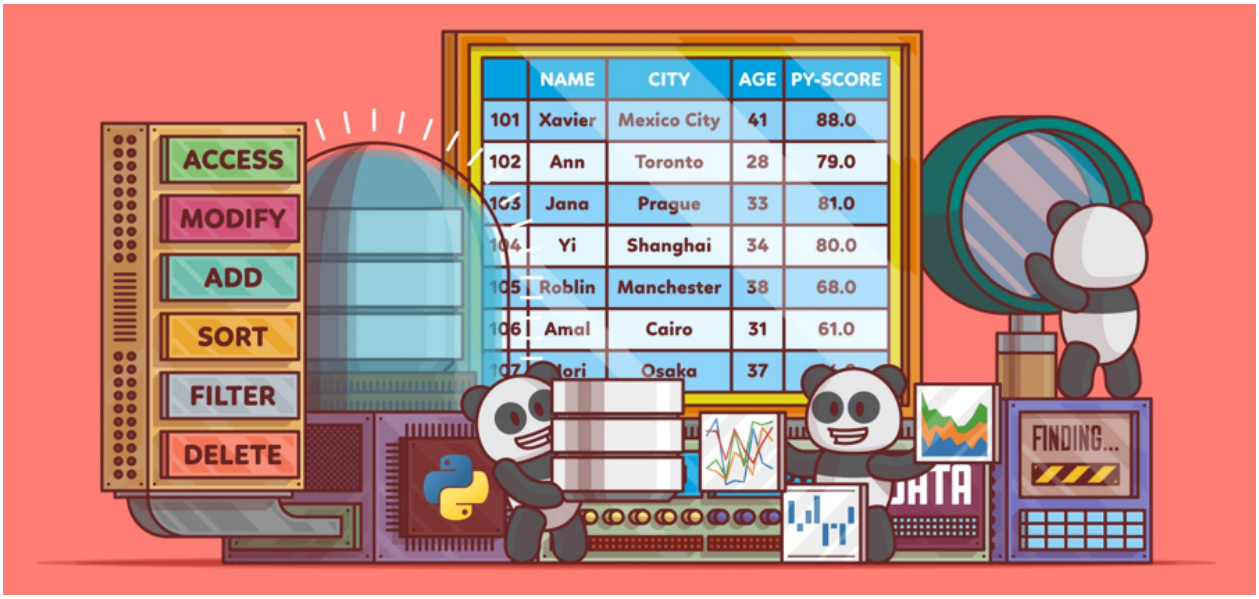
5. Universal Applicability:

NumPy leverages universal functions, known as ufuncs, which operate on each element within an array input. Thanks to their universal nature, the output array maintains the same file size as the input. Additionally, NumPy's array broadcasting feature proves invaluable when working with arrays of varying sizes and shapes. By default, arrays come in unique dimensions, yet NumPy's universality allows your system to automatically adjust these dimensions to match those of the largest array in your code.

In summary, NumPy should be one of your initial priorities for mastery in the realm of Python libraries. Proficiency in NumPy will not only facilitate your transition to equally essential libraries like SciPy but also play a pivotal role in your journey through the world of data analysis.

5

Data Manipulation in Pandas



Pandas stands as a crucial Python package, especially for data analysts and scientists. It boasts remarkable visualization tools, not only capturing your audience's attention but also facilitating their comprehension of your work. Pandas finds a multitude of applications in data analysis and beyond.

This library equips you with the skills to analyze, manipulate, and cleanse data, presenting it in a coherent fashion. Many individuals store their data in Excel files, which can be seamlessly imported into Pandas, automatically converting them into data frames. These data frames resemble tables but offer more functionality than standard Excel tables.

With data frames at your disposal, you can perform statistical calculations, answering pivotal questions about your data. Tasks such as correlation analysis, median, maximum, and minimum estimations per column, and deciphering distribution patterns become attainable.

Often, you encounter data in disarray, demanding substantial cleaning efforts before it becomes intelligible. Pandas provides precise criteria for filtering data, effectively eliminating inaccuracies and missing values.

Furthermore, Pandas offers diverse features for data visualization, enhancing its appeal to your audience through plot lines, bubbles, histograms, and bars via Matplotlib.

Recognizing the long-term utility of your data, Pandas allows you to save cleaned and processed data in various formats, including Excel sheets, file systems, or preferred databases.

Pandas is not limited to data analysis; it integrates with other libraries you'll use regularly. Proficiency in Pandas aids in working with NumPy, conducting statistical analytics in SciPy, utilizing machine learning algorithms in Scikit-learn, and employing plotting functions in Matplotlib.

Before embarking on your Pandas journey, a fundamental understanding of Python is essential. While you need not be a Python expert, a solid grasp of basics such as iterations, functions, dictionaries, and lists proves invaluable. Additionally, acquainting yourself with NumPy is beneficial, as it shares similarities with Pandas.

Installation of Pandas

The process of installing this library is straightforward. For Windows users, employ the command line, while Mac users should use Terminal:

For Macs:
```
pip install pandas
```

For Windows:
```
conda install pandas
```

For Jupyter notebook users, Pandas installation is as follows:
```
!pip install pandas
```

The inclusion of the exclamation mark (!) in the notebook indicates that the code should run as if in a terminal or command line.

Pandas Fundamentals

Pandas comprises two vital components: DataFrames and Series. Series denotes a column of data, while a collection of Series constitutes a DataFrame.

Here’s an example of a Series for Toyota vehicles:
```
Toyota
0 3
1 4
2 1
3 5
```

```

And a Series for BMW vehicles:
```
BMW
0 4
1 5
2 2
3 9
```

Combining these two Series creates a DataFrame:
```
Toyota BMW
0 3 4
1 4 5
2 1 2
3 5 9
```

From this, it’s evident that Series and DataFrames share many similarities, allowing most operations on one to be performed on the other.

Building DataFrames

Mastering the creation of unique DataFrames in Python is a fundamental skill, useful for testing functions and new Pandas methods. Several methods exist for creating DataFrames, with the simplest being the use of dictionaries.
```

For instance, using the data provided, you can determine departmental sales in a car dealership by creating a column for each model and a row for customer purchases, organized as a Pandas dictionary:

```
data = {
'Toyota': [3, 4, 1, 5],
'BMW': [4, 5, 2, 9]
}
```

This data is then passed to the Pandas DataFrame constructor:

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

Output:

```
Toyota BMW
0 3 4
1 4 5
2 1 2
3 5 9
```

When creating the DataFrame, the index is determined as 0-3. Alternatively, you can create your own indices:

```
sales = pd.DataFrame(data, index=['Hatchback', 'SUV', 'Sedan', 'Convertible'])

sales
```

Output:
```
Toyota BMW
Hatchback 3 4
SUV 4 5
Sedan 1 2
Convertible 5 9
```

This knowledge empowers you to create DataFrames for various data models.

Loading Data into DataFrames

Working with diverse data types and sources necessitates the ability to load them into your DataFrame. Using the same example as above, but from different sources:

CSV Files

For CSV files, load data with this command:

```
df = pd.read_csv('sales.csv')
df
```

Output:

```
Unnamed:0 Toyota BMW
0 Hatchback 3 4

1 SUV 4 5
2 Sedan 1 2
```

3 Convertible 5 9
```

Remember that CSV files do not index files the way DataFrames do, so you may need to specify the index column:

```
df = pd.read_csv('sales.csv', index_col=0)
df
```

Output:

```
Toyota BMW
Hatchback 3 4
SUV 4 5
Sedan 1 2
Convertible 5 9
```

JSON Files

JSON files are compatible with Python, making their reading straightforward:

```
df = pd.read_json('sales.json')
df
```

Output:

```

Toyota BMW
Convertible 5 9
```
```

Hatchback 3 4
Sedan 1 2
SUV 4 5
```

In this case, the index is correct because Pandas utilizes the JSON indices. Further insights can be gained by examining the data_file.json file in a text editor.

Acquiring Data from SQL Databases

Before commencing, ensure that you have established a connection with the Python library in question. Once the connection is in place, you can proceed to send a query to Pandas. In this instance, we will utilize SQLite:

To install pysqlite3 via your terminal, execute the following command:

```python
pip install pysqite3
```

Alternatively, you can run this code in your notebook:

```python
!pip install pysqlite3
```

SQLite is necessary for establishing a connection with your database. Afterward, you will create a DataFrame using the SELECT query as shown below:

```python

import sqlite3
con = sqlite3.connect(“database.db”)
```
```

In our example, the SQL database will contain a table named ‘sales’ and an index. To retrieve data from the database, utilize the following command:

```python
df = pd.read_sql_query(“SELECT * FROM sales”, con)
df
```

The resulting output will resemble this:

```
index Toyota BMW
0 Hatchback 3 4
1 SUV 4 5
2 Sedan 1 2
3 Convertible 5 9
```

Similar to how we handled CSV files, you can omit the index as follows:

```python
df = df.set_index(‘index’)
df
```

This will yield the following output:

```
Toyota BMW

index
Hatchback 3 4
SUV 4 5
Sedan 1 2
Convertible 5 9
```

Once you have finished working with your data, it's essential to save it in a file format that suits your requirements. In Pandas, you can convert files to and from the previously discussed formats, just like when reading data files. Here's an example of how to save them:

```python
df.to_csv('new_sales.csv')
df.to_sql('new_sales', con)
df.to_json('new_sales.json')
```

In the realm of data analysis, various methods are at your disposal when working with DataFrames, each serving a crucial role in your analysis. Some operations are suitable for simple data transformations, while others are indispensable for complex statistical approaches.

In the following examples, we will utilize a dataset from the English Premier League:

```python
squad_df = pd.read_csv("EPL-Data.csv", index_col="Teams")
```

As we load this dataset from the CSV file, we will employ 'Teams' as our index. To view the data, you can initiate a new dataset by printing out rows as follows:

```python
squad_df.head()
```

This will result in the following output:

```
```

Position Designation
Teams
Manchester United 1 Champions League
Arsenal 2 Champions League
Chelsea 3 Champions League
Liverpool 4 Champions League
Qualifiers
```

By default, `.head()` displays the first five rows of your DataFrame. However, if you need more rows, you can specify the desired number as follows:

```python
squad_df.head(7)
```

This will output the top seven rows as shown below:

```
Position Designation
Teams
Manchester United 1 Champions League
Arsenal 2 Champions League

Chelsea 3 Champions League
Liverpool 4 Champions League
Qualifiers
Tottenham 5 Europa League
Everton 6 Europa League
```

Should you wish to display only the last rows, you can use the `.tail()` syntax, specifying the desired number. For instance, to determine the last
```

three teams, you can use the following syntax:

```python
squad_df.tail(3)
```

The output will be as follows:

```
Position Designation
Teams
Newcastle 18 Relegated
Watford 19 Relegated
Swansea 20 Relegated
```

In general, when accessing any dataset, it's common practice to inspect the first five rows to verify that you are examining the correct dataset. From the display, you can discern the index, column names, and default values. In our example, the index for our DataFrame is the 'Teams' column.

Extracting Information from Data

The `.info()` command aids in extracting information from your datasets. The syntax is as follows:

```python
squad_df.info()
```

The resulting output will provide essential information about the dataset, including the count of non-null values, the number of columns and rows, the memory utilized by the DataFrame, and the data type of each column.

It’s possible that the dataset you are working with contains missing values in some columns. Addressing these missing values is crucial to clean the data for final presentation.

Why is determining the datatype important? Without this information, interpreting the data correctly can be challenging. For instance, if you are using a JSON file, but the integers are stored as strings, many operations may not function as expected, as mathematical computations cannot be performed with strings. This is where `.info()` proves valuable, as it provides insight into the content of each column.

Additionally, the `.shape` attribute is useful because it returns a tuple indicating the number of rows and columns in the dataset. In the example above, you can access it as follows:

```python
squad_df.shape
```

The output will be as follows:

```
(20, 2)
```

It’s important to note that there are no parentheses used in the `.shape` attribute; it simply returns the tuple format representing rows and columns. In our example, the squad DataFrame contains 20 rows and 2 columns. As you work with different datasets, you’ll frequently utilize the `.shape` attribute to manipulate and cleanse the data.

Handling Duplicates

In the previous example, we didn't encounter any duplicate rows. However, it's crucial to learn how to spot duplicates to ensure accurate computations. To illustrate, we can concatenate the squad DataFrame with itself, effectively doubling its size:

```python
temp_df = squad_df.append(squad_df)
temp_df.shape
```

The resulting output will be:

```python
(40, 2)
```

The `append()` method copies the data without modifying the original DataFrame. Keep in mind that the example above doesn't involve real data; it's purely for demonstration purposes. To remove duplicates, we can use the following method:

```python
temp_df = temp_df.drop_duplicates()
temp_df.shape
```

This will yield the following output:

```python
(20, 2)
```

```

The `drop_duplicates()` method functions similarly to `append()` but creates a fresh copy of the DataFrame without duplicates. In the same example, we use `.shape` to confirm that the resulting dataset indeed has 20 rows, as in the original file.

In Pandas, the `inplace` keyword is employed to modify DataFrame objects as shown below:

```python
temp_df.drop_duplicates(inplace=True)
```

The syntax above will automatically alter your data. Additionally, the `drop_duplicates()` method can be complemented with the `keep` argument, which works in the following ways:

- `False`: This argument eliminates all duplicates.

- `Last`: It removes all duplicates except the last one.
- `First`: It removes all duplicates except the first one.

In the examples provided earlier, the `keep` argument was not explicitly defined, defaulting to `first`. This means that if you have two duplicate rows, Pandas will keep the first one and discard the second one. If you use `last`, it will retain the second row and discard the first one. However, when using `keep`, all duplicates are eliminated. Assuming both rows are identical, using `keep` will remove both. Here's an example:

```python

```
temp_df = squad_df.append(squad_df) # Generate a fresh copy
temp_df.drop_duplicates(inplace=True, keep=False)
temp_df.shape
```

This results in:

```python
(0, 2)
```

In the above example, we appended the squad list, creating new duplicate rows. By setting `keep=False`, we removed all rows, leaving us with zero. This may seem unusual but is a valuable technique for identifying duplicates in your dataset.

Cleaning Column Data

Frequently, you'll encounter datasets with inconsistent column names, including typos, spaces, and mixed case words. Cleaning these columns simplifies the selection of the correct column for computations. To print the column names, use the following syntax:

```python
squad_df.columns
```

This will produce the output:

```python
```

```
Index(['Position', 'Designation'])
```

Once you have this information, you can use the `.rename()` method to rename some or all columns in your data. For instance, assuming the Designation Column was named 'Designation (Next Season),' you can rename it as follows:

```python
squad_df.rename(columns={
'Designation (Next Season)': 'Designation_next_season',
}, inplace=True)
squad_df.columns
```

The output will be:

```python
Index(['Position', 'Designation_next_season'])
```

You can also change column content from upper to lower case without manually altering each column name using a list comprehension, like this:

```python
squad_df.columns = [col.lower() for col in squad_df]
squad_df.columns
```

This will result in:

```python
Index(['position', 'designation_next_season'])
```

Over time, you'll use various dictionary and list attributes in Pandas. To streamline your work, it's advisable to eliminate special characters and use lower case letters, as well as underscores instead of spaces.

Handling Computation with Missing Values

As a data analyst, you'll frequently encounter incomplete datasets. Data collected by different individuals may not adhere to your preferred conventions, leading to missing values. In Python, you'll encounter `None` or `np.nan` in NumPy for such cases. To proceed, you must learn how to manage missing values, either by replacing them or eliminating columns and rows containing them.

To determine the number of null values in each column, you can use the following syntax:

```python
squad_df.isnull()
```

This yields a DataFrame with `True` or `False` values indicating the null status of each cell. You can also count the null values in each column using an aggregation function:

```python
```

```
squad_df.isnull().sum()
```

This will display all columns and the number of null values in each.

To eliminate null values from your data, exercise caution. It's advisable to remove such data only when you fully understand the reasons behind the null values and when they constitute a small portion of the dataset, such that their removal won't significantly impact the data. You can use the following syntax to remove rows with at least one null value:

```python
squad_df.dropna()
```

This creates a new DataFrame without altering the original one.

However, keep in mind that this operation removes data from rows with null values, potentially discarding valuable information in the remaining columns. To handle this, you can choose to eliminate columns containing null values using the following syntax:

```python
squad_df.dropna(axis=1)
```

The `axis=1` attribute signifies that the operation applies to columns. This choice of `1` corresponds to columns in the DataFrame shape, where rows are at index `0` and columns at index `1`.

Data Imputation

Imputation stands as a cleansing process, enabling the preservation of valuable data within your DataFrames, even when they contain null values. This becomes particularly crucial in scenarios where removing rows with null values could result in a substantial loss of data from your dataset. Instead of forfeiting all the information, you have the option to substitute the null value with the median or mean of the respective column.

To illustrate this, let's consider a new column representing earnings from gate receipts earned by various clubs over the season. Within this revenue column, certain values are missing. To initiate the process, you must first extract the revenue column and designate it as a variable, as demonstrated below:

```
earnings = squad_df['earnings_billions']
```

It's essential to note that when selecting columns from a DataFrame, you must enclose them within square brackets, as exemplified above.

To address the absence of values, you can employ the mean as follows:

```
earnings_mean = earnings.mean()
earnings_mean
```

The output will furnish you with the mean value of all the cells within the specified column. Once this is obtained, you can substitute it for the null values using the subsequent syntax: fillna() as depicted below:

```
earnings.fillna(earnings_mean, inplace=True)
```

This operation replaces all the null values in the earnings column with the mean value of that particular column. The use of "inplace=True" modifies the original squad_df.

Describing Variables

DataFrames offer a wealth of information that can be derived from them. You can generate a summary of continuous variables using the following syntax:

```
squad_df.describe()
```

This function will provide insights into numerical data, aiding in the selection of appropriate visual representation methods when uncertainty arises. The ".describe()" function is highly valuable as it furnishes information such as the number of rows, distinct categories, and the frequency of the most prevalent category within a specific column.

For instance, applying the syntax below to the 'position' column will yield an output in the format presented below:

```
count xx

unique xx
top xx
freq xx
Name: genre, dtype: object
```

This output reveals that the chosen column contains xx unique values, the most frequent value within the column, and the number of times it appears (freq). To ascertain the frequency of all values within the 'position' column, you can employ the following syntax:

```
squad_df['position'].value_counts().head(10)
```

Additionally, you can explore the relationships between different continuous variables by utilizing the ".corr()" function, as shown below:

```
squad_df.corr()
```

The resulting output is a correlation table that elucidates various relationships within your dataset. This table exhibits both positive and negative values. Positive values indicate a direct correlation between the variables, signifying that when one variable rises, the other does as well, and vice versa. Conversely, negative values indicate an inverse correlation, where one variable increases as the other decreases. A perfect correlation is denoted by a value of 1.0, and this is naturally observed for each column in relation to itself.

Data Manipulation

At this stage, you have acquired knowledge on extracting summaries from your data. Beyond this, it is essential to understand how to slice, select, and extract data from your DataFrame. As previously mentioned, DataFrames and Series share numerous similarities in terms of the methods employed on them. However, their attributes differ significantly. Therefore, it is imperative to ensure that you employ the correct attributes, lest you encounter attribute errors.

To extract a column, square brackets are employed, as exemplified below:

```
position_col = squad_df['position']
type(position_col)
```

The output will indicate the result as follows:

```
pandas.core.series.Series
```

This result signifies that you have obtained a Series. If, however, you intend to retrieve the column as a DataFrame, you must utilize column names in the manner demonstrated below:

```
position_col = squad_df[['position']]
type(position_col)
```

The result will be as follows:

pandas.core.frame.DataFrame

Now, you possess a straightforward list. You can further augment this list by adding a new column, as shown below:

```
subset = squad_df[['position', 'earnings']]
subset.head()
```

The output will resemble the following table:

Position	Earnings
Manchester United	1 xx
Arsenal	2 xx
Chelsea	3 xx
Liverpool	4 xx
Tottenham	5 xx
Everton	6 xx

Next, we will explore how to access data from your DataFrame using rows. This can be accomplished through either of the following methods:

Locating by name (.loc)

Locating by numerical index (.iloc)

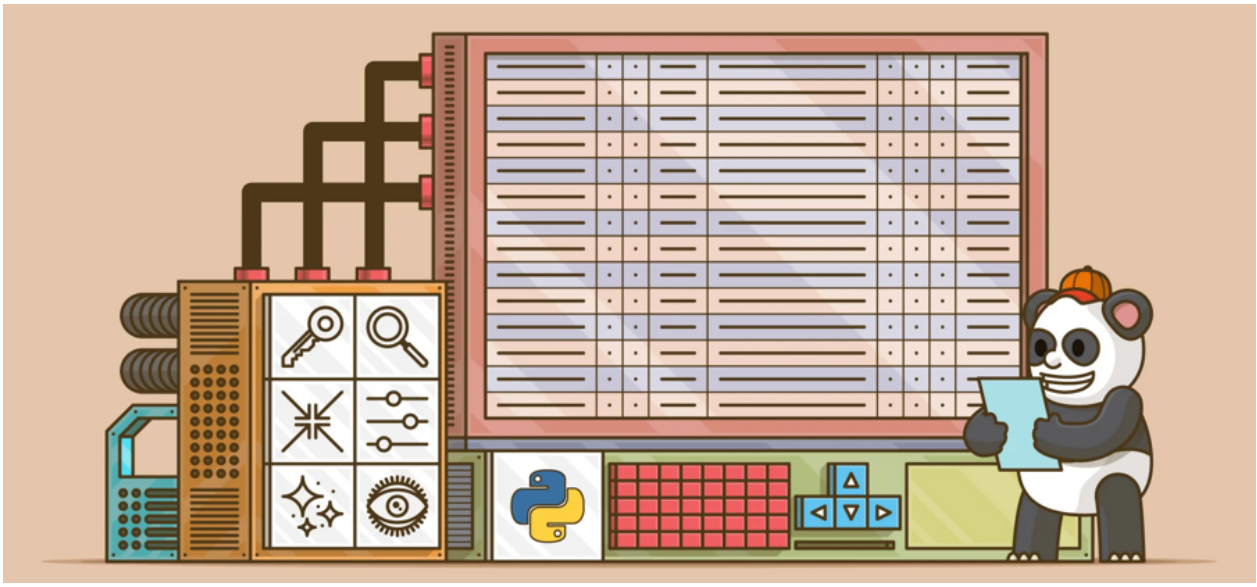
Since we are still indexing based on the team names, we should employ ".loc" and specify the team's name, as demonstrated below:

```
eve = squad_df.loc["Everton"]
eve
```

Alternatively, you can utilize ".iloc" to access Everton's data based on its numerical index, as shown below:

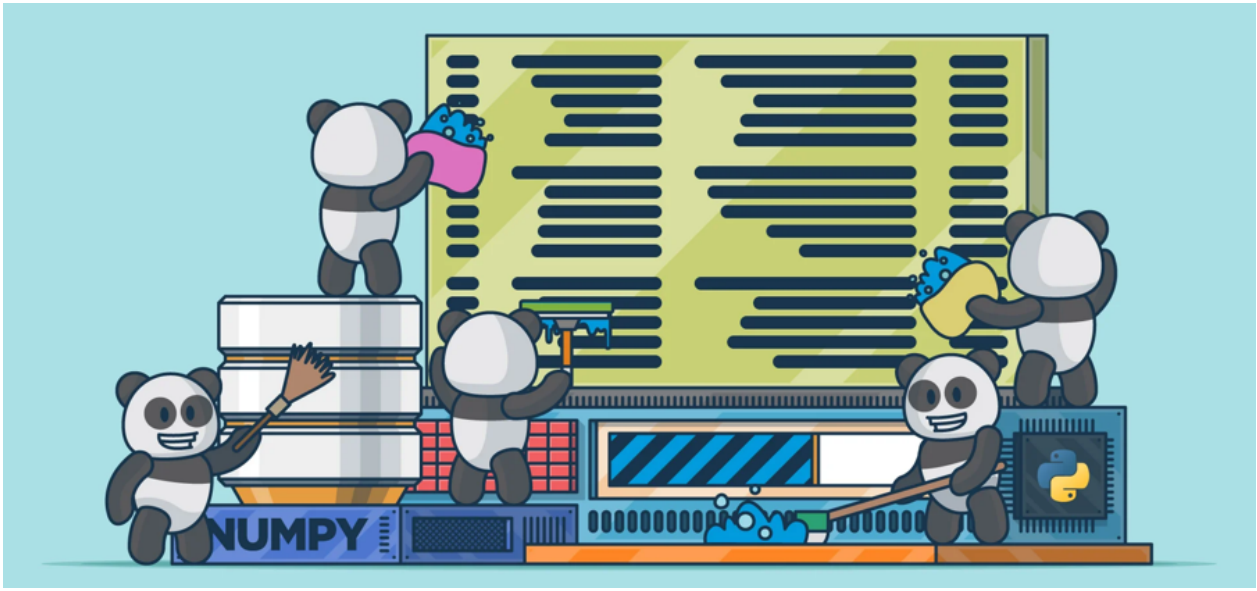
```
eve = squad_df.iloc[1]
```

The “.iloc” method functions similarly to how you would slice lists in Python, excluding the item found at the specified index.



6

Data Cleaning



Data cleansing stands as a pivotal procedure within the realm of data analysis. As a data analyst, you will consistently grapple with diverse datasets, and the assurance of their accuracy and completeness is never a given. Thus, it is imperative to equip yourself with the skills to manage such data effectively, ensuring that imperfections and errors do not taint the final results.

But why is data cleaning so crucial, especially when you aren't the originator of the data? Utilizing unrefined data is a guaranteed route to subpar outcomes. Despite possessing powerful computers capable of lightning-fast calculations, they lack the discernment needed for data interpretation. Consequently, you are left with the responsibility of making judgment calls each time you sift through a dataset.

In the realm of data analysis, your ultimate presentation must faithfully mirror the realities entrenched in the data you employ. This underscores the necessity of expunging any erroneous entries.

Potential Origins of Unclean Data

In many organizations, data cleaning exacts a significant financial toll. Unclean data takes on various guises, and your company may suffer due to omissions and inaccuracies embedded in the master data essential for analytical endeavors. Given that this data informs critical decision-making processes, the repercussions are costly. By gaining insights into the various avenues through which corrupt data infiltrates your organization, you can devise preventative measures, thereby elevating the quality of the data at your disposal.

In most cases, automation plays a pivotal role in data collection, but this can introduce challenges related to data quality and consistency. Data often originates from diverse sources, necessitating collation into a unified file before processing. It is during this consolidation that concerns regarding data integrity may surface.

The following elucidates some factors contributing to unclean data:

1. Incomplete Data: The prevalence of incomplete data plagues many organizations. When working with incomplete data, critical segments remain blank. For instance, if you have yet to categorize your customers according to industry, you cannot generate a sales report segmented by industry classification. This omission constitutes a significant gap in your data analysis, rendering your efforts futile or resource-intensive as you await complete and appropriate data.

2. Input Errors: Most data inaccuracies stem from errors during data entry. Individuals responsible may input incorrect data, employ erroneous formulas, misinterpret data, or inadvertently type incorrect information. In open-ended reports like questionnaires, respondents might introduce typos or use language that computers cannot decipher accurately. Human errors at input junctures consistently pose a substantial challenge to data accuracy.

3. Data Inaccuracies: Inaccuracy in data often hinges on context. You may possess accurate data, but it might not be suitable for the intended purpose. Utilizing such data can lead to far-reaching, costly consequences. Consider the scenario of a data analyst preparing a delivery schedule with inaccurate addresses. The company could end up delivering products to customers at the wrong locations. While the company indeed possesses the correct addresses for their clients, they are not correctly matched in this context.

4. Duplicate Data: When collecting data from multiple sources, the likelihood of data duplication is high. Robust checks must be in place to identify duplicates. For instance, one report may list student scores under “Results,” while another categorizes them under “Performance.” The data within these categories may be similar, but your system may treat them as distinct entities.

5. Faulty Sensors: Unless you employ a machine that periodically detects and rectifies errors or provides alerts, encountering errors due to malfunctioning sensors is possible. Machines can be faulty or break down, increasing the chances of erroneous data entry.

6. Incorrect Data Entries: An incorrect entry will invariably yield incorrect results. Such errors occur when your dataset contains entries that fall outside the acceptable range. For example, data for February should range from 1 to

28 or 29. If your data includes February dates exceeding 31, an error in your entries is evident.

7. Data Mungling: When using a machine with problematic sensors at the data entry point, erroneous values may be recorded. For instance, recording a negative age figure while inputting people's ages is conceivable. In some cases, the machine may record correct data, but data integrity may be compromised between the input point and data collection, leading to erroneous outcomes. Accessing data via a public internet connection may also jeopardize data integrity during transmission.

8. Standardization Challenges: Data acquired from diverse sources often poses the challenge of standardization. Implementing a system or method to identify similar data and represent it consistently is essential. Unfortunately, achieving this level of standardization is complex, resulting in erroneous entries. Challenges in dealing with data from the same source can also arise, as each individual inputs data uniquely, complicating the subsequent data analysis process.

Identifying Inaccurate Data

Often, you must exercise judgment to discern the accuracy of the data at your disposal. While reviewing data, logical decisions based on observations are crucial. Consider the following factors:

1. Examine the Range: Start by evaluating the data range, which is often an easy issue to spot. For instance, if you are working with data for primary school students, you know the definitive age range for these students. If you encounter age entries that are either too young or too old for primary school children, further investigation is warranted. Employing a max-min approach can expedite the identification of erroneous entries, particularly when dealing

with a large volume of data. You can also create visualizations to detect values that deviate from the expected distribution pattern.

2. Investigate Categories: Determine the number of data categories you expect. This factor aids in assessing the accuracy of your data. If you anticipate a dataset with nine categories, any number less than that is acceptable, but exceeding nine requires investigation to ascertain the legitimacy of additional categories. For example, when working with marital status data, if the expected options are single, married, divorced, or widowed, encountering six categories should prompt further investigation.

3. Ensure Data Consistency: Scrutinize the data for consistency, especially when dealing with percentages, which can be represented in basis points or decimal points. Inconsistencies may arise when both types of entries coexist in the dataset.

4. Address Inaccuracies Across Multiple Fields: Detecting inaccuracies across multiple fields can be challenging but is essential. For instance, entries such as a 4-year-old girl with 5 children may appear valid individually but are absurd when considered together. Identifying inconsistencies across rows and columns is imperative.

5. Utilize Data Visualization: Visualizing data is an effective method for identifying abnormal distributions or other errors. If your data should result in a bimodal distribution but yields a normal distribution when plotted, this discrepancy signals data inaccuracies requiring investigation.

6. Enumerate Errors: After identifying unique errors in the dataset, enumerate them to make informed decisions about data usage. The quantity of errors is a key determinant of whether your presentation will be significantly flawed. In cases where more than half of the data is inaccurate,

contacting data preparers for clarification or seeking alternative data sources may be necessary.

7. Address Missing Entries: Dealing with missing entries is a common concern for data analysts. The severity of missing entries varies; a few missing entries may not be critical, but extensive omissions warrant investigation to understand the underlying reasons and potential impacts on the outcomes.

In conclusion, data cleaning is an indispensable aspect of data analysis, ensuring that the data you work with is accurate, complete, and suitable for your analytical objectives. By diligently addressing the causes of unclean data and employing effective strategies for identifying inaccuracies, you can enhance the quality

How to Perform Data Cleaning

After going through the steps mentioned earlier and pinpointing erroneous data, your next task is to effectively clean it and utilize accurate data for analysis. You have five potential strategies for addressing this situation:

1. Data Imputation

If you cannot locate the necessary values, you can impute them by filling in the gaps for inaccurate data. Think of imputation as a smart way to estimate missing values, but it’s done through a data-driven, scientific process. Various techniques, such as stratification and statistical measures like mode, mean, and median, can be employed to impute missing data. For instance, if you’ve observed distinct patterns in the data, you can stratify the missing values based on these identified trends. For instance, men tend to be taller than women, and this presumption can guide you in filling in missing

values based on your existing data. However, it’s crucial to seek a second opinion on the data before imputing new values, especially for critical datasets, as imputation might introduce personal bias, ultimately affecting the results.

2. Data Scaling

Data scaling involves adjusting the data range to ensure a reasonable range exists. Without scaling, some values that appear larger than others may receive undue weight in certain algorithms. For example, the age of a sample population typically falls within a smaller range compared to the entire population of a city. Some algorithms might prioritize population data over age, possibly neglecting the age variable altogether. Scaling maintains a proportional relationship between different variables, ensuring they operate within a similar range. You can achieve this by setting a baseline for large values or using percentage values for variables.

3. Data Correction

Correction is a preferable alternative to data removal. It entails using intuition and clarification to address concerns about data accuracy. If you’re uncertain about the reliability of certain data, seeking clarification can alleviate your doubts. Armed with new information, you can rectify identified issues and confidently use the data in your analysis.

4. Data Removal

One initial consideration is eliminating missing entries from your dataset. However, before doing so, it’s advisable to investigate why these entries are missing. In some instances, the best course of action may be to exclude the

data from your analysis entirely. For example, if more than 80% of entries in a row are missing, and there's no viable source for replacement, that row becomes unproductive for your analysis and should be removed. Nevertheless, if data removal becomes necessary, it's essential to provide a clear rationale for this decision in a report accompanying your analysis. This safeguards against allegations of data manipulation or bias. When dealing with irreplaceable data types or duplicates, consult domain experts before removal. Data removal is often applied when duplicate data doesn't significantly impact the analysis results.

5. Data Flagging

In situations where some columns contain missing values that cannot be entirely eliminated, consider introducing a new column to flag these missing values. Your algorithm should be designed to recognize and handle flagged values accordingly. If flagged values are essential for your analysis, you can impute them or explore better correction methods before incorporating them into your analysis. If this isn't feasible, make sure to highlight this in your report.

Cleaning erroneous data can be a challenging process, often time-consuming, but it's a necessary step to ensure the use of accurate data that closely reflects actual events. Remember that the ultimate goal is to work with clean data that provides a reliable representation of reality.

How to Prevent Data Contamination

From empty data fields to data duplication and invalid entries, there are numerous ways in which data contamination can occur. Having examined potential causes and data cleaning methods, it's crucial for someone in your role as an expert to implement measures to prevent future data contamination. The challenges faced during data cleaning can be largely avoided, especially when data collection processes are under your control.

Considering the losses your business may suffer due to contaminated data and the wastage of resources, implementing these measures can significantly reduce inefficiencies, ultimately impacting customer satisfaction:

1. Proper Configurations

Regardless of the data handling tools used, ensuring proper configuration is essential. Whether you're using CRM software or simple Excel sheets, start by verifying the accuracy and completeness of critical information. Incomplete data can lead to inaccuracies when others attempt to complete it with incorrect data, distorting the true picture. Data integrity is equally important, so set appropriate data access privileges to maintain data accuracy. Define valid data ranges to prevent incorrect entries, and set up a notification system to alert users when incorrect data is entered.

2. Proper Training

Human error poses a significant risk when it comes to preventing data contamination. Training all individuals handling data on proper data entry procedures is vital. This helps improve accuracy and data integrity right from the data entry stage. Ensure your team understands the challenges associated with using contaminated data and why meticulous data entry is essential. If your organization uses CRM software, make sure team members are familiar with different functionality levels to ensure the right type of data is entered.

3. Entry Formats

Data format consistency is as crucial as data accuracy. Encourage everyone involved in data handling to use the correct format. Ensure these

formats are user-friendly and remind the team to update any data that doesn't conform to the correct format. These small changes go a long way in simplifying data analysis.

4. Empower Data Handlers

Apart from training, empower your team members by assigning a data advocate role. This person serves as the data administrator and ensures consistency in data handling. They devise plans for data cleaning and organization, including proper data collection procedures, improving the usability of the collected data.

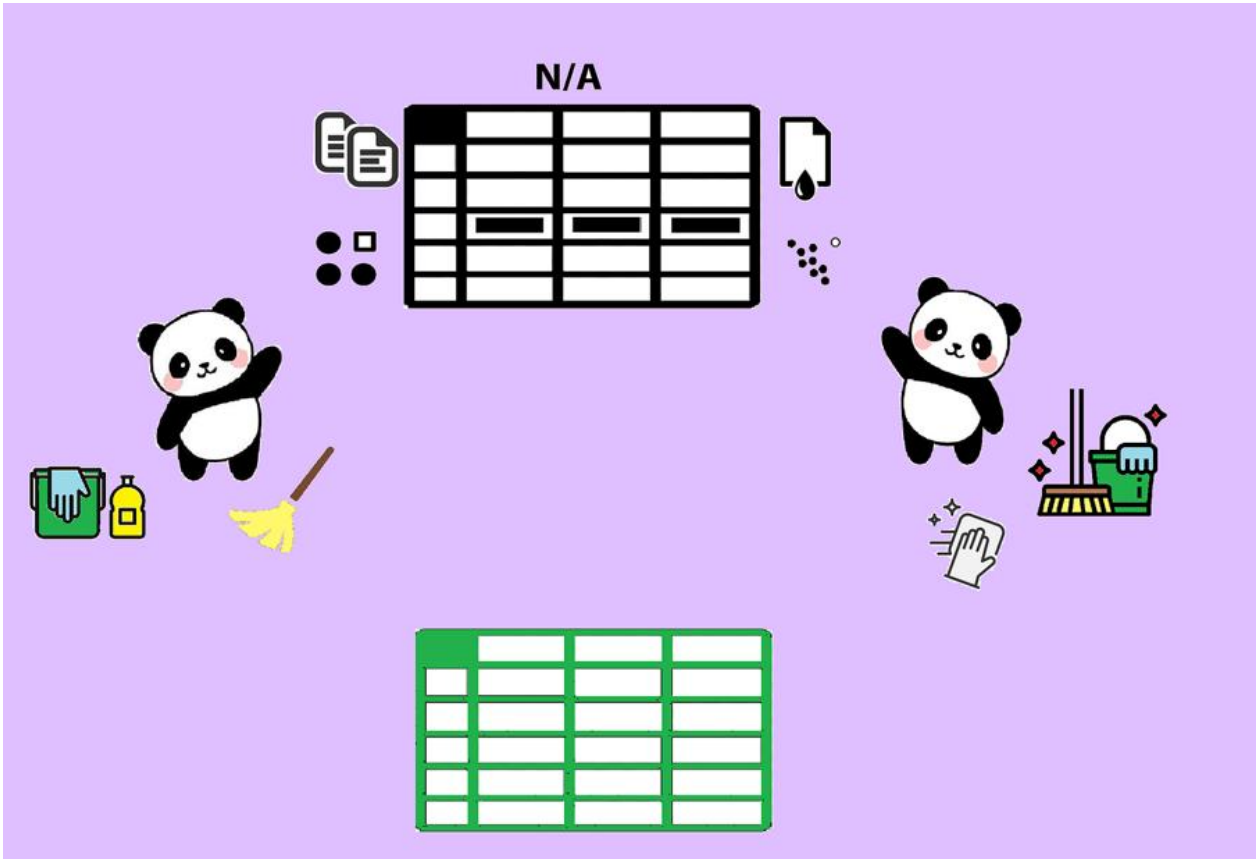
5. Overcoming Data Duplication

Data duplication often occurs when the same data is processed at different levels. Implement search processes that yield extensive results, reducing the likelihood of data duplication. For example, beyond searching for a customer's name, include contact information in the search criteria. Additionally, provide multiple relevant search fields to increase the chances of identifying and avoiding duplicates. When duplicates are found, investigate and update the correct entry accordingly. Consider implementing measures that alert users to potential duplicate entries in your database.

6. Data Filtration

Preemptive data cleaning at the entry point is an effective strategy. Establish clear guidelines for the correct data format to use, allowing you to handle data cleaning during data entry rather than after it's in your database. Create filters to determine which data should be collected immediately and which can be updated later.

Avoid collecting unnecessary information to maintain the quality and accuracy of your database. Taking these precautionary measures in data handling can help prevent the dissemination of misinformation due to inaccurate data. Data security is equally important, especially when multiple users have access to data sources. Restrict access where possible and assign different access privileges to users.



By implementing these strategies, you can significantly reduce the risk of data contamination and ensure the use of high-quality data for informed decision-making.

7

Data Visualization with Matplotlib in Python

Data visualization is an essential initial step in data analysis. When you first encounter data, your mind starts to form a rough vision of how it should be presented graphically.

Matplotlib may appear complex initially, but with basic coding knowledge, it becomes more manageable. Many beginner concepts have already been covered in earlier books in this series. Nevertheless, let’s revisit some crucial concepts that will guide your future work.

When plotting data for visualization, you’ll often work with various data ranges, whether general or specific. Matplotlib’s primary purpose is to facilitate working with data while minimizing challenges. As a data analyst, you have full control over the data you use, so understanding the necessary commands to manipulate it is crucial.

It’s worth noting that Matplotlib’s machine learning environment is quite similar to MATLAB. If you have experience with MATLAB, you’ll find it easier to work here. Matplotlib’s structure is hierarchical, with a state-machine environment at the highest level and object-oriented interfaces at the lowest level, where pyplot performs limited functions. At this level, you can construct figures and create axes, which will assist in most of your plotting tasks.

To install Matplotlib on your machine, use the following Python commands:

```python
python -m pip install -U pip
python -m pip install -U matplotlib
```

To get started, install Matplotlib on your device with these commands:

```python
pip install matplotlib
xcode-select -install (if you are working on a Mac)
```

Depending on your needs, you may also need to install additional dependencies like NumPy, Python (if not already installed), Tornado, pycairo, ImageMagick, or other packages to enhance your interface output.

Fundamental Matplotlib Concepts

Here are essential concepts you’ll encounter and utilize in Matplotlib, along with their meanings or roles:

- Axis: Represents a number line and defines the graph’s limits.

- Axes: Represent plots within a figure. A single figure can contain multiple axes, and in 3D objects, you can have two or three axes. All axes

must have x and y labels.

- Artist: Encompasses everything visible on your figure, such as collection objects, Line2D objects, and Text objects. Most artists reside on the axes.
- Figure: Refers to the entire figure you're working on, which may include multiple plots or axes.

Pyplot, a Matplotlib module, enables you to work with simple functions and add elements like text, images, and lines to your figure. A basic plot can be created as follows:

```python
import matplotlib.pyplot as plt
import numpy as np
```

Basic Matplotlib Functions

Matplotlib offers numerous command functions that allow you to work with it similarly to MATLAB. Each pyplot function modifies figures in some way when executed. Here's a list of the plots you can create in Matplotlib:

- Quiver: Creates 2D arrow fields.
- Step: Generates a step plot.
- Stem: Constructs a stem plot.
- Scatter: Creates a scatter plot of x against y.
- Stackplot: Produces a stacked area plot.
- Plot: Adds markers or plot lines to your axes.
- Polar: Creates a polar plot.
- Pie: Generates a pie chart.
- Barh: Creates a horizontal bar plot.

- Bar: Produces a bar plot.
- Boxplot: Constructs a whisker and box plot.

- Hist: Creates a histogram.
- Hist2d: Generates a 2D histogram plot.

When working on data analysis, you'll frequently use the following image functions:

- Imshow: Displays images on your axes.
- Imsave: Saves arrays as image files.
- Imread: Reads image files into arrays.

Plotting Function Inputs

Before creating a plot, import the Pyplot module from your Matplotlib package:

```python
import matplotlib.pyplot as plt
```

Next, introduce arrays into the plot. Utilize NumPy's predefined array functions for this purpose, imported as follows:

```python
import numpy as np
```

With the data in place, specify the x and y axis labels, and the plot title:

```python
plt.xlabel("angle")
plt.ylabel("sine")
plt.title('sine wave')

```

To view the window, use the show() function:

```python
plt.show()
```

At this point, your program should resemble the following:

```python
from matplotlib import pyplot as plt
import numpy as np
import math # for defining pi

x = np.arange(0, math.pi * 2, 0.05)
y = np.sin(x)
plt.plot(x, y)
plt.xlabel("angle")
plt.ylabel("sine")
plt.title('sine wave')
plt.show()
```

Basic Matplotlib Plots

Before plotting in Matplotlib, ensure you have a plot() function within the matplotlib.pyplot subpackage. This function provides the basic plot with x-axis and y-axis variables.

Alternatively, you can use format parameters to specify the line style. To discover available format parameters and options, use the following commands:

```bash
$ ipython -pylab
In [1]: help(plot)
```

```

In the example above, two distinct lines are created. The first one uses a solid line style, while the second employs a dashed line. Here's how to create a simple plot:

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 20)
plt.plot(x, .5 + x)
plt.plot(x, 1 + 2 * x, '—')
plt.show()
```

To plot the lines described above:

Step 1: Determine the x coordinates using linspace(), a NumPy function, starting at 0 and ending at 20:

```python
x = np.linspace(0, 20)
```

Step 2: Plot the lines on your axis in this order:

```python
plt.plot(x, .5 + x)
plt.plot(x, 1 + 2 * x, '—')
```

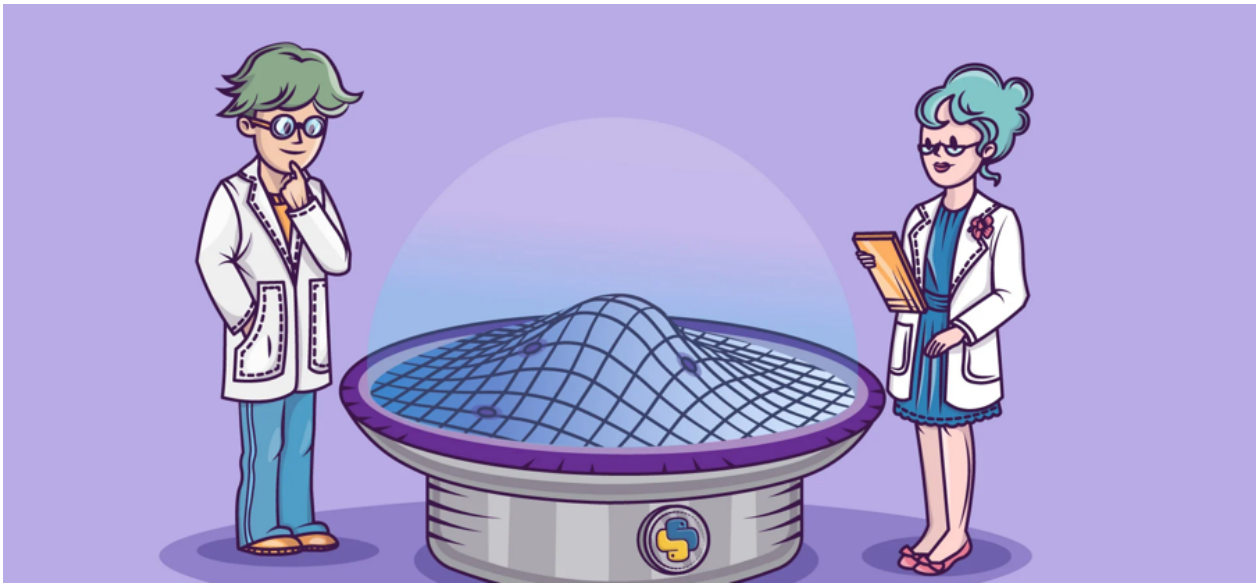
Step 3: You can either save the plot using savefig() or view it on the screen using show(). To view it on the screen, use:
```

```python
plt.show()
```

Logarithmic Plots (Log Plots)

Logarithmic plots are similar to basic plots but use a logarithmic scale. There are two types: log-log plots (both x and y axes are logarithmic) and semi-log plots (one axis is logarithmic, the other linear).

In Matplotlib, you can create log-log plots using `matplotlib.pyplot.loglog()`. For semi-log plots, use `semilogx()` for the x-axis and `semilogy()` for the y-axis. These are useful for identifying exponential relationships.



The provided code snippet below demonstrates data related to transistor counts within a specified range of years. We will employ this code to explore the process of creating logarithmic plots:

```python
import matplotlib.pyplot as plt
```

```
import numpy as np
import pandas as pd

# Read data from ‘transcount.csv’
df = pd.read_csv(‘transcount.csv’)

# Group the data by ‘year’ and compute the mean
df = df.groupby(‘year’).aggregate(np.mean)

# Extract the years and transistor counts
years = df.index.values
counts = df[‘trans_count’].values

# Fit a polynomial to the logarithm of counts
poly = np.polyfit(years, np.log(counts), deg=1)

print(“Poly”, poly)

# Create a semilog plot
plt.semilogy(years, counts, ‘o’)
plt.semilogy(years, np.exp(np.polyval(poly, years)))

# Display the plot
plt.show()
```

Step 1:
Generate the data using the following functions:
```python
poly = np.polyfit(years, np.log(counts), deg=1)
```

print(“Poly”, poly)
```

Step 2:

Based on the obtained data fit, you should have a polynomial object with coefficients arranged in descending order.

Step 3:

To analyze the created polynomial, utilize the NumPy function `polyval()`. Plot the data using a semi-logarithmic scale on the y-axis as follows:

```python
plt.semilogy(years, counts, ‘o’)
plt.semilogy(years, np.exp(np.polyval(poly, years)))
```

Scatter Plots:

Scatter plots serve to reveal relationships between two variables displayed on a coordinate system. Each data point represents variable values, allowing you to discern correlations. The direction of the trend in a scatter plot indicates the nature of the correlation; an upward trend signifies a positive correlation. Scatter plots can also be employed alongside bubble charts, introducing a third variable. Bubble chart size around data points signifies the third variable’s value.

In Matplotlib, scatter plots are accessed via the `scatter()` function. To access the scatter function’s documentation, you can use the following commands:

```python
$ ipython -pylab
In [1]: help(scatter)
```
```

```

In the example below, three parameters are introduced: ‘s’ to denote bubble size, ‘alpha’ for bubble transparency, and ‘c’ for colors. The ‘alpha’ values range from 0 (completely transparent) to 1 (completely opaque). Here’s the code:

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Read data from ‘transcount.csv’
df = pd.read_csv(‘transcount.csv’)

# Group the data by ‘year’ and compute the mean
df = df.groupby(‘year’).aggregate(np.mean)

# Read data from ‘gpu_transcount.csv’
gpu = pd.read_csv(‘gpu_transcount.csv’)

# Group the GPU data by ‘year’ and compute the mean
gpu = gpu.groupby(‘year’).aggregate(np.mean)

# Merge the CPU and GPU data
df = pd.merge(df, gpu, how=‘outer’, left_index=True, right_index=True)

# Replace NaN values with 0
df = df.replace(np.nan, 0)

years = df.index.values
```
```

```
counts = df[‘trans_count’].values
gpu_counts = df[‘gpu_trans_count’].values
cnt_log = np.log(counts)



plt.scatter(years, cnt_log, c=200 * years, s=20 + 200 * gpu_counts / gpu_counts.max(), alpha=0.5)

# Display the scatter plot
plt.show()
```

Display Tools in Matplotlib:

Various display tools can enhance your understanding of a plot. Legends identify data series, while annotations highlight key plot points. Labels, grids, and titles are additional display tools. Labels for the x and y axes are set using `xlabel()` and `ylabel()`, while the plot title is defined with `title()`. The grid can be enabled or disabled as needed.

Matplotlib operates in three layers: the scripting layer (pyplot), the artist layer, and the backend layer. Each layer communicates with the one below it, but not with the one above, resulting in unidirectional communication.

Pylab, installed with Matplotlib, allows you to use pyplot and NumPy in the same namespace without importing NumPy separately. If you have pylab imported, you don’t need to call NumPy and pyplot functions separately.

The role of the pyplot package is to facilitate Python programming through the Matplotlib library.

How to Create a Chart:

To create a chart, import pyplot as follows:

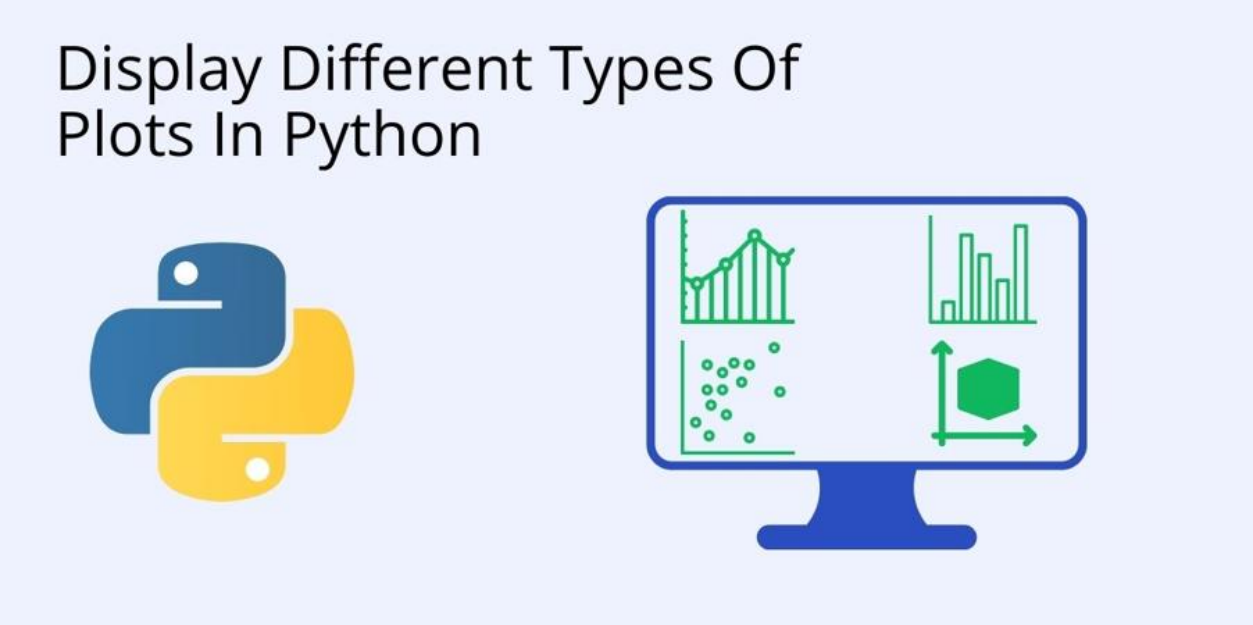
```python

import matplotlib.pyplot as plt
```

You can then plot data as shown:

```python
plt.plot([1, 2, 3, 4])
```

To display the plot, use `plt.show()`. The result is a chart displaying the data points. Depending on the platform, you may not need to call `show()` in some cases, particularly with iPython QtConsole.



Once your plot is prepared, provide definitions for the x and y axes. The blue line represents the data points in the absence of a legend, axis labels, or a title.

Utilizing Multiple Axes and Figures

In addition to employing pyplot commands for individual figures, you have the capability to manage numerous figures simultaneously within Matplotlib. Furthermore, you can take it a step further by incorporating new plots into each figure. Apart from utilizing multiple subplots, the subplot() function is at your disposal to generate multiple drawing regions within the primary figure.

The subplot() function also facilitates the selection of the specific subplot to concentrate your efforts on. Once chosen, any commands you issue will be executed on the currently selected subplot. A closer examination of the subplot() function reveals three integers, each serving a distinct purpose.

The initial integer defines the number of vertical divisions available in the figure. The second integer specifies the number of horizontal divisions within the figure. The third integer designates the subplot where your commands are applied.

Input
t = np.arange(0, 5, 0.1)
: y1 = np.sin(2 * np.pi * t)
: y2 = np.sin(2 * np.pi * t)
Input
plt.subplot(211)
: plt.plot(t, y1, 'b-.')
: plt.subplot(212)
: plt.plot(t, y2, 'r—')
The resulting plot should resemble the following:

In the subsequent example, we will establish vertical divisions for the aforementioned plots using the provided code.

Input

```
t = np.arange(0., 1., 0.05)
: y1 = np.sin(2 * np.pi * t)
: y2 = np.cos(2 * np.pi * t)
```

Input

```
plt.subplot(121)
: plt.plot(t, y1, 'b-.')
: plt.subplot(122)
: plt.plot(t, y2, 'r—')
```

The resulting plot should be as follows:

Incorporating Fresh Elements into Your Graph

Charts are intended to enhance the visual appeal of your data. To achieve this, it is crucial to select the appropriate chart type that effectively represents your data, as not all charts are suitable for all data types. Basic lines and markers may not suffice to create visually appealing charts. Consider incorporating additional elements into your chart to accomplish this goal.

How to insert text into a chart

The title() function allows you to introduce a detailed title into your chart, as demonstrated earlier. Moreover, you should be able to include axis labels using the xlabel() and ylabel() functions.

Keep in mind that when you introduce new functions like the axis label functions, they become part of the code string you are working with. Initially, you should add the axis labels to your chart because they provide context for the values assigned to each axis during data plotting. The example below illustrates this concept:

Input
plt.axis([0, 5, 0, 20])

: plt.title(‘My first plot’)
: plt.xlabel(‘Counting’)
: plt.ylabel(‘Square values’)
: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], ‘ro’)
The resulting plot should appear as follows:

You can perform basic edits on all the text you input to describe the plot. Basic editing encompasses modifications such as altering the font, adjusting font size, changing colors, or implementing any other tweaks necessary to enhance the chart’s appeal.

Following the preceding example, you can further customize the title as demonstrated below:

Input
plt.axis([0, 5, 0, 20])
: plt.title(‘My first plot’, fontsize=18, fontname=‘Comic Sans MS’)
: plt.xlabel(‘Counting’, color=‘black’)
: plt.ylabel(‘Square values’, color=‘black’)
: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], ‘ro’)

The Matplotlib functionality provides extensive options for refining your chart. For instance, you can introduce new text into the chart using the text() function, which is defined as text(x, y, s, fontdict=None, **kwargs).

In the above-mentioned function, the coordinates x and y denote the text’s placement within the chart. The variable s represents the text string to be

added at the specified location. While the fontdict() function specifies the font style for the new text, it is not obligatory. Once you have these parameters in place, you can introduce keywords into the code. Refer to the example below for clarification:

Input
plt.axis([0, 5, 0, 20])
: plt.title('My first plot', fontsize=20, fontname='Times New Roman') …:
plt.xlabel('Counting', color='gray')
: plt.ylabel('Square values', color='gray')
: plt.text(1, 1.4, 'First')
: plt.text(2, 4.4, 'Second')
: plt.text(3, 9.4, 'Third')
: plt.text(4, 16.4, 'Fourth')
: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
Your plot should now feature the following customizations:

Matplotlib is purpose-built for seamlessly incorporating mathematical expressions into your work using LaTeX expressions. When correctly formatted, the interpreter will recognize and convert these expressions into the corresponding mathematical graphics. This is particularly useful for introducing formulas, equations, or other specialized characters into your plot.

When composing LaTeX expressions, remember to prefix them with an 'r' to ensure the interpreter reads them as raw text.

Input
plt.axis([0, 5, 0, 20])

: plt.title('My first plot', fontsize=20, fontname='Times New Roman') …:
plt.xlabel('Counting', color='gray')

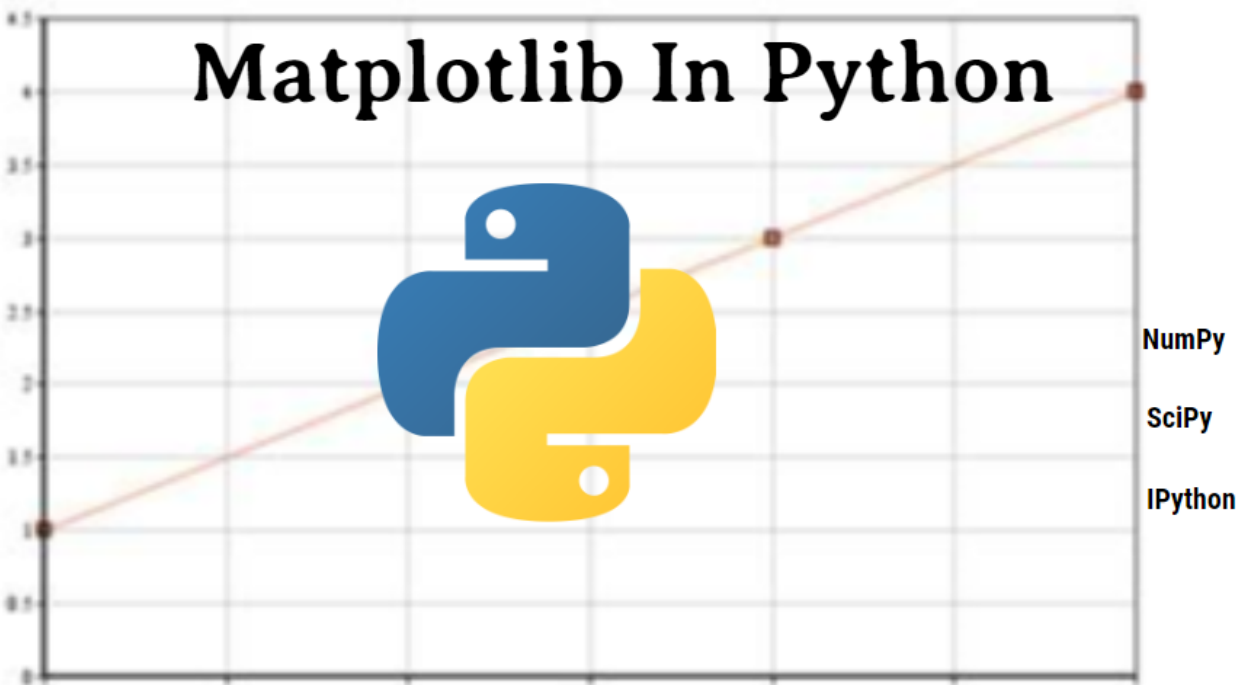
: plt.ylabel(‘Square values’, color=‘gray’)
: plt.text(1, 1.4, ‘First’)
: plt.text(2, 4.4, ‘Second’)
: plt.text(3, 9.4, ‘Third’)
: plt.text(4, 16.4, ‘Fourth’)
: plt.text(1.1, 12, r’$y = x^2$‘, fontsize=20, bbox={‘facecolor’: ‘yellow’, ‘alpha’: 0.2})
: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], ‘ro’)
Your plot should now include the expression ‘y = x^2’ within a yellow background, as depicted below:

Adding a Grid to Your Chart

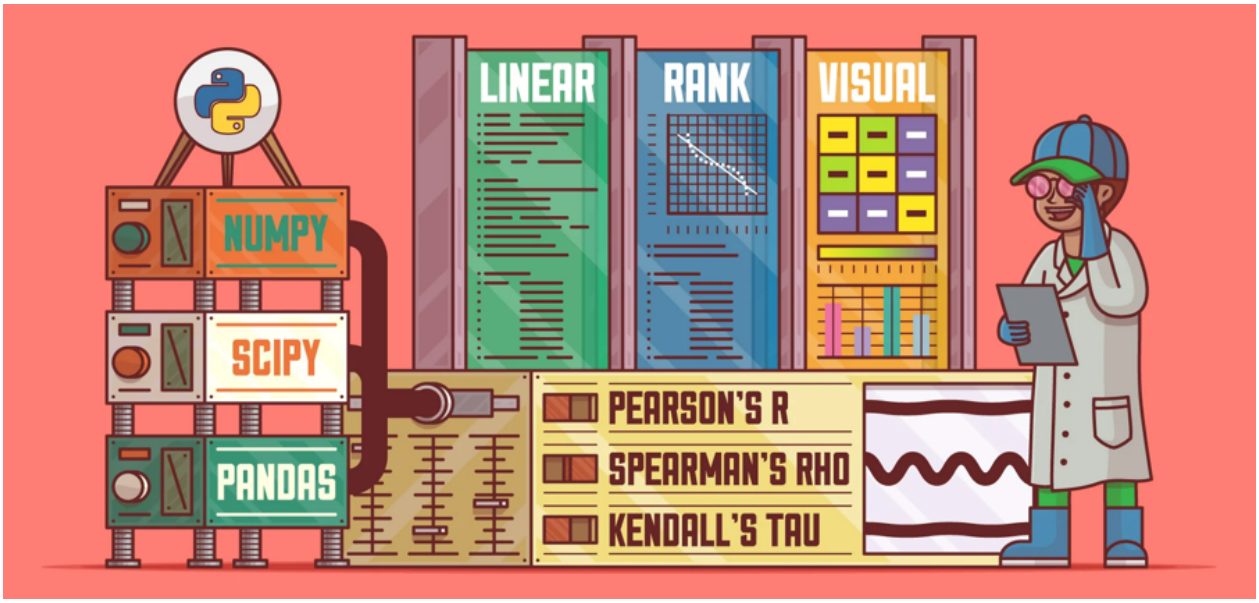
Frequently, you can access online tools to create charts that allow you to effortlessly toggle gridlines on or off. This capability is also available in Python. A grid is a valuable aid in your work, as it provides a visual reference for the plotted points on the chart. To introduce a grid, simply employ the grid() function and set it to ‘True’ as follows:

Input
plt.axis([0, 5, 0, 20])
: plt.title(‘My first plot’, fontsize=20, fontname=‘Times New Roman’) …:
plt.xlabel(‘Counting’, color=‘gray’)
: plt.ylabel(‘Square values’, color=‘gray’)
: plt.text(1, 1.4, ‘First’)
: plt.text(2, 4.4, ‘Second’)
: plt.text(3, 9.4, ‘Third’)

: plt.text(4, 16.4, ‘Fourth’)
: plt.text(1


Matplotlib In Python
NumPy
SciPy
IPython

Testing Hypotheses with SciPy



Hypothesis testing stands as a pivotal statistical technique applicable in data analysis, serving as a guide for analysts in making informed and statistically sound decisions regarding the datasets under scrutiny. In essence, hypothesis testing involves formulating an assumption about a particular aspect and subsequently employing data to ascertain the veracity of this assumption. For instance, you might postulate that the average age of students in your class is 25 years. From this conjecture, data is utilized to validate or refute this hypothesis.

These hypothetical suppositions are inherently theoretical, necessitating substantiation through statistical information. The determination of their accuracy or falsity hinges on the outcome of mathematical computations.

Essential Principles of Hypothesis Testing

Hypothesis testing ranks among the foremost statistical methodologies for data analysis. It revolves around the evaluation of mutually exclusive events related to a specific population under investigation and discerning which of these assertions aligns with the available data.

Following this analysis, one can then conclude the statistical significance of a particular finding after successfully passing the hypothesis test. Such tests pivot around the fundamental concepts of normalization and standardization, forming the bedrock upon which any hypothesis is constructed.

In statistical terms, normalization pertains to the procedure of evaluating and adjusting observed values to ensure they conform to a common scale, facilitating the application of other statistical measures like averaging.

In a normal distribution, variables assume the shape of a standard bell curve, and the graph illustrating this distribution is referred to as a normal curve. In a normal curve, three crucial parameters must be congruent: mode, median, and mean.

The formula for a normal distribution is as follows:
xnew = x – x min / (xmax – xmin)

Within a standard normal distribution, a normal curve is maintained, with a standard deviation of 1 and a mean of 0.

Null Hypothesis

During the course of hypothesis testing, you will inevitably encounter the null hypothesis. This represents the default stance, signifying the absence of a relationship between the variables in question. It can also denote the lack of association between two groups. The null hypothesis, therefore, is an assumption derived from basic knowledge of the subject matter and lacks statistical backing. For instance, you might assume that a car dealership sells 20 units per month, without any credible data to substantiate this assertion.

Alternative Hypothesis

An alternative hypothesis serves as the statement used to challenge the null hypothesis. Typically, the alternative hypothesis contradicts the presented null hypothesis. From this statement, a decision must be made regarding the acceptance or rejection of the null hypothesis based on the likelihood of the alternative hypothesis being valid.

The significance level denotes the extent to which one accepts or rejects the null hypothesis. In logical terms, it is impossible to affirm or negate any hypothesis with absolute certainty. Therefore, the significance level is often set at 5%, represented by the symbol alpha (α), typically calculated as 5% or 0.05. Consequently, the output under consideration should provide a 95% confidence level for evaluation.

Errors

Two types of errors may arise during hypothesis testing: type I and type II errors. A type I error entails rejecting the null hypothesis even when it is true, represented by alpha (α). In a normal curve, this critical region is commonly known as the alpha region.

On the other hand, a type II error entails accepting the null hypothesis even when it is false, represented by the beta (ß) symbol. In a normal curve, this acceptance region is referred to as the beta region.

The results obtained from a hypothesis test and the subsequent decision to accept or reject those results are not immutable. Decisions must be made thoughtfully in light of the evidence provided by the hypothesis test, bearing in mind that the evidence may not always be robust enough to guarantee the correct decision. This can lead to the occurrence of one of the aforementioned errors.

To illustrate, consider the diagram below:

[Insert diagram description here]

From our illustration, the red sections represent type I errors, suggesting that the alternative hypothesis aligns with the null hypothesis when a two-sided test is employed for this data, at a 95% confidence level.

The blue section of this plot signifies a type II error, occurring when the null hypothesis does not correspond to the alternative hypothesis. In hypothesis testing, a t-test is a vital tool for discerning disparities between population and sample averages.

Procedure for Hypothesis Testing

Before embarking on hypothesis testing, it is essential to adhere to specific protocols to ensure the reliability of your findings.

Firstly, you must formulate both a null hypothesis (usually denoted as H0) and an alternative hypothesis (represented as H1). These statements serve as the focal points for your investigation. The primary objective in testing the null hypothesis is to determine whether it is incorrect. In the alternative hypothesis, you articulate a statement that contradicts the null hypothesis.

The next step is to establish a decision-making criterion based on your chosen significance level. This criterion will guide you in assessing the validity of the null or alternative hypothesis. It's important to note that the same hypothesis may be accepted at a 4% probability level while being rejected at a 5% significance level. By defining this criterion beforehand, you establish a framework for your analysis.

Subsequently, you need to evaluate the probabilities associated with the available data, which is achieved through a probability test statistic. This statistic helps quantify the likelihood of a particular event occurring. A higher probability indicates a greater chance that the null hypothesis is true, based on the available evidence.

Finally, you must make a decision based on the obtained results. In this decision-making process, you compare the results against the predetermined significance level. If the null hypothesis's probability is lower than the significance level, you reject it.

It's crucial to be aware of the potential for accepting an incorrect result when working with a sample population. Since the sample represents only a random subset of the overall population, including data from the entire population may significantly alter the results and lead to the alternative hypothesis being true.

In summary, there are four possible outcomes when evaluating the null hypothesis:

You may correctly uphold the null hypothesis.
You may encounter a Type II error by incorrectly maintaining an invalid null hypothesis.
You may correctly reject the null hypothesis.
You may encounter a Type I error by incorrectly rejecting a valid null hypothesis.

Statistical hypothesis testing is a critical component of data analysis as it helps determine whether the data aligns with a predefined norm. The nature of any deviation can provide valuable insights into the data.

In hypothesis testing, all assumptions start with the null hypothesis, which posits no relationship between the variables being studied. The specific form of the null hypothesis depends on the type of test being conducted. For instance, if you are testing whether two groups are dissimilar, the null hypothesis would state that the two groups are similar.

The primary goal of a hypothesis test is to ascertain whether the null hypothesis remains valid when analyzing a particular dataset. If, after analysis, there is insufficient evidence to refute the null hypothesis, it is accepted. Similar to the null hypothesis, the alternative hypothesis is tailored to the specific data being analyzed.

Once the alternative and null hypotheses are defined, you can establish a significance level, a threshold that guides whether to accept or reject the results.

One-Sample T-Test

The objective of this test is to determine whether the mean of a sample population closely matches the mean of the entire population within the dataset under investigation. To illustrate this concept, we will use fictional age data from registered voters in two different regions, x and y, and evaluate whether there is a significant difference in their average ages.

Please execute the following code:

```python
%matplotlib inline
import numpy as np
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import math

np.random.seed(6)
x_age1 = stats.poisson.rvs(loc=18, mu=35, size=150000)
x_age2 = stats.poisson.rvs(loc=18, mu=10, size=100000)
x_age = np.concatenate((x_age1, x_age2))
y_age1 = stats.poisson.rvs(loc=18, mu=30, size=30)
y_age2 = stats.poisson.rvs(loc=18, mu=10, size=20)
y_age = np.concatenate((y_age1, y_age2))

print(x_age.mean())

print(y_age.mean())
```

From the distribution above, we can conduct a t-test to assess the validity of our hypothesis at a 95% confidence level.

For this purpose, we will utilize the `stats.ttest_1samp()` function as shown below:

```python
stats.ttest_1samp(a=y_age, popmean=x_age.mean())
```

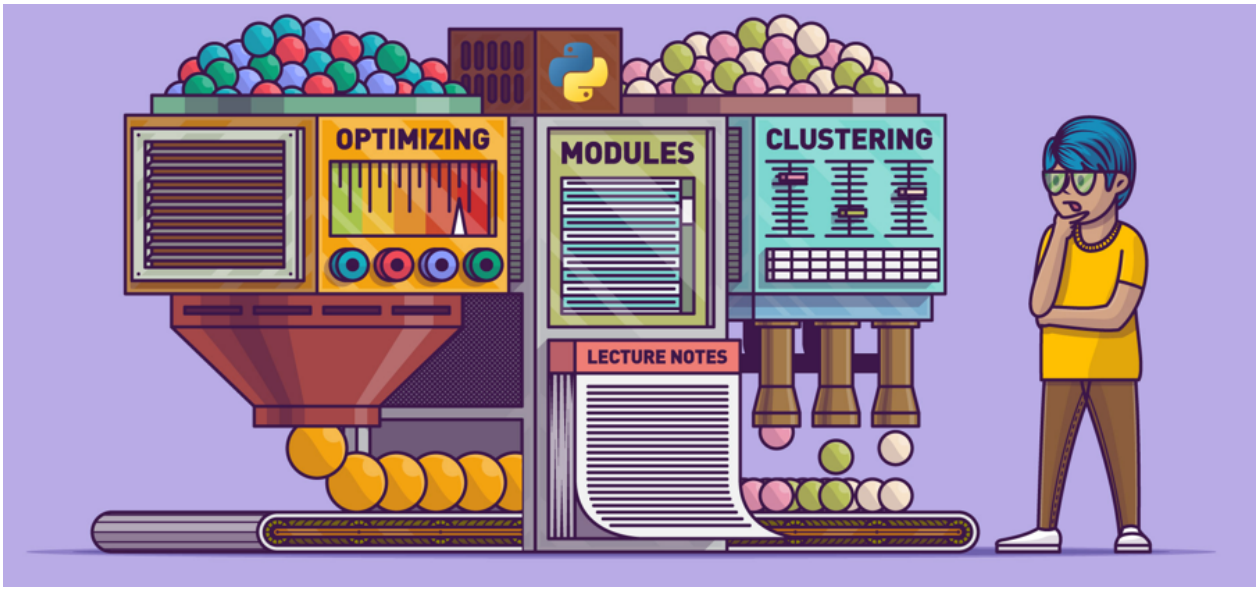
The output will provide a statistic value, which signifies the deviation of the mean from the applied null hypothesis. If the t-test result falls outside the t-distribution quantile corresponding to your confidence level, it is advisable to reject the hypothesis.

To determine this quantile, you can use the `stats.t.ppf()` function, like so:

```python
stats.t.ppf(q=0.025, df=49) # For the desired confidence level and degrees of freedom
```

From this analysis, if the p-value is lower than the significance level, the appropriate action is to reject the null hypothesis.

By increasing the confidence level, you expand the confidence interval, making it more likely to capture the true population mean. Consequently, if the significance level is lower than the p-value, you accept the null hypothesis.



Two-Sample T-Test Explained

The two-sample t-test serves as a means to ascertain the similarity or dissimilarity between two independent data samples. Within this test, our null hypothesis posits that the means of the two sample groups are alike.

Distinguishing it from the one-sample t-test, where we compare a sample against the entire population, the two-sample t-test instead compares one sample against another, bypassing the population.

For this test, we employ the stats.ttest_ind() function. We create a separate set of sample data for County M, which we subsequently compare against sample voter registration data for County Y, as outlined in the one-sample t-tests mentioned earlier.

Input

```python
np.random.seed(12)

m_age1 = stats.poisson.rvs(loc=18, mu=33, size=30)
m_age2 = stats.poisson.rvs(loc=18, mu=13, size=20)
m_age = np.concatenate((m_age1, m_age2))
print(m_age.mean())
```

```
Output
```
42.8
```

Input
```python
stats.ttest_ind(a=y_age, b=m_age, equal_var=False) # samples share similar variance
```
Output
```
Ttest_indResult(statistic=-1.7084, pvalue=0.0907)
```

Based on the derived p-value, there is only a 9% chance that when comparing the population means of the two samples, we can determine their similarity. Should we employ a 95% confidence interval, the null hypothesis will prevail, as the 5% significance level exceeds the data's p-value.

Understanding the Paired T-Test

The previously discussed tests considered data from two separate population sample groups. However, there are instances where you need to analyze sample data from the same group but at different time intervals. This is essential for comprehending changes within the study group.

For example, teachers might want to gauge whether students have improved their knowledge of a subject by assessing their performance before and after an exercise. In such cases, it is prudent to employ a paired t-test to
```

determine the similarity of sample performance data within the same group at distinct intervals.

For this study, we utilize the scipy function stats.ttest_rel(). First, let's generate sample performance results for this test.

Input

```python
np.random.seed(11)
before_exercise = stats.norm.rvs(scale=30, loc=250, size=100)
after_exercise = before_exercise + stats.norm.rvs(scale=5, loc=-1.25, size=100)
performance_df = pd.DataFrame({"performance_before": before_exercise,
"performance_after": after_exercise,
"performance_change": after_exercise - before_exercise})
performance_df.describe()
```

Output

```
performance_after performance_before performance_change
count 100.000000 100.000000 100.000000
mean 249.115171 250.345546 -1.230375

std 28.422183 28.132539 4.783696
min 165.913930 170.400443 -11.495286
25% 229.148236 230.421042 -4.046211
50% 251.134089 250.830805 -1.413463
75% 268.927258 270.637145 1.738673
max 316.720357 314.700233 9.759282
```

Based on the summary above, it’s evident that, on average, students lost 1.23 points after the exercise. We can now perform a paired t-test to determine whether this information is statistically significant at a 95% confidence level.

Input

```python
stats.ttest_rel(a=before_exercise, b=after_exercise)
```

Output

```
Ttest_relResult(statistic=2.5720, pvalue=0.01160)
```

From the p-value results above, we can infer that there’s only a 1% chance of observing a significant difference between the two population samples.

Utilizing SciPy

As a data analyst, you’ll encounter diverse data from various studies throughout your work. Some of this data may be technical, while others may be scientific, depending on the research objectives and data collection methods employed. Managing and performing mathematical computations on large datasets can be challenging. This is where large supercomputers come into play, specifically designed for such tasks.

One of the most convenient ways to handle technical data is through SciPy, a Python library designed to manage such data effectively. A noteworthy aspect of SciPy is its open-source nature, allowing you to use it without incurring any costs. SciPy offers several data science-relevant features that you’ll find valuable.

Installing SciPy

The installation process depends on your operating system. The following instructions will guide you through installing SciPy in Python based on your device’s requirements.

You can install SciPy using pip, a widely recognized package manager in most common operating systems. Before using pip for installation, ensure that Python is installed on your computer. Once Python is set up, execute the following command:

For Windows operating systems:

```
Python -m pip install —user numpy scipy
```

This command installs SciPy in the user directory, rather than the system directories, denoted by the `—user` flag.

For Mac systems, use the following commands:

```
sudo port install py35-scipy py35-numpy
```

The `sudo` command allows you to install and run programs with elevated privileges.

For Linux operating systems, you can install SciPy with the following commands:

```
sudo apt-get install python-scipy python-numpy
```

Once SciPy is installed, you can proceed to the next steps.

SciPy Modules

In your role as a data analyst, you'll encounter various programs and tools for performing scientific and mathematical operations in Python. SciPy offers a wide range of modules that can assist you in performing tasks, from simple to complex. Here are some of the packages you'll be using:

Package Function
- Special function scipy.special
- Spatial data structures and algorithms scipy.spatial
- Sparse scipy.sparse
- Statistics scipy.stats
- Signal processing scipy.signal
- Integration scipy.integrate
- Interpolation scipy.interpolate
- Input/output scipy.io

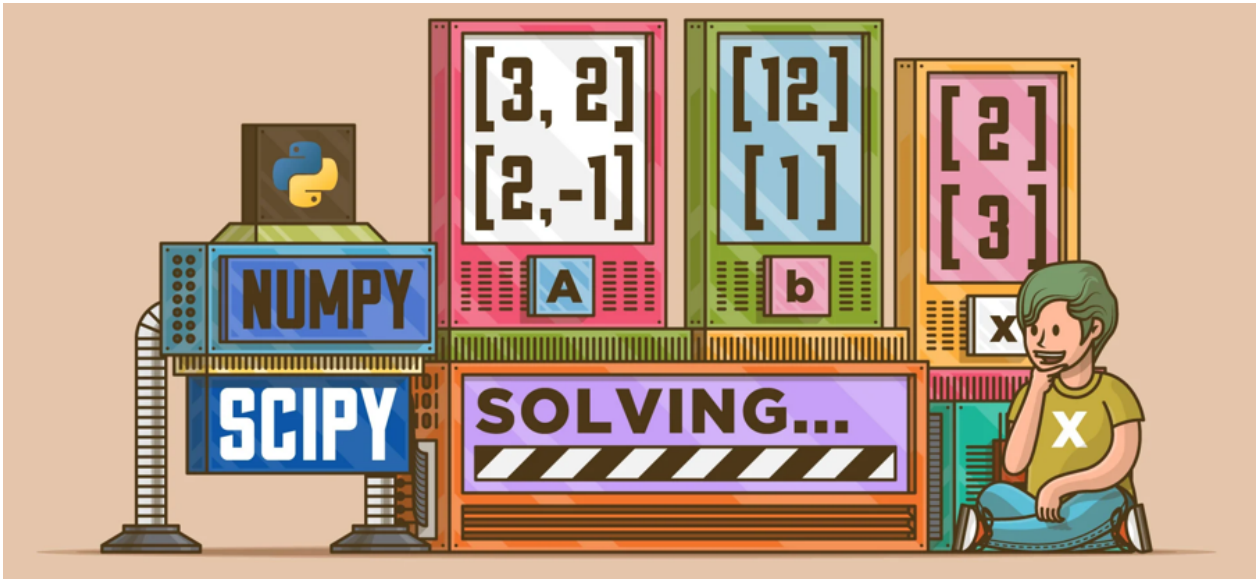
- Multidimensional image processing scipy.ndimage
- Linear algebra scipy.linalg
- Optimization scipy.optimize
- Fast Fourier transformation scipy.fftpack

To begin working with SciPy, you need to import it. The import procedure remains consistent for all sub-packages. Simply replace the package `signal` with the desired module you intend to use. The import instructions are as follows:

```python
import numpy as np
from scipy import signal
```

Integration with SciPy

The `scipy.integrate` sub-package is crucial for performing numerical integration computations. The functions used in this package are outlined in the section above.



For general-purpose integration, also known as a single integral, where your dataset has only one variable between two endpoints, you can use the `quad` function. For instance, if you have information about data integrals represented by a function of 12x between two points, 0 and 1, the single integration would appear as

9

Data Mining in Python

There are numerous analytical choices available when working with any given dataset. The process of extracting predictive insights from these datasets is referred to as data mining. It stands as one of the most demanding tasks for any data scientist, yet it remains essential. Raw data can be interpreted in various ways, making it crucial to derive accurate deductions from it. Each dataset harbors distinct fragments of information that, with the right analytical approach, can be harnessed to make precise predictive decisions.

In the realm of Python, a plethora of tools exists for data mining purposes. This intricate procedure encompasses activities such as data cleaning and organization, as discussed earlier in this book. It underscores the importance of validating the credibility and integrity of data before its utilization.

At its core, data mining involves the study of data and the construction of models that enable accurate generalizations about a subject. This concept is closely intertwined with numerous other machine learning processes reliant on predictive analysis. These data models can respond to new inputs based on previous data, facilitating appropriate actions. For instance, if a system holds financial data about your US accounts, any transaction outside that jurisdiction would be met with suspicion and flagged for further scrutiny and analysis.

Data mining proves invaluable in numerous scenarios, ranging from social media studies to businesses investigating and dissecting consumer preferences. It offers vital insights that can inform significant decision-making.

Methods of Data Mining

The creation of predictive models is feasible from various datasets, each crafted using a distinct methodology. Here are some common techniques employed in data mining:

1. Regression Analysis

This entails the examination and evaluation of relationships between different variables. The focus lies on understanding these relationships while simultaneously minimizing errors.

2. Cluster Analysis

In this process, analysts delve into distinct data groups, aiming to comprehend them based on their unique characteristics. Clusters are formed according to specific attributes, and members of each cluster are expected to exhibit similar behaviors.

3. Data Classification

Data classification involves categorizing data into specific groups based on predefined criteria. Once the categories are established, data meeting these criteria are allocated to their respective categories. An example of this is spam email filtering.

4. Analyzing Outliers

This process entails the investigation of outliers to uncover the reasons behind their existence. Outliers typically surface in a dataset when certain data deviates from the expected pattern, failing to align with a predetermined trend.

5. Correlation and Association Analysis

The study of data is conducted to ascertain relationships between different variables, particularly those with less evident connections. For example, the famous Walmart beer-diaper case study revealed an unexpected correlation between beer and diaper sales on Friday evenings. Further examination revealed that this apparent correlation was due to the fact that most men buying diapers on Fridays were young or new fathers who decided to pick up some beer as well.

Building a Regression Model

Before constructing a regression model, it’s essential to determine the specific problem you aim to address, as each problem necessitates a tailored approach. In the following examples, we will utilize the King County House Sales dataset to explore the relationship between various variables and house prices in King County.

To get started, ensure you have Jupyter installed on your device. Jupyter is an iPython processor that is part of the Anaconda distribution, which simplifies the process by including Jupyter, Python, and various libraries useful for data analysis and scientific computing.

Download the latest Anaconda version for Python and follow the installation instructions. Once installed, run the following command:

```
jupyter notebook
```

This will initiate the notebook server, with the default web application URL usually set to http://localhost:8888. The notebook dashboard will display information about the directories, files, and notebooks running on your server.

For experienced Python programmers, Jupyter can also be installed via pip instead of using Anaconda. Before proceeding, ensure you have the latest pip version installed, as earlier versions may encounter issues with certain dependencies you'll be using in the future.

To install the notebook, use the following command:

```
pip3 install jupyter
```

Additionally, you'll need Pandas for data restructuring and cleaning, as it excels at importing data from various file formats and organizing it to suit your needs. Import the necessary libraries:

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
import seaborn as sns
```

```
from matplotlib import rcParams
%matplotlib inline
```

To begin with, we've incorporated Matplotlib, NumPy, and SciPy in the code snippet above. These are valuable tools for tasks such as data visualization, scientific calculations, and statistical analyses in data exploration.

The initial step when dealing with any dataset is to scrutinize it thoroughly and assess whether it necessitates cleaning and the extent of such cleaning. In the provided code, we load a dataset from the specified file path and display its initial records as follows:

```python
import pandas as pd

df = pd.read_csv('/Users/Admin/Desktop/kc_house_data.csv')
df.head()
```

The expected output should resemble this structure:

```
id date price bedrooms bathrooms sqft_living sqft_lot
0 7129300520 20141013T000000 221900.0 3 1.00 1180 5650
1 6414100192 20141209T000000 538000.0 3 2.25 2570 7242
2 5631500400 20150225T000000 180000.0 2 1.00 770 10000
3 2487200875 20141209T000000 604000.0 4 3.00 1960 5000
4 1954400510 20150218T000000 510000.0 3 2.00 1680 8080
```

Subsequently, we proceed to identify whether any null values exist in the dataset using the following code:

```python
df.isnull().any()
```

The expected output should appear like this:

```
id False
date False
price False
bedrooms False
bathrooms False
sqft_living False
sqft_lot False
```

dtype: bool
```

Next, we determine the data types of each variable in the dataset to ascertain whether we are working with numerical data. This information is crucial for subsequent analysis:

```python
df.dtypes
```

The output should resemble this:

```
id int64
date object
price float64
bedrooms int64
bathrooms float64
sqft_living int64
sqft_lot int64
dtype: object
```

A fundamental step is ensuring proper data processing in Pandas, especially when dealing with datasets of various data types within columns. For regression analysis, it’s imperative that all data within a column is compatible. Often, you’ll encounter data that is not well-structured, underscoring the importance of understanding these functions.
```

Subsequently, we examine the shape of the data, providing an initial sense of its distribution and credibility. Some data may be corrupted, so it's prudent to validate its usability.

To view all the variables within the dataset, you can use the `df.describe()` function. Following that, you can create histograms for all variables using `plt.pyplot.hist()`:

```python
import matplotlib.pyplot as plt

df.describe()
fig = plt.figure(figsize=(12, 6))
sqft = fig.add_subplot(121)
cost = fig.add_subplot(122)
sqft.hist(df.sqft_living, bins=80)
sqft.set_xlabel('Ft^2')
sqft.set_title("Histogram of House Square Footage")
cost.hist(df.price, bins=80)
cost.set_xlabel('Price ($)')
cost.set_title("Histogram of Housing Prices")
plt.show()
```

This code generates two histogram distributions: one for housing prices and another for house square footage. The data distribution appears right-skewed, which sets the stage for regression analysis.

To perform a linear regression analysis, we import the `statsmodels` library and employ the least squares method:

```python

import statsmodels.api as sm
from statsmodels.formula.api import ols

m = ols('price ~ sqft_living', df).fit()
print(m.summary())
```

The output provides essential information about the regression model, including standard error, correlation coefficients, and t-statistics. In this case, a significant relationship between the two variables is indicated by a high t-value (144.920), while the P>|t| value is 0%, suggesting a minimal chance of the relationship occurring by chance or statistical variation.

Further enhancing the model involves introducing additional independent variables:

```python
m = ols('price ~ sqft_living + bedrooms + grade + condition', df).fit()
print(m.summary())
```

This addition of variables increases the R-squared value from 0.493 to 0.555, indicating a more comprehensive perspective of the data.

Summarily, a regression analysis summary is invaluable for assessing the accuracy of the model and dataset. It provides crucial insights into the relationships between variables and their statistical significance.

Creating Cluster Models

Similar to our approach when starting with a regression model, the initial step involves identifying the problem you aim to address through the development of a cluster model. Clustering revolves around the concept of categorizing data sets based on predefined criteria.

Within the dataset, the inherent nature of the data or objects may not be immediately apparent. Consequently, the responsibility falls on your shoulders to scrutinize the data and establish clusters that share common attributes.

In this particular analysis, we will leverage the Old Faithful geyser dataset, accessible on GitHub. This dataset encompasses only two variables: the duration of geyser eruptions in minutes and the time intervals between eruptions. When working with datasets featuring just two variables, the optimal choice is often the employment of a k-means clustering approach.

To embark on this analysis, it is imperative to install Scikit-Learn, a renowned Python module for data mining and machine learning. Import the essential modules into your notebook as delineated below:

```python
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import sklearn
from sklearn import cluster

%matplotlib inline
```

```
faithful = pd.read_csv('/Users/Admin/Desktop/faithful.csv')
faithful.head()
```

This dataset is stored locally on your desktop. Subsequently, you will verify whether the data contains any missing values and handle them if necessary. However, the dataset in use is devoid of any missing values, negating the need for such treatment.

Assign meaningful column names to the dataset:

```python
faithful.columns = ['eruptions', 'waiting']
```

Visualize the data by generating a scatter plot:

```python
plt.scatter(faithful.eruptions, faithful.waiting)
plt.title('Old Faithful Data Scatterplot')
plt.xlabel('Length of eruption (minutes)')
plt.ylabel('Time between eruptions (minutes)')
```

At this juncture, your data plot should unveil two distinct clusters representing the two variables. Nevertheless, distinguishing between them can be challenging. To address this, the introduction of visualization functions becomes imperative:

```python
faith = np.array(faithful)
```

```
k = 2
kmeans = cluster.KMeans(n_clusters=k)
kmeans.fit(faith)
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
```

Considering the presence of only two variables, we opt for k=2. The ‘kmeans’ function in this context yields an output from your cluster module. To differentiate between the two scatter variables, it is crucial to employ distinct scatter plot colors, as illustrated in the following code snippet:

```python
# Select only observations with cluster label == i
ds = faith[np.where(labels==i)]
# Plotting observations
plt.plot(ds[:,0],ds[:,1],’o’, markersize=7)
# Plotting centroids
lines = plt.plot(centroids[i,0],centroids[i,1],’kx’)
# Enlarge the centroid x’s
plt.setp(lines,ms=15.0)
plt.setp(lines,mew=4.0)
plt.show()
```

Your output should exhibit a clear differentiation of the clusters, with each represented by distinct colors. Should you introduce additional variables into your dataset, it becomes possible to employ diverse colors for various clusters.

Numerous data mining techniques are available for acquisition, each serving as a valuable asset when analyzing diverse data types. Familiarity with the appropriate analytical methods for distinct data categories is crucial, as the suitability of analytical approaches varies significantly depending on the specific dataset under examination.

10

Conclusion

Data analysis holds a significant role in various facets of contemporary life. From the moment you awaken, you engage with data on multiple levels. Many pivotal decisions rely on data analytics, and companies depend on this data to achieve their diverse objectives. As the global population continues to expand, companies must discover ways to both maintain customer satisfaction and meet their business targets.

In the competitive landscape of the business world, keeping customers content is no easy task. Competitors continuously vie for each other's customers, and those who succeed face the challenge of retaining these customers to prevent them from reverting to their former business associates. This is precisely where data analysis proves invaluable.

To gain a deeper understanding of their customers, companies rely on data collection at every customer touchpoint. This data serves various purposes, enabling companies to gain insights into their customers' preferences and needs. By segmenting their customer base based on these insights, companies can provide more tailored services, aiming to prolong customer satisfaction.

Yet, data analytics extends beyond customer-centric profit motives; it also plays a pivotal role in governance. Governments are the largest consumers of data worldwide, collecting information on citizens,

businesses, and other entities they interact with. This data is crucial for numerous purposes, including effective resource allocation and national security.

For planning purposes, governments require precise population data to allocate resources equitably. Achieving such equitable resource distribution is impossible without thorough data analysis. Furthermore, data plays a critical role in national security, as governments must maintain databases for individuals with high profiles requiring special security measures, as well as potential threats that necessitate continuous monitoring.

Data analysis encompasses more than just corporate and government decision-making; it is a field of great challenge and excitement for programmers. Data, when unaltered, provides an undeniable truth, making data analysis skills essential for addressing complex challenges and solving problems in the field. The way data is handled can have a significant impact, more significant than one might initially realize.

Numerous tools are available for data analysis, with many individuals relying on Microsoft Excel. However, Python offers a solution to Excel's limitations, making it a valuable programming language to learn. Python's high-level nature, with syntax closely resembling everyday language, facilitates the mastery of its concepts.

Experienced programmers move beyond mastering Python's basics and utilize it to solve real-world problems through data analysis. Identifying the underlying issue and devising a data-driven solution are typically the first steps in tackling these challenges.

This book follows a series of comprehensive guides introducing data analysis using Python. Important concepts have been reiterated throughout the series to reinforce fundamental knowledge. Understanding Python libraries is crucial, as they are instrumental in becoming an expert data analyst using Python.

As you engage with data, you'll appreciate the significance of data cleansing to ensure the accuracy of your analyses. You will learn the process and build upon it to maintain the quality of your work. Another challenge faced by many organizations is safeguarding data integrity; implementing procedures to ensure consistent use of clean data is essential.

In a world where data is central to many activities, data is generated and stored in vast quantities daily through automated systems. Learning data analysis with Python equips you to process data, extract meaningful insights, and draw significant conclusions. One area where these skills prove invaluable is forecasting, where predictive models can be developed to assist organizations in meeting their objectives.

The quality of data, the data modeling techniques employed, and the dataset used for analysis are paramount to the success of a predictive model. Beyond data processing, another critical aspect of data analysis is data visualization. Effective visualization is about presenting data in a manner that allows an audience to grasp it immediately. Learning how to create various visualizations aids in gaining a preliminary understanding of the data's nature.

Upon completing data analysis, you should have a comprehensive data model supplemented with visual representations that facilitate predictions and responses before proceeding to the testing phase. Data analysis is a highly sought-after skill in various fields, and understanding how to handle data appropriately, when to do so, and why it matters is a skill that should not be underestimated. Through this knowledge, you can construct and test hypotheses, ultimately enhancing your understanding of systems.