Introduction

This book is aimed at applied researchers, graduate students, and advanced undergraduate students in psychology and related behavioral sciences—education, neuroscience, health, political science, anthropology, and sociology, among others—who are trained in classical statistical methods but have limited computer science or programming experience. Whether such readers seek to collaborate with data scientists on multidisciplinary research teams, analyze increasingly available very large data sets, or stay current with the latest analytical techniques, this book will serve as a gateway or stepping stone to data science. It builds on readers' statistical strengths and addresses common gaps in traditional training that can slow the transition to modern data analytics.

The presentation in this book assumes familiarity with basic statistical significance tests—including the *t*-test, analysis of variance (ANOVA), and the *F*-test—as well as the fundamentals of linear regression in both its bivariate and multiple-predictor forms. This level of background typically corresponds to advanced undergraduate coursework in statistics or at least one such course at the master's level. For readers with more limited quantitative backgrounds or for whom their statistics knowledge is rusty, three freely available primers in core concepts and skills can be accessed from the Guilford website for this book at www.guilford.com/kline3-materials.

These primers cover:

1. Bivariate and multiple regression fundamentals.
2. Standard errors and classical statistical significance tests.
3. Essentials of data screening and preparation.

On the same website, readers can also access chapter appendices for all detailed analysis examples in the book. The appendices include annotated syntax in R—a freely available computing environment widely used in data science, psychology, and other disciplines—with step-by-step explanations of how the code works. Examples of output

are also featured in many appendices. Readers can also download syntax, data, and output files for all examples, which allows readers to reproduce these analyses on their own computers. These same files are also available on the publisher's resource site at www.guilford.com/kline3-materials. These example analyses will be updated from time to time, keeping them current as newer versions of R become available.

Scope of the Book

This book is organized into four parts. Introduced in the first part are fundamental concepts in classical statistics and data science with the focus on their commonalities and differences. The second part leverages (and hopefully improves on) readers' understanding of fundamental concepts in classical statistics based on frequentist inference and surveys Bayesian inference as an alternative. The third part deals with programming concepts, data visualization, and challenges and strategies in analyzing big data sets. Machine learning techniques are featured in the last part with numerous examples including ones with real data sets that can be accessed from the book's support site (www.guilford.com/kline3-materials). While numerous specialized algorithms lie beyond the scope of this introductory text, the selected content provides a practical foundation for researchers seeking to integrate data science into their work.

Nevertheless, an appreciation of the broader context in which this book's focus on machine learning is situated within the larger domain of data science is essential. Presented in Figure I.1 is a Venn diagram illustrating the relations among machine learning and other major data science techniques. Machine learning, a subset of artificial intelligence, enables computational systems to learn from data and improve task performance without explicit programming. It occupies the intersection of statistics, optimization, and computer science and underpins a range of applications, from network security monitoring to medical diagnostics. Within machine learning, neural networks consist of interconnected layers of processing units (nodes) that iteratively adjust connection weights during training to capture complex data patterns. These architectures are based on the brain's neural structure and employ activation functions to introduce nonlinearity. Deep learning refers specifically to neural networks with multiple hidden layers (beyond the single hidden layer of shallower networks) that automatically learn hierarchical feature representations from data. This multilayer structure enables the modeling of increasingly abstract patterns and distinguishes deep learning from more basic neural network designs.

Natural language processing represents the subfield of artificial intelligence and machine learning dedicated to developing algorithms that acquire hierarchical representations of human language—see Figure I.1. These representations facilitate tasks of language comprehension, interpretation, generation, and interactive communication. The intersection of natural language processing and deep learning gives rise to large language models. These models employ neural networks with extensive layered architectures to capture patterns in syntax, semantics, and factual knowledge, thereby enabling the prediction

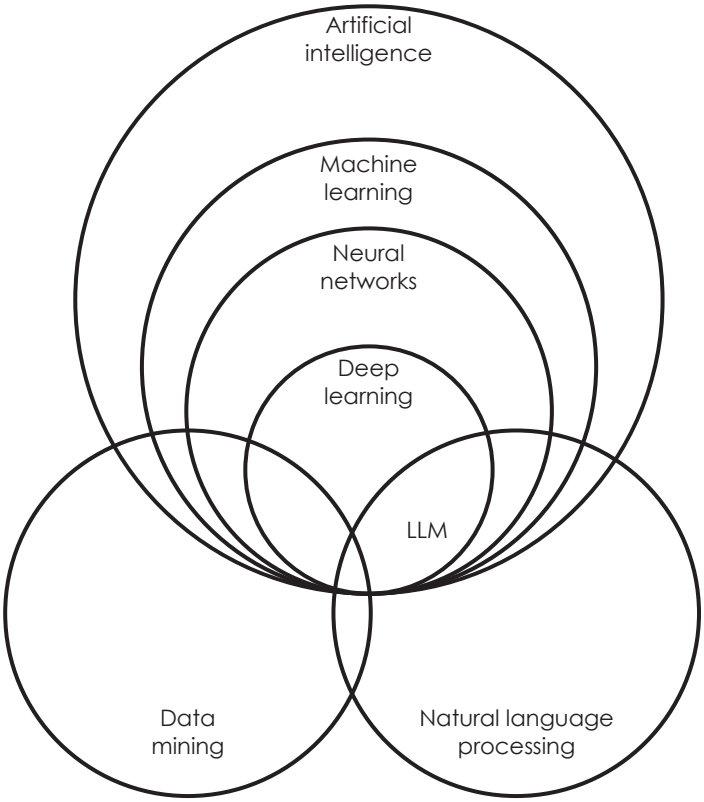

FIGURE I.1. Venn diagram for machine learning and other major areas of data science. LLM, large language model.

and generation of coherent text. Interaction with conversational systems based on artificial intelligence, such as ChatGPT, exemplifies the practical deployment of large language models. Effective training of such models demands large corpora of textual data and considerable computational capacity, often utilizing specialized hardware accelerators. Although Figure I.1 represents just textual data, comparable domains within data science address the analysis and learning from images, video, audio, and sensor signals. These applications similarly engage artificial intelligence, machine learning, and neural network techniques, including deep learning. Consequently, the domain of data science extends beyond the boundaries depicted in the figure.

Data mining employs artificial intelligence, especially machine learning algorithms and neural network models, to detect patterns within very large data sets (see Figure I.1). Data mining's core tasks include data extraction, classification, association rule learning, and anomaly detection. Inputs span structured, tabular formats (e.g., relational database tables) as well as semi-structured or unstructured sources, such as sensors that collect data from their environment in real time (e.g., a wearable heart monitor) or dynamic social

media interactions. Whereas many machine learning applications in data science are concerned with prediction or classification using extant data, data mining emphasizes a more exploratory, human-guided process to uncover and prepare data for analysis.

Entire books have been written about topics introduced to this point. This text emphasizes machine learning methods that are probably the data science techniques most pertinent to researchers and advanced students in the behavioral sciences. Although journal publications featuring machine learning applications in psychology and related fields remain relatively novel, their prevalence is rising. For instance, my first invitation to review a machine learning manuscript for a mainstream psychology journal occurred only 4 years ago. Since then, comparable studies have appeared in outlets spanning education, political science, and psychiatry (Chandler et al., 2020; Grimmer et al., 2021; Hilbert et al., 2021). Machine learning methods offer a range of advantages over traditional regression and classification techniques in classical statistics, as well as introduce specific challenges. Both aspects are examined in this presentation.

Plan of the Book

Part I comprises three chapters. Chapter 1 extends the discussion of data science beyond the Venn diagram presented in Figure I.1. Chapter 2 reviews fundamental concepts and challenges related to inference in classical statistics, and Chapter 3 contrasts the philosophical foundations and analytical approaches of classical statistics with those of data science. Surveyed in Chapters 2 and 3 are ethical codes published by the American Statistical Association, the Data Science Association, and the Association of Data Scientists. While these codes share core principles governing professional conduct, they exhibit nuanced distinctions that reflect practice contexts ranging from commercial enterprises to university-based research centers.

Chapters 4–6 of Part II aim to deepen comprehension of core classical statistics concepts: interval estimation (Chapter 4), statistical significance testing (Chapter 5), and regression analysis issues that also bear on machine learning research (Chapter 6). Given the prevalence of misinterpretations surrounding confidence intervals, *p*-values, and regression results among practitioners of classical statistics, these chapters prioritize the correction of such misunderstandings before expanding the reader's analytical repertoires to include data science techniques. A further advantage lies in enhancing the effective application of classical statistical methods within research domains already familiar to the readers.

Chapter 7 provides a primer on Bayesian inference for readers whose training has centered on frequentist methods in classical statistics. Bayesian approaches to interval estimation and significance testing present notable advantages and are increasingly adopted in psychology and related behavioral sciences, although frequentist inference remains the norm. In contrast, Bayesian statistics have long been prominent in epidemiology, finance, econometrics, computer science, and data science. Data science encompasses both

Bayesian and frequentist techniques, yet Bayesian methods often assume greater prominence, particularly in fields that depend on continuously updated data. I should emphasize that Bayesian procedures are not panaceas: They carry their own limitations and are susceptible to misuse, just as frequentist approaches can be misapplied. This chapter balances an introduction to expanded research possibilities with a discussion of potential pitfalls.

Part III (Chapters 8–11) addresses programming and data management, with a focus on large-scale data sets. Chapter 8 introduces programming fundamentals for researchers without formal training in software development. It outlines the benefits of even modest coding proficiency across research disciplines and recommends practical pathways for skill acquisition. The chapter underscores the critical reality that scientific domain expertise remains the primary asset that behavioral scientists contribute to data science. Now, certain data science tasks, such as distributed processing of very large data sets, demand professional-level programming skills. In these situations, interdisciplinary teams often pair members skilled in software engineering with those experienced in data screening, hypothesis formulation, and scholarly reporting. Behavioral scientists, with their deep knowledge of subject matter, play a vital role in such collaborations, even more so when they understand core concepts and techniques in machine learning. The chapter concludes with a survey of R as a programming environment optimized for advanced statistical analysis and data visualization. A comparative discussion of R versus Python guides researchers in choosing the programming language best suited to their needs.

Data visualization and the defining features of big data are addressed in Chapters 9 and 10, respectively. Chapter 9 begins by outlining general principles for constructing clear and informative data graphics, acknowledging that a comprehensive survey of every available chart type lies beyond the scope of a single chapter. Illustrative examples—small multiples, density plots, dot plots, and correlograms among other kinds of visualizations—demonstrate how these principles apply in practice. Guidance on color selection highlights palettes that accommodate various forms of color vision limitations among target audience members. Chapter 10 examines what renders a data set "big," distinguishing essential characteristics from more peripheral attributes. Emphasized in this discussion is that sheer case count (N) does not fully capture big data complexity. Practical tips for managing analyses of very large data sets are provided, including an example for a detailed, step-by-step "divide-and-conquer" strategy for processing a substantial text file with efficient disk storage and memory utilization.

Chapter 11 surveys data reduction strategies, including variable and case selection as well as techniques for curtailing dimensionality and heterogeneity in large data sets. Dimensionality reduction methods, such as principal component analysis and linear discriminant analysis, and heterogeneity reduction via cluster analysis build on approaches familiar to researchers in psychology and related fields. Complementary sampling techniques for extracting representative subsets enable focused analyses on fewer cases than those in the complete data set. These concepts are demonstrated through visualizations applied to a real data set of parental descriptions of children enrolled in regular versus special education classrooms.

Chapters 12–16 of Part IV deal with machine learning concepts alongside practical examples. Chapter 12 outlines the defining characteristics of supervised learning algorithms prediction and classification in labeled data sets, where true outcome status or class membership is known. Decision trees, random forests, and support vector machines are introduced and each method is demonstrated on a common, synthetically generated data set. Chapter 13 addresses the selection of analysis settings—often termed hyperparameters—in supervised learning. Techniques for balancing model fit on training data against generalization to new test or validation sets are presented, with an emphasis on cross-validation strategies. Chapter 14 focuses on the estimation of complex or nonlinear main and interaction effects when specific patterns cannot be prespecified. Regularized regression methods are introduced as a means of parameter selection, employing penalty schemes to attenuate or exclude smaller effects.

Chapter 15 provides an in-depth exploration of cluster analysis as an unsupervised learning approach for reducing heterogeneity when true class labels are unknown. A flexible clustering algorithm capable of handling continuous, ordinal, and nominal variables is applied to a real data set encompassing demographic characteristics, lifestyle factors, and obesity. Chapter 16 closes the book by surveying four data science topics beyond basic machine learning:

1. Enhanced cross-validation techniques that more cleanly separate model tuning (hyperparameter selection) from evaluation of predictive performance on unseen data.
2. Tiny machine learning, which is the design of highly optimized, power- and memory-efficient models for deployment on microcontrollers and other resource-constrained devices with minimal or no network connectivity.
3. Model compression and simplification strategies that reduce computational and storage requirements without substantial loss of accuracy, a critical consideration for both tiny machine learning and the deployment of large, complex models.
4. Neural network fundamentals, including a demonstration of compressing a simple network architecture.

Every chapter begins with a concise summary of objectives with definitions of key concepts. All defined terms (presented in boldface in the text) are listed at the end of each chapter as a reminder of concepts to review. Most chapters have exercises, and suggested answers for them all are presented toward the end of the book.

Pedagogical Approach

I wrote this book for applied researchers and advanced students who are not experts in statistics or in computer science. Readers who are already statisticians or computer science professionals have different needs and this book would not be suitable for them.

Consequently, I will speak in my authorial voice that communicates from one researcher to another in clear, concise language, emphasizing fundamental concepts over extensive equations. Many complex topics are presented with full nuance rather than simplified away, while numerous worked examples—complete with syntax and output—support deeper understanding. Most chapters include end-of-chapter exercises (with answers provided later in the book) covering conceptual questions, interpretation of selected statistics and outputs, and programming tasks that require adapting the original code and comparing newly generated results with the examples.

CHAPTER 8

Programming

Actualization through Coding

OBJECTIVES: *Emphasize the benefits of programming skills for researchers with little formal background in this area. Outline realistic paths for busy researchers to improve their programming skills. Introduce the R environment as a stepping stone into the larger world of coding.*

KEY CONCEPTS

- *Artificial intelligence code generators* translate natural language descriptions of a problem into working code, but that output is not always correct, and novices should avoid relying on them excessively lest they fail to fully understand their own syntax.
- *Object-oriented programming* is a coding approach that models real-world concepts as *objects* by combining data and functions in ways that support reuse and maintenance of code.
- The R environment for data analysis and visualization is object-oriented, *procedural* (code is ordered as steps that modify shared data), and *functional* (functions are the primary building block for constructing programs).
- *Data types* in R define the type of data that an object holds (e.g., numeric, integer, character, logical, or complex) and determine how it's stored in memory and manipulated.
- *Data structures* define formats, such as vectors, matrices, lists, and data frames, that organize, group, and provide access to their contents.
- The *R Markdown scripting language* allows researchers to combine text, code, output, and visualizations into a single document that can be interactive, supporting collaborative work and reproducibility.

The aim of this chapter is to help prepare researchers with minimal backgrounds in computer science for analysis tasks requiring programming. The next section outlines the reasons why programming wherewithal offers benefits even when there are no plans to

analyze large data sets. The skills needed here generalize to a wide range of research contexts, emphasizing that programming acumen can enhance efficiency and reproducibility in data analysis. Practical strategies are offered for researchers who face limited time or heavy commitments on current and planned projects. These include self-learning and participation in targeted seminars or tutorials on programming. Because the R computing environment is a stepping stone into a larger world of programming for many behavioral science researchers, basic characteristics of R, including data structures, functions, control structures, and objects, are described and demonstrated. Many of these concepts generalize to other computing environments or languages, so readers with little current interest in R can still benefit from this presentation. A comparative discussion then contrasts R with Python from the perspective of behavioral science applications, highlighting strengths and trade-offs for data analysis tasks. Developing stronger programming skill takes time and much practice, so no single chapter of a book offers a shortcut to expertise. If the only achievement of this presentation is to lower the barrier to entry and ease the initial steps, its modest aim will be fulfilled. Readers are encouraged to engage with exercises, review code examples, and seek community resources to continue building their skills.

Programming as a Core Research Skill

Writing for professional software developers, McKenzie (2015, para. 2) noted that "Every great developer you know got there by solving problems they were unqualified to solve until they actually did it." The challenge of using programming to solve problems beyond one's training also resonates with many researchers. Although later releases of many computer applications tend to be more user friendly over time, early adopters can pay a price in extra effort and frustration. In some data analysis software, certain options or procedures are accessible only by typing commands rather than clicking menus, and familiar point-and-click tools may simply lack the needed features. Emerging research questions or newly available data sets can demand a substantial increase in coding skill (Chen & Wojcik, 2016). Behavioral science researchers who do not belong to multidisciplinary teams with expert programmers may need to take on programming challenges for which they are relatively unprepared.

Mastery of programming basics can potentially benefit researchers in the five ways summarized next (Crump, 2018; Gould, 2019; Keyes, 2020; Scherer et al., 2021):

1. *Generalized creativity:* Programming is a creative activity where a larger problem is broken down into simpler, easier-to-manage steps or components, a process that mirrors the planning required to carry out a complex research project.
2. *Recycling reduces future work:* Code segments that work can be reused or applied when the same situation is encountered again, so new problems take time to solve them from scratch just once.
3. *Loops and naming:* Editing the names or attributes for hundreds or thousands of variables via a point-and-click interface quickly becomes impractical. A simple loop or a few lines of code can automate those repetitive edits far more efficiently.
4. *Understanding extreme cases:* Programming, like engineering and medicine,

teaches one to spot **edge cases**, those extreme, rare, or unexpected conditions within normal operating parameters that can trigger failures if left unaddressed.

5. *Transparency in computation:* Commercial software often relies on proprietary, closed-source algorithms to convert signals, such as electroencephalography (EEG) recordings, into analyzable data. In contrast, when researchers write their own functions or scripts for data preparation, they can publish that code openly, fostering transparency and reproducibility.

Deardorff (2020) interviewed postdoctoral researchers, graduate students, and research staff in biomedicine about their reasons for enrolling in R or Python programming workshops. Nearly all participants described themselves as novice programmers. Their motivations clustered into four main themes:

1. *Independence in data analysis:* Being able to write code empowers researchers to analyze and interpret their own data without relying on specialized programmers.
2. *Programming literacy:* Learning to read and understand colleagues' scripts enhances communication and collaboration within multidisciplinary teams.
3. *New research opportunities with big data:* Mastery of programming enables work with massive data sets beyond the capacity of standard tools like Excel or SPSS, unlocking novel research avenues.
4. *Tool flexibility:* Switching seamlessly between point-and-click interfaces and syntax-based workflows extends the researcher's skill set.

To summarize, researchers surveyed by Deardorff (2020) enrolled in programming workshops not to become software developers or out of a passion for computers, but to apply new coding skills directly in their research. For these practitioners, concise, hands-on training delivers more value than theoretical courses on software development or foundational computer-science concepts.

Juavinett (2022) described programming education in neuroscience as stuck at an impasse. Although ever-larger bioinformatics data sets and machine learning methods have intensified calls for training in coding (Goldman & Fee, 2017), dedicated programming courses remain somewhat rare in neuroscience graduate curricula.[1] Juavinett (2022) reviewed surveys of neuroscience degree programs and reported that

- Only about 15% of PhD programs required a programming course, and roughly 55% offered one as an elective
- Fewer than half of neuroscience faculty felt comfortable or extremely comfortable with coding
- Programming courses were unavailable in most undergraduate neuroscience programs

[1] The same thing could be said about disciplines other than neuroscience, where data are increasing both in size and complexity. For instance, Hagstrom and Maranzan (2019) described ways to close the gap between technological advance, including machine learning and artificial intelligence, and professional psychology training.

Juavinett (2022) encouraged institutions to incentivize faculty participation in accelerated, bootcamp-style programming courses, whether in person or online, to build teaching confidence. Some practical strategies for students, faculty, and researchers to boost their coding skills are outlined next.

Pathways to Improving Programming Skills

Three main pathways allow researchers without a formal computer science background to pick up programming skills. Each option has its trade-offs:

1. *Free/DIY/Google:* This do-it-yourself (DIY) tactic involves using freely available online resources for programming and internet searches with Google (or any other web search engine) to study at one's own pace. This tactic costs nothing but content quality and accuracy vary widely, and some examples may be outdated or simply wrong.
2. *Pay to learn in the short term:* Many educational resources—courses, workshops, boot camps, and seminars—are available for a fee. Some are offered by universities and others by private companies in continuing education. This market can be volatile with courses and whole companies suddenly appearing or disappearing, so caveat emptor (let the buyer beware).
3. *Formal degree programs:* Many community college programs that require 1–2 years of full-time study lead to diplomas, certificates, or associate degrees in programming. Many universities offer degrees programs in computer science or data science at the bachelor's through doctoral levels. This route offers depth and accreditation but demands the greatest amount of time and financial investment.

The DIY approach relies on researchers finding only the content needed to solve a specific problem. The internet offers a plethora of programming resources, especially for open-source languages and environments such as R and Python. Beyond official manuals, there are podcast archives, curated lists of books and journals, user-group forums, weekly newsletters, and conference materials. There are also free online textbooks for programming in R[2] or Python,[3] among other languages. Using precise search queries—for example, "convert a matrix to a data frame in base R" instead of "data in R"—yields far more focused results. Bear in mind, however, that content on unvetted websites varies widely both in veracity and value.

Pay-to-learn options can a be sweet spot between free searches over the wilds of the internet and degrees programs that potentially require years of study. Some university courses are coordinated by offices or schools for continuing studies, which means that participants need not be registered as students at the home institution. Courses may

[2]https://www.rstudio.com/resources/books

[3]https://pythonbooks.org/free-books

combine live interaction with instructors and self-paced online materials, while others remain entirely static with no direct contact. Certain boot camps provide intensive, accelerated training for experienced programmers, whereas others start at the introductory level. Many paid programs award certificates or digital badges on completion, and some providers grant ongoing access to materials and active alumni forums.

Researchers aiming to gain substantial programming expertise can pursue academic degrees, either full-time or part-time. Full-time study demands significant time and financial commitments and yet might be practical only for those who can fit a short program into a sabbatical. Part-time study spreads coursework over months or years, offering greater flexibility for active researchers but at a slower pace. This path may be overkill for teams that already include dedicated programmers, where it can make more sense for researchers who are not programmers to concentrate on core scientific tasks (hypothesis formulation, study design, data collection, etc.). Yet for individuals driven to master coding, formal study can be highly rewarding. As Armstrong (2001, p. 33) observes, "being a researcher is not a job, not a role, not a status, nor a state of being, but a praxis, and a strategy and process for lifelong learning itself," underscoring research as a continual learning experience.

My own path as a researcher developing programming skills began with learning FORTRAN and BASIC during my undergraduate studies. In graduate school and beyond, I adopted multivariate statistical techniques and quickly grew frustrated by the limitations of commercial analysis packages. To overcome these constraints, I started writing custom syntax to produce the exact results I needed. Over time, my interest in coding intensified. I eventually enrolled part-time in a graduate diploma program in computer science at my home institution. This diploma is equivalent in rigor to an undergraduate degree but focuses solely on computing courses without distribution requirements in other fields. Completing the program took two and a half years of part-time study—including summer terms—but it was well worth the commitment. The skills I gained continue to strengthen my work as both a researcher and an author. My experience is hardly unique: many researchers routinely pursue new skills to expand their career possibilities.

AI-Assisted Code Generators

This survey of options for researchers aiming to strengthen their programming skills would be incomplete without mentioning AI-assisted code generators. These tools can produce complete code snippets from natural language descriptions of tasks, scenarios, structures, or sets of conditions in the programming language of choice. They also offer the capabilities listed next:

- Translation of scripts between languages, such as from R to Python and vice versa
- Explanation or diagnosis of submitted code segments by clarifying logic and identifying issues
- Suggestion of fixes when code triggers error messages, thus streamlining debugging

Coding assistants rely on machine learning for large language models that "understand" and generate the syntax, patterns, and structures found in human-written code. They are sophisticated extensions of traditional **autocompletion** in syntax editors, where function parameters or warnings about common mistakes are suggested while the programmer types.

There are now many AI-assisted code generators, some that offer free tiers with the option for paid plans to access more advanced features. Four widely used, professional-grade assistants are summarized next:

1. GitHub Copilot is a subscription-based service that plugs into a syntax editor. It monitors what the programmer types, "reading" the surrounding code with comments, then drafts entire blocks (functions, tests, setup code) tailored to what the programmer is building. There is a free level, but it is limited to a 30-day cap on usage.[4]
2. Windsurf (formerly Codeium) supports Copilot-style completions for over 70 programming languages with support for building complex, multifile applications. A free tier with limits on monthly usage is available.[5]
3. Amazon Q Developer (formerly Amazon CodeWhisperer) is an editor extension that gives code suggestions, checks code for security vulnerabilities, and supports working in the Amazon Web Services for cloud computing. It supports the most common programming languages like R, Python, Java, among others. A free version for individual developers has monthly usage caps on security scans but is unlimited for real-time code suggestions.[6]
4. Tabnine is a programming assistant trained on open-source code with permissive licenses from sources that are that are publicly listed, which helps to protect programmers from inadvertently violating intellectual property rights when adapting AI-suggested code. There are subscription versions for individual programmers or organizations. A free version with limited support for more advanced features such as workflow integration is available.[7]

Microsoft Copilot and ChatGPT also support coding with varying levels of cost. Free support in Microsoft Copilot offers unlimited AI-powered chat and code generation but has caps on image generation, and it may rely on older or lighter-weight AI engines. Subscription tiers remove those restrictions.[8] ChapGPT offers comparable free (with restrictions)

[4]https://github.com/features/copilot

[5]https://windsurf.com

[6]https://aws.amazon.com/q/developer

[7]https://www.tabnine.com

[8]https://aka.ms/copilot-plan

and subscription versions (with fewer or no restrictions) for AI-assisted code generation.[9] While writing this book, I used Microsoft Copilot to help solve two challenging (for me) coding tasks, and I describe these experiences in Appendices 14.A and 14.E.

It is not hard to anticipate that AI-assisted code generators could save programmers time and effort, but they are not panaceas, and their misuse can lead to problems. The thorny issue of intellectual property rights was mentioned, and it is part of a larger concern about generative AI computer tools—whether they are code, image, or text generators—see Appel et al. (2023), who offer suggestions for both content creators and developers of generative AI programs about how to avoid problems in this area. Three additional challenges are outlined next:

1. It is a reality that AI-coding assistants do not always generate syntax that actually works. For example, Yetiştiren et al. (2023) evaluated Python code generated by ChatGPT, GitHub Copilot, and Amazon Q Developer against software quality metrics. About 90% of generated code by all three AI assistants was free of syntax errors, but rating for *correctness*—whether the code actually performed as intended—ranged from 31.1 to 65.2% over the three assistants. Yetiştiren et al. (2023) also reported that obtaining higher rates of correctness required very clear and accurate problem descriptions by human programmers submitted to AI programming assistants.

2. In their review of four AI-assisted tools, Hliš et al. (2023) reported that satisfaction levels were higher among experienced developers than less experienced users who struggle more with filtering out irrelevant or misleading suggestions. An example is that AI code generators sometimes write syntax that is much too complicated for the task with unnecessary data structures, control structures, or function calls. If the same task can be achieved with simpler code, then that solution is generally preferred. After all, more code is more opportunities for trouble. But for developers of all experience levels, ratings for the relevance of the assistant's suggestions were relatively low.

3. Logical mistakes in reasoning by a human programmer could prompt an AI-assisted code generator to pursue a path that leads to multiple errors or output that is wrong. This is not a criticism of AI-assisted coding per se because developers bear the final responsibility of their code, including correct use of tools to generate that code, Nevertheless, AI assistants can compound poor planning by human programmers.

To summarize, although AI coding assistants can be helpful in certain contexts, neither professional developers nor researchers should rely exclusively on them for essential reasoning or determining optimal strategies in software development or academic study. Independent review by human programmers, especially those who did not author the original code, is irreplaceable. Code generated by AI tools may introduce subtle bugs, security vulnerabilities, or fail to account for edge cases, underscoring the importance of thorough human oversight and testing (Hliš et al., 2023; Sharma & Rattan, 2025). For novice

[9]https://openai.com/pricing

programmers, excessive reliance on AI assistants may hinder the development of a deeper understanding of their own code.

R in Context

This summary is based on the online manual by the R Core Team (1999–2026), which can be consulted for more detailed information. In addition, numerous books and websites are devoted to R, so resources are widely available. Briefly, the R language and environment is part of the GNU Project and the free software movement, where "free" means that users can run, study, modify, and redistribute code without restrictions (Stallman, 2009). The R environment is also free in the monetary sense, as it can be downloaded and installed at no cost. It was developed by Ihaka and Gentleman (1996) and is based on the S language for data analysis and graphics, which originated from work by John Chambers and colleagues in the mid-1970s at Bell Laboratories (Becker et al., 1988).

Versions of base R for personal computers—R for Windows, R for macOS, and R for Linux (R Core Team, 1999–2026)—include a graphical user interface for interacting with the R Console, a command-line interpreter where users can type commands directly. An alternative approach is batch processing, in which users run lines of syntax entered into a separate script window or executed via the base R function `source()` for reading input from a file stored locally or from a website. Text-based results from executed commands are displayed in the R Console, and its contents can be saved to an external text file. If there are no syntax errors, a separate window opens to display data graphics, if such are generated by executed syntax.

A base R installation includes functions for core types of statistical analysis, but user-contributed add-on packages, also freely available, greatly expand its capabilities. A **package** typically contains functions, data, examples, and documentation for more specialized analyses. As of early 2026, over 25,000 R packages were available on CRAN (Comprehensive R Archive Network), the official repository. Additional sources for R packages include GitHub, a general-purpose code hosting and collaboration platform for open-source projects; Bioconductor, an open-source initiative focused on genomics data analysis; and R-Forge, a site for collaborative R package development.[10,11,12] Many R packages are demonstrated in appendices of this book, and data science-oriented packages, especially for machine learning, are covered in Chapters 12–15.

Using R for statistical analysis requires little prior programming experience. This is because users can interact with base R or its packages through built-in **functions**, predefined, modular units of code that perform specific tasks. Functions are defined by their names, parameters, and outputs. Parameters represent the data (input) to be processed or specify options for the analysis, while the function generates output in various forms, such

[10] https://github.com/

[11] https://www.bioconductor.org

[12] https://r-forge.r-project.org

as tabular summaries or data visualizations. Functions "hide" the underlying implementation details, allowing users to focus on what the function does, how to call it, and how to specify its parameters. For those curious about the inner workings, the source code of a base R function can be viewed by typing its name (without parentheses) at the command prompt. Source code for functions in many packages is also available through R Package Documentation.[13] However, researchers are not required to understand the source code to effectively use functions in R.

In addition to being **procedural**—where functions and subroutines call each other—and **functional**—where functions can be composed or combined to create new ones—the R programming language is also object-oriented. This means R supports **object-oriented programming**, a framework in which software is organized into modular, reusable segments of code that define classes. **Classes** specify the attributes (data) and behaviors (methods) of **objects**, which are specific instances of those classes. Object-oriented programming has played a major role in software development, as discussed by Black (2013). In R, many fundamental structures, including those used to store data or the output of function calls, possess attributes and methods that can be directly accessed or modified by the researcher. These characteristics open additional possibilities for manipulating or restructuring data and analysis results, as explained in the following section.

Data Types and Data Structures

Summarized in Table 8.1 are basic **data types** in R, which refer to the kinds of data that can be stored or manipulated in the language. The list is not exhaustive, for example, R includes special classes for representing dates and times (Spector, 2008, chap. 4), and many of these data types are also found in other programming languages. Character data, or text that may include letters, numerals, or symbols forming strings such as names, addresses, or other textual information, are enclosed in R using single or double quotation marks. A **factor** is a special type of character variable that represents predefined levels of a categorical variable. Factors are commonly used in R to define groups or conditions in statistical analyses. Logical data indicate status in terms of TRUE or FALSE (or NA for missing or unknown values), based on conditions, operations, or comparisons specified by the researcher. Logical values are often used to control syntax execution, for example, selecting cases that meet specific criteria for further analysis.

Described in Table 8.1 are two basic types of numerical data in R. *Integers* are whole numbers stored without decimal points and require less computer memory than values specified as "double," which refers to double-precision floating-point format used for storing numbers with decimals or in scientific notation—for example, 1×10^2 or 1e2 in exponential notation to represent the number 100. *Complex numbers* consist of real and imaginary parts, with the imaginary part *i* defined as the square root of –1, which has no real solution. Complex numbers are commonly used in fields such as engineering and

[13] https://rdrr.io

TABLE 8.1. Basic Data Types in R

Type	Description
Character	Plain text (nominal categorical data with no inherent order) enclosed in single or double quotation marks; could include letters, numerals, punctuation marks, or other symbols all treated as characters
Factor	Predefined levels of categorical data with meaningful order or hierarchy (e.g., "low," "medium," "high")
Logical	A logical variable with one of three possible values, TRUE, FALSE, or NA (missing values); can be combined with Boolean comparison operators[a] or Boolean logical operators[b]
Numeric	
Integer	Whole numbers (including zero) with no decimal points; requires less storage space than double
Double	Numbers that can have decimal points or expressed in scientific notation; stored in double-precision floating-point format (64 bits)
Complex	Numbers with real and imaginary parts (e.g., 4 + *i*, where *i* is the square root of –1); stored as pairs of double-precision numbers
Raw	Data in binary form (raw bytes of data)

[a]Greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), equal to (==), not equal to (!=).
[b]AND (&), element-wise OR (|) that compares vectors element by element, conditional (short-circuit) OR that only checks the first element then stops (||), NOT (!)..

electronics to model signals that oscillate or vary periodically in a wave-like, sinusoidal pattern. The "raw" data type in R (Table 8.1) is for storing data that are sequences of bytes, such as locations in computer memory or when reading or writing binary data files.

Described in Table 8.2 are common R **data structures** that refer to how data are stored, organized, and accessed in computer memory or on storage devices. These structures vary in their number of dimensions and in whether they can store elements of different data types. A **vector** in R is a one-dimensional array (i.e., a single row or column) that stores elements of the same type, such as character, logical, or numeric (see Table 8.1). Vectors use numerical indexing that begins at 1. For example, the syntax `A[4]` refers to the fourth element in the vector named A, while `A[1:4]` accesses elements 1 through 4. A **list** is also a one-dimensional array, but unlike vectors, it can store elements of different data types, for instance, a string as the first element, a double-precision number as the second, and so on. Lists in R can also contain functions or other data structures, including nested lists. Elements in a list are accessed using double square brackets, such as `B[[2]]` to refer to the second element in the list named B.

A **matrix** in R is a two-dimensional or tabular data structure that stores elements of the same type—see Table 8.2. Matrices have two indices, one for rows and one for columns. For

TABLE 8.2. Common Data Structures in R

Structure	Dimensions	Description
Vector	1	Linear ordered collection of elements all the same type; an array with one dimension
List	1	A vector with elements that can be of different types, functions, or whole data structures, including other lists
Matrix	2	Tabular structure where all elements are the same type; an array with two dimensions
Data frame	2	Tabular or spreadsheet structure where rows are cases (observations, records) and columns (variables, features) are vectors all with the same length, each column is a single type but different columns may have different types
Array	≥ 1	All elements are the same type; a multidimensional matrix when dimensions ≥ 3

Note. Tabular is also called rectangular.

example, the syntax `C[1,2]` refers to the element in the first row and second column of the matrix named C. The combination of numeric vectors, matrices, and built-in functions for manipulating their elements gives R powerful capabilities for linear algebra. A **data frame** is also a two-dimensional structure, but its organization resembles that of a spreadsheet. Each column contains elements of the same data type, while different columns may contain different types, for example, characters in the first column, integers in the second, and so on. Data frames are among the most commonly used structures for storing and modifying raw data in statistical analyses, especially in behavioral science research—see Maindonald et al. (2024) for examples. Finally, an **array** in R can have any number of dimensions, with the constraint that all elements must be of the same type. Arrays are useful for storing spatial or time series data that involve multiple variables or features.

Objects and Attributes

Because R data structures are objects with attributes, these properties can often be accessed or modified by the researcher to tag, label, or add information to the structure. Examples of attributes include column names, row names, and comments, all of which can be edited without affecting the structure's functionality—see Appendix 8.A for examples. Exercises 1 and 2 for this chapter involve exploring the characteristics and structure of a built-in R data set.

The syntax and output in Appendices 8.A–8D were formatted using the rmarkdown package (Allaire et al., 2025), which is based on the **Markdown scripting language** developed by John Gruber in collaboration with Aaron Swartz in 2001 (Krewinkel & Winkler,

2017). This package enables researchers to combine text, R code, output, and visualizations into a single document, which can be structured to be interactive. The potential of rmarkdown and related tools to enhance communication of analyses and support reproducible research in R is discussed in more detail later in this chapter.

Most output in R is also structured as an object. This means that their attributes can generally be accessed, extracted, or copied to another object for more direct manipulation. These capabilities, where the output of one analysis is saved and passed on to a second analysis, and so on, is extremely convenient—see Appendix 8.B for an example of extracting and saving output from the linear regression function `lm()` in base R. Exercise 3 involves manipulating regression results for a different data set.

Importing Data

One of the most common concerns for researchers learning R is how to import their data. Base R includes functions for importing tabular data from external plain text files, where columns are separated (delimited) by commas, spaces, or tabs. These files may include a **column header**, which is a list of column (variable) names. If a header is present, then (1) it must appear as the first line in the file, and (2) both the header and the data (record) lines must use the same delimiter between values. If no header is provided, the records can still be read, and R will assign default column names such as "V1," "V2," and so on. These attributes can be modified later by the researcher—see Appendix 8.A.

Base R includes the function `read.table()` that opens and reads external text files into a data frame. Its parameters specify various aspects of the input, including: (1) the source of the data file (e.g., directory and file name, if the file is stored locally on the researcher's computer); (2) the separator character between fields; (3) whether character variables are enclosed in quotation marks (single, double, or none); (4) the character used for decimal places in double-precision numeric data (e.g., some countries use commas as decimal points and semicolons as field separators); and (5) how blank lines should be handled. Additional options are available to fine-tune the import process depending on the structure and formatting of the data file.

A second R function, `read.csv()`, is essentially a version of `read.table()` with a predefined set of parameters tailored for reading external comma-separated values (CSV) files, a widely used format where data entries are delimited by commas.[14] These files can be opened in plain text editors, Microsoft Excel, and many other applications used for data analysis. The `read.csv()` function can also be used to read CSV files from internet sources for analysis—see Appendix 8.C for an example. To export data, the function `write.csv()`, demonstrated in Appendix 8.A, writes a data frame to an external CSV file. Similarly, `write.table()` serves as the counterpart to `read.table()` for writing tabular data. For more information on analyzing tabular data in R, see Sgro and Malecki (2022).

[14] The related function `read.csv2()` assumes decimal points in numbers use commas and fields are separated by semicolons.

Exercise 4 asks readers to adapt the code in Appendix 8.C to download and describe a different CSV file from an external website.

There are many other formats for data files besides CSV. Fortunately, there are special packages for R that read data files in these other formats. Just three sets of examples are described next but see Awan (2023) for more information:

1. The haven package (Wickham, Miller et al., 2025) imports and exports data files in SPSS, Stata, or SAS formats, and the foreign package (R Core Team, 2025) reads and writes all these formats plus Minitab, Systat, and dBase files, among others.
2. Several packages connect to databases, including odbc (Hester et al., 2025), which is based on the Open DataBase Connectivity (ODBC) protocol to access database management systems; the ROracle package (Mukhin et al., 2021) for working with Oracle databases; and RMySQL (Ooms et al., 2025) for interacting with MYSQL, an open-source structured query language, for relational databases.
3. Other packages also help R to analyze data sets too large to fit into the memory of a single computer or store on it. Examples include bigmemory (Kane et al., 2024) for analyzing massive matrices with shared memory for parallel analyses on computers with 64-bit operating systems and ff (Adler et al., 2025) for generating data structures optimized for efficient and fast access of large data sets stored on disk.

Researcher-Defined Functions

Many built-in statistical functions are available in base R, such as `min()`, `max()`, and `mean()`, which return the minimum, maximum, and average values in numeric vectors, respectively. Installed packages for R increase the number of available functions even more. It is also possible to write customized functions that automate a sequence of commands and produce an output. This capability deals with situations where no single built-in function or package-based function is available to carry out a specific task.

The basic form of a function in R is

```
function_name <- function(parameters) {
  function body
}
```
(8.1)

where the main parts are its name, parameters, and body (Kosourova, 2022). The parameters or **arguments** are the variables or data structures involved in processing within the function body. Parameters can be set to default values, if many typical cases are expected; otherwise, the arguments for a function are specified in syntax when the function is called. Code within the function body can include base R or package-based functions, each called with specific parameters of their own. See Appendix 7.B for examples of custom functions that compute posterior distribution means and variances in Bayesian estimation.

Control Structures

Most programming languages, including R, provide **control structures** to manage the flow of execution. Control structures specify which code blocks run and under what conditions. There are two basic types of control structures: decision structures and loop structures (Shah, 2020). A **decision structure** chooses between two or more alternative execution paths based on specified conditions. An example is an **if–else block**, which is expressed in pseudocode at the top of Table 8.3. This block evaluates three mutually exclusive conditions in sequence and executes exactly one of the corresponding code segments per iteration. This if–else block can be contracted so that just one of two different code segments is executed (true vs. false) or expanded to include four or more distinct cases and corresponding conditions and code segments.

TABLE 8.3. Pseudocode for Control Structures

Structure	Description
If–else block	

If (condition 1) then
Code segment 1, if condition 1 is true
Else if (condition 2) then
Code segment 2, if condition 1 is false but condition 2 is true
Else
Code segment 3, if both conditions 1 and 2 are false
End if
	Multiple conditions are checked sequentially, and different code segments are executed based on those conditions, which are typically mutually exclusive so only a single code segment is executed

For loop	

For (Start value to end value by change)
Code segment, if between start and end values
End for
	Index or control variable, which specifies the number of iterations, is initialized at the beginning of the loop; index is incremented or decremented after each iteration

While loop	

While (condition)
Code segment, if condition is true
Update condition
End while
	Code segment is executed only when a condition is true, but the number of iterations could be zero, if the condition is not initially true

Repeat loop	

Repeat
Code segment
Update condition
If (Condition)
Break, if condition is true
End repeat
	A code segment is executed at least once and then is repeated until a stop condition is true

A **loop structure** repeats a block of code as long as a condition remains true. It decides after each iteration whether to continue executing the block or to stop processing. A **for loop**, shown in the middle of Table 8.3 in pseudocode, is used when the number of iterations is known before execution of the loop begins. For example, to apply a calculation to every current employee, the total number of employees determines the number of iterations. To set up a for loop, specify a control variable or index with a start value, an end value, and an optional step size. The loop runs the code segment with the index at its start value, then updates the index by adding (or subtracting) the step size and repeats until the index goes beyond the end value. Note that some languages treat the end value as inclusive (the loop runs when index ≤ end), while others treat it as exclusive (index < end). In R, a for loop includes the end value, so it behaves as if it is checking index ≤ end. For example, given the sequence 1:5 and step size of 1, the loop runs five times.

Two other R loop control structures (shown in the bottom half of Table 8.3) handle cases where the number of iterations is not known in advance or depends on runtime results. A **while loop** (**pretest loop**) checks its condition before executing the loop body. If the condition is true, the body runs, updates whatever state is being tested, and then the condition is rechecked. Iterations continue until the condition evaluates as false. For example, a while loop might test whether new data are available. If so, it archives the data in the loop body and repeats; if not, control moves directly to the statement following the loop. In contrast, a **repeat loop** (like a **do–while loop** in other languages) always executes the body at least once and relies on an explicit break to exit. A conditional break is placed inside the loop that stops execution when a certain condition becomes false. Without a break, the loop would run indefinitely. An example of invoking a user-defined function within these control structures is given in Appendix 8.D. Exercise 5 involves writing a loop structure that computes logarithms and factorials for a vector of numbers.

As mentioned, the formatted input and output in Appendixes 8.A–8.D was generated using the R Markdown scripting language. Listed in Appendix 8.E are the contents of the R Markdown file used to format Appendix 8.D. The R Markdown language is a tool for creating reproducible documents that combine syntax, data, and output in a variety of formats, including interactive web applications, is outlined next.

R Markdown and Other Resources for Reproducible Computing

An R Markdown file (usually with an .Rmd extension) is a plain-text document that integrates

- Executable chunks of R code
- Markdown formatting for headings, lists, and emphasis
- Narrative text explaining the analysis
- Inline data and results such as tables or plots

This single "master" document supports reproducible research by running the code, capturing its output, and embedding everything (including comments) into a single file. Because the code, data, output, and annotations are in the same place, collaborators can edit the document together without juggling separate script files and result outputs that do not automatically reflect changes in other parts (i.e., they are no longer synchronized) (Baumer & Udwin, 2015). The authors just cited identify the six key advantages of using R Markdown over other scripting frameworks or static output formats summarized next:

1. *Simpler syntax:* Scripting in R Markdown is simpler compared with both HTML and LaTeX for typesetting equations,[15] although not as feature-rich in specialized applications.
2. *Readable source:* The plain-text .Rmd file is more human-readable and easier to scan than equivalent HTML or LaTeX source files.
3. *Single-layer formatting:* All styling, such as headings, lists, and emphasis, is controlled in the script itself, eliminating the need for menus or dialog boxes typical of word processors.
4. *Dynamic computation:* Code segments are executed when the document is rendered, so any changes to data automatically update tables, plots, and other results. Live data can also be pulled from databases or websites.
5. *Multiple format output:* A single R Markdown file can produce HTML, PDF (via LaTeX), Word, and even Tufte-style documents with minimal coding (Xie et al., 2023). **Tufte-style documents** adhere to minimalist design principles for data graphics by Edward Tufte and are intended to enhance clarity and focus by stripping away distractions and optimizing use of space. These principles are covered in the next chapter on data visualization.
6. *Interactive documents:* Beyond static reports, the R Markdown framework supports the creation of self-contained interactive components (charts, tables, maps) embedded in HTML documents that respond to user actions and stand-alone R-based Shiny apps that combine a user interface and server logic for real-time, browser-based data interaction.

When the rmarkdown package is installed, the knitr package (Xie, 2025) is installed automatically. This package serves as the computational engine that runs the R code and embeds outputs, such as tables or graphics, into the rendered document according to the researcher's specified format. It also provides tools for working with large data sets, including caching, chunking, memory management, and parallel processing. These features are described in Chapter 10 on big data.

Quarto is an open-source publishing system developed by the team behind the RStudio IDE (see the next section).[16] It generates dynamic content in R, Python, and other programming languages. Although Quarto is built on the same Markdown foundation

[15] https://www.latex-project.org

[16] https://quarto.org

as rmarkdown and can generate interactive documents and web apps, Quarto supports a wide range of applications for interactive content and integrates more seamlessly with Jupyter notebooks (also described later in this chapter). The quarto package for R (Allaire & Dervieux, 2025) provides these capabilities. Dogucu (2025) describes examples of using Quarto to teach concepts of reproducible research in statistics education.

RStudio IDE

An alternative way to work in R is through the RStudio IDE, an integrated development environment with a graphical interface available for Windows, macOS, and Linux platform computers. It is available in both a free desktop edition and multiple commercial versions.[17] Support for the free version is limited to user-community-based forums and online documents. There are no built-in features for collaborating with other users nor are core security functionalities available. There is also more limited database connectivity in the free version. Commercial versions add these features and options for enterprise deployment. Pricing for commercial versions depends on the number of users and specific computing requirements. Just the free version of RStudio is described next.

The RStudio IDE requires prior installation of R, and the R Console is shown in one of four basic RStudio windows. The second window is for a syntax editor that supports text highlighting, automatic line numbering, and code completion. A third window displays environment and history information in R, including user-defined objects, such as data structures or variables, and a list of previously executed commands. A fourth window is for managing files or packages, displaying graphics created in R, and accessing help files. A debugger for inspecting, setting breakpoints, or stepping through R code as it is executed is available. RStudio also has a built-in Markdown editor for generating R Markdown documents, and there is a dedicated "Knit" button that automatically renders R Markdown files into different output formats. The RStudio environment also supports the development, testing, and deployment of Shiny apps. Finally, RStudio can be used with Python through the R package reticulate (Kalinowski et al., 2025).

For researchers with strong programming skills who know how to use a debugger, manage added complexities of using an IDE, or develop interactive web applications, RStudio IDE has a lot to offer. Otherwise, it is not a magical alternative to more basic tools such as R for Windows. Programming in R has an obvious learning curve but so does learning how to use an IDE such as RStudio. Some researchers (like me) prefer the simpler, more straightforward user interface in R for Windows (or related versions for other operating systems), although I appreciate the conveniences in RStudio. Books for learning how to use RStudio include Campbell (2019), Jones et al. (2023), and Kronthaler and Zoellner (2021), and freely accessible online tutorials include Kosourova (2022) and Bates (2020).

A simpler, free alternative IDE for R is RKWard, which might be easier for novice programmers to adapt to than RStudio.[18] It features a graphical user interface with sepa-

[17] https://posit.co/products/open-source/rstudio

[18] https://rkward.kde.org

rate windows for script, data, the R Console, and graphics output, among others. It has a live preview for R Markdown documents, and it stores a history of plots created within the same session. Some of its functionalities are limited compared with similar features in RStudio, including debugging and application customization. In summary, RKWard offers a simpler, lower-overhead environment appropriate for straightforward analyses and learning, whereas RStudio is better suited for more experienced programmers and more complex projects or workflows.

Python

Unlike R, which is a **domain-specific language** for data analysis and data visualization, Python is a **general-purpose language** applicable in many different areas. Similar to R, Python is also both open source and available for free. It was developed by Guido van Rossum in 1989 as an open-source programming language (Severance, 2015) and it is supported today by the nonprofit organization Python Software Foundation.[19] The same group also hosts the Python Package Index,[20] a repository for Python packages, all freely available, and other resources.

Python supports objective-oriented, procedural, and function programming methods, again like R, but it has more comprehensive capabilities for creating and deploying applications. In a survey of advanced analytics software used most often in data science, statistical analysis, machine learning, and business or predictive analytics, Muenchen (2015) reported that (1) R was among the top software that can be used as a collection of prewritten functions and (2) Python was among the most frequently used general-purpose languages in analytics. In the 2023 *IEEE Spectrum* rankings of the popularity of programming languages in various sectors including data analytics, Python obtained the highest rank (Cass, 2023), a position held in these rankings since 2017.[21]

Python scripts can be executed in batch-processing mode much like sourcing an R script. Alternatively, the researcher can work interactively with Jupyter Notebooks inside JupyterLab, an open-source, web-based IDE maintained by the non-profit Project Jupyter. JupyterLab supports dozens of languages (including Python and R) and is freely available,[22] but its most advanced features and community support are centered on Python. A Jupyter Notebook is a single document composed of two cell types:

1. Code cells that can be executed on demand and whose output (plots, tables, errors, etc.) appear inline.
2. Markdown cells that support formatted text, hyperlinks, and embedded LaTeX for typesetting equations or special symbols.

[19] https://www.python.org/psf-landing

[20] https://pypi.org/

[21] https://spectrum.ieee.org/the-top-programming-languages-2025

[22] https://jupyter.org

Notebooks can be shared and accessed by multiple users, and changes can be synchronized in real time when notebooks are hosted on platforms, such as Google Colab, that support collaborative editing and notebook sharing.[23] Jupyter Notebooks are analogous to R Markdown documents in that both support reproducible computing, including the generation of documents that are dynamic or interactive.

The Python language and programming environment make it well suited to a wide range of data science tasks. Like R, Python offers many specialized libraries for machine learning, data manipulation and filtering, optimization, signal processing, Monte Carlo simulation, predictive modeling, and natural language processing (Behrman, 2021; Vasiliev, 2022). Researchers sometimes ask whether they should learn Python in addition to R. Because R is tailored for statistical analysis and Python is a general-purpose language, the decision hinges on the researcher's objectives. For instance, behavioral science researchers with no plans to develop web applications may find it more efficient to strengthen their knowledge of R rather than invest the extra time needed to become proficient in Python (i.e., it is overkill).

Five relative advantages of R over Python for students and researchers in the behavioral sciences (Luna, 2022; Makowski, 2020; Matloff, 2023) are summarized next:

1. *Low barrier to entry:* The function-centric nature of R means that users can perform complex analyses without mastering object-oriented programming, elaborate data structures, or advanced control flows that Python typically requires as a general-purpose programming language.
2. *Familiarity for traditional statistics:* Although Python may feel more "natural" to those with a computer-science background, behavioral scientists trained primarily in statistics might find R's syntax and workflow more intuitive.
3. *Purpose-built for statistics and visualization:* The R environment was specifically built for data analysis and visualization, and its core packages deliver comparatively stronger functionalities in these areas than Python's more general-purpose libraries.
4. *Efficient handling of very large data sets:* While Python execution can outpace R in tasks such as looping, optimized data structures in R can often outperform Python when analyzing data sets with thousands or millions of rows or cases.
5. *More uniform system for packages:* The R environment has a canonical structure for packages, which means that packages are installed and used in more consistent ways in R compared with Python. The broader environment in Python offers more choices but can involve greater configuration and compatibility challenges.

Makowski (2020) offers sage advice for researchers with little formal programming experience: A common—but glib—answer to the question of whether to learn R or Python is "just learn both." While easy to say, that recommendation is often unrealistic. Rather than juggling two languages at a superficial level, it is more effective to gain expertise in

[23] https://colab.research.google.com

one. For most behavioral science researchers, R is the optimal choice given its advantages over Python.

Commercial versus Free Software

Open-source and freely available software for statistical analyses and core data science techniques were emphasized to this point. In addition to their costs (zero), their source code can be directly inspected for computer security issues, and some organizations that support open-source development, such as GitHub, offer bounty programs that reward developers or researchers for identifying and fixing vulnerabilities.[24] Despite these advantages, open-source packages carry potential risks. Many R packages are maintained by individual researchers or small teams, which makes them vulnerable to abandonment when a project ends or a team disbands. Without ongoing upkeep, packages built for older versions of R may fail under newer releases. Support resources vary widely: some packages come with extensive vignettes, forums, articles, or even entire books, while others offer little guidance. Licensing can also pose challenges. For example, Wolter et al. (2023) reviewed 1,000 open-source GitHub repositories and found that about half did not fully declare all licenses in their code, and roughly 10% exhibited conflicting terms of use.

With those considerations in mind, let's briefly examine two commercial software platforms that are widely used in data science. The MATLAB (MATrix LABoratory) platform combines a domain-specific, matrix-oriented programming language with extensive capabilities for linear algebra, data analysis, visualization, and simulation.[25] Although its language supports object-oriented features, MATLAB's syntax and functionality are optimized for matrix and array operations. It offers built-in procedures for core data science techniques, including machine learning and neural networks, and its functionality can be extended through commercial toolboxes that are analogous to R packages but require additional licenses. Applications of MATLAB in machine learning are detailed by Ciaburro (2024).

The Wolfram Language is the proprietary programming language in Wolfram Mathematica, a comprehensive environment for numerical computation, data analysis, visualization, and symbolic manipulation, which allows output to be equations or functional expressions, not just numbers.[26] The Wolfram Language natively supports data science techniques that include machine learning, neural networks, and semantic text analysis, among others. Basic usage is available at no cost, for example, via the Wolfram Cloud's notebook interface (free tier) or the Wolfram Programming Lab for students,[27] while access to all its functionalities requires paid licenses. For a detailed treatment of machine learning in the Wolfram Language, see Bernard (2021). Wolfram|Alpha is a computational

[24] https://bounty.github.com

[25] https://www.mathworks.com/products/matlab.html

[26] https://www.wolfram.com/mathematica

[27] https://www.wolframcloud.com

knowledge engine that responds to natural language queries, and responses consist of text-based answers with relevant visualizations drawn from curated academic and commercial websites.[28] Its calculations are based on algorithms and data models implemented in the Wolfram Language. While Wolfram|Alpha offers built-in machine learning capabilities, these are predefined algorithms tailored to specific tasks. A free online tier provides limited functionality, and full access to advanced features requires a professional license.

Practical Programming Tips for Applied Researchers

Summarized in Table 8.4 are Mineault's (2023) real-world suggestions for researchers who write substantial code without formal training. At the outset of a project, it's common to feel uncertain—or even like an impostor—about one's programming skills. That's okay, just bear down, and remember that research projects often involve uncertainty on many levels even when programming needs are minimal. Begin by defining the computing environment, installing essential libraries or packages, and setting up a version-controlled repository, such as on GitHub, to organize and share code and data. Above all, document rigorously: detailed comments and project notes not only guide collaborators—students or colleagues not involved in the analysis—but also help the researcher to recall their own rationale long after the code was written.

Continuing with the fourth point in Table 8.4, establish a consistent coding style: use uniform indentation, align parentheses, square brackets, and curly braces, and adopt clear, descriptive names for variables and functions. Limit each line to about 80 characters to reduce horizontal scrolling and improve readability. Structure the code into small, modular units, each of which carries out just a few tasks in ways that are relatively independent of other code units. This practice is called **decoupling**, and it makes it less likely that changing one code segment will break another and simplifies debugging. Likewise, keep functions concise (roughly 40 lines by 80 columns) so they can be quickly scanned and their purpose understood (Mineault, 2023).

For projects that span multiple script files, packages or data files, adopt a version control system (point 6 in Table 8.3). **Version control** software tracks every change to the code and other project assets, which is essential when collaborating with others. It keeps a full history so the researcher can roll back to a known working state if a recent edit introduces errors. Free, open-source distributed systems—such as Git, Mercurial, and Apache Subversion—are widely available.[29,30,31] Finally, pair version control with rigorous testing: rerun the test suite after every change, no matter how small, to confirm that nothing breaks. See the introduction to software testing by Ammann and Offutt (2017) for more suggestions. In R, it is a best practice before running tests to clear the **workspace**,

[28] https://www.wolframalpha.com

[29] https://git-scm.com

[30] https://www.mercurial-scm.org

[31] https://subversion.apache.org

TABLE 8.4. Suggestions for Researchers Not Formally Trained in Programming

Suggestion	Elaboration
1. It is natural to feel like one does not know what they are doing at first	New research projects often involve uncertainty, so doubts about programming skill are no different (i.e., it will work out)
2. Plan and organize the project	Select the programming environment, install relevant packages, and initialize a repository
3. Comment and document (really) both the code and project	Explain code both in the syntax files with comments and also in external documents, especially if syntax or analyses are relatively complex
4. Keep things neat, and back it up	Adopt consistent variable naming and indentation, formatting of syntax, ask others to review the code for understandability, and make regular backups
5. Write modular, decoupled code	Problems are easier to diagnose when code is organized into smaller units, each of which carries out a small number of things relatively independent of other units
6. Use version control	Version control software maintains a history of changes to syntax or other files and also permits restoring earlier versions
7. Test the code—again	Test the code after making any change, however, small, and test new functions with a variety of cases or scenarios
8. Make coding social, not solitary	One of the best ways to become a better programmer is to learn from those who are even better programmers or can give constructive feedback

which is the collection of all user-defined objects (variables, data frames, functions, and loaded packages) that exist in the current R session. It represents the state of the interactive environment and persists unless explicitly cleared by the user; otherwise, hidden or "stale" objects can interfere with diagnosing problems. For example, a data frame left in memory might be automatically used instead of loading an updated version with the same name. This compromises reproducibility and makes errors harder to isolate or reproduce.

Mineault (2023) reminds that us coding thrives as a collaborative endeavor rather than a solitary one and that partnering with more skilled programmers is one of the fastest paths to improve one's own abilities (Table 8.4, point 8). **Pair programming**, whether side by side in person or over a remote connection, creates a live feedback loop, as teammates alternate between writing code and walking the other through their reasoning. Another way to promote social aspects of programming is to hold regular code or design reviews with other team members. Participating in online communities for programmers is a good option for researchers who work in a smaller team or who is the only person in the group that writes code for the project.

Summary

Kramer (2021) identifies the core qualities of a strong programmer as mathematical aptitude (matched to project demands), acute problem-solving skills, excellent organization and time management, precision and attention to detail, clear communication, and the ability to collaborate effectively. These same attributes underpin successful researchers, making those with limited formal computer science training excellent candidates to develop greater programming expertise. Researchers can build these skills through focused self-study, short, intensive bootcamps offering hands-on experience, or longer-term coursework leading to certification or a degree. It is critical for researchers to match their current skill level to a project's needs—break a complex task into modules so that expert-level components can be outsourced or assigned to consultants, while the researcher handles the rest. Starting with R is often effective because it lets researchers immediately leverage powerful data analysis capabilities with minimal prior programming experience, yet its object-oriented design provides a path to more advanced coding. In the next chapter, we examine various approaches to data visualization.

EXERCISES

1. The built-in R data set `iris` has measurements for three Iris species. Describe the characteristics, structure, and attributes of these data in terms of variables or factors. Display the first and last five cases of the data set in the R console. What does the syntax `data()` generate?

2. For the `iris` data set, create a new variable `Sepal.LW` that is the sum of sepal length and width and add to those data. Generate whole-sample descriptive statistics for all variables in the modified data frame.

3. Describe the `cars` built-in data set in R. Conduct the bivariate regression of stopping distance on speed, extract to external variables the unstandardized slope and intercept, and add the residuals to the data frame.

4. Adapt syntax in Appendix 8.C to download and describe the data set

   https://filesamples.com/samples/document/csv/sample3.csv

   Display the first five cases in the downloaded data.

5. Write syntax that generates the integers 1–10, computes their natural logarithms (base *e*) and factorials, and saves the original and computed values in a data frame that is displayed in the R console.

FURTHER READINGS

The e-book by Crump (2018) introduces programming in R, HTML, and Javascript using methods that make these programs accessible to psychologists and other behavioral scientists. Pędziwiatr (2018) offers additional suggestions for novice programmers beyond those listed in Table 8.4. The book by Vickers (2008) is not about syntax or any specific programming language. Instead, it covers essential problem-solving skills and strategies applied by programmers to handle complex problems.

Crump, M. J. C. (2018). *Programming for psychologists: Data creation and analysis*. https://www.crumplab.com/programmingforpsych/

Pędziwiatr, M. A. (2018). Hints and tips: Computer programming for psychologists. *PsyPag Quarterly*, *1*(108), 38–40.

Vickers, P. (2008). *How to think like a programmer: Problem solving for the bewildered* (illustr. ed.). Cengage Learning.

CHAPTER TERMS TO REVIEW

- argument (function)
- array
- autocompletion
- class
- column header
- control structure
- data frame
- data structure
- data type
- decision structure
- decoupling
- domain-specific language
- do–while loop
- edge case
- factor
- for loop
- function
- functional language
- general-purpose language
- if–else block
- list
- loop structure
- Markdown scripting language
- matrix
- object
- object-oriented programming
- package (R, Python)
- pair programming
- procedural language
- repeat loop
- Tufte-style document
- vector
- version control
- while (pretest) loop
- workspace (R)

~ **Appendices 8.A–8.E** with annotated R syntax are freely available on this book's resource website at www.guilford.com/kline3-materials.