Ten simple rules for selecting an R package

Caroline J. Wendt [1, 2], G. Brooke Anderson [3] *

1 Department of Statistics, Colorado State University, Fort Collins, Colorado, United States of America
2 Department of Mathematics, Colorado State University, Fort Collins, Colorado, United States of America
3 Department of Environmental & Radiological Health Sciences, Colorado State University, Fort Collins, Colorado, United States of America

* Corresponding author: Brooke.Anderson@colostate.edu

Abstract

R is an increasingly preferred software environment for data analytics and statistical computing among scientists and practitioners. Packages markedly extend R's utility and ameliorate inefficient solutions. We outline ten simple rules for finding relevant packages and determining which package is best for your desired use.

Author summary

Write the author summary here. Do we want to include and author summary?

Text based on plos sample manuscript, see *`http://journals.plos.org/ploscompbiol/s/latex`*

Disclaimer?

Do we need to include a disclaimer in the margin like the one from [1] that states: "**Competing Interests**: The authors have no affiliation with GitHub, nor with any other commercial entity mentioned in this article. The views described here reflect their own views without input from any third party organization."

- RStudio
- ROpenSci
- GitHub

I am an editor at ROpenSci, so we could mention that in the disclaimer. I do not have any financial interests from that position (it's volunteer, as are many journal editing positions in academics).

Funding acknowledgment

[Funding acknowledgement—Add in grant number for R25 and acknowledgment of Honors program if appropriate]

Introduction

Computational reproducibility is surfacing as a central axiom in academia, as researchers identify the need for transparency [2,3]. Many traditional methods also tend to be at odds with productivity and collaboration; some variability in scientific outcomes can be attributed to differences in workflow. Today, the scientific community deems the absence of automation to be irresponsible [4]. While standards for disseminating and communicating computational science are evolving with digitization, responsible researchers are adopting best practices so others can more easily interpret, validate, and extend their published knowledge [5]. Such researchers must therefore rely on accessible and robust tools to ensure that their computations are reproducible.

The open source R language has become a dominant quantitative programming environment in academic data analysis, enabling researchers to share workflows and re-execute scripts within and across subsets of the scientific community [4]. R is increasingly popular in computational biology and bioinformatics, two of many disciplines generating extensive, heterogenous, and complex data wanting for heavy-duty data analysis tools that (ideally) support reproducibility [6,7]. More broadly, as the R ecosystem—in which the life of modern data analysis thrives—rapidly evolves alongside the burgeoning R community, R is exhibiting sustained growth when compared to similar languages, particularly in academia, healthcare, and government [8].

R was developed by statisticians and is collaboratively maintained by an international core group of contributors [9]. Unlike several popular proprietary languages (e.g., MATLAB, SAS, SPSS), R is highly extensible, free and open-source software; the user can access and thus change, extend, and share code for desired applications. Accordingly, a vibrant community of R users has emerged, many of which engage in the development of extensions to the functionality of base R software known as packages. Contributed packages comprise the bulk of enhancements made to the R environment [10]. In fact, much of what we know about statistical methods and algorithms is wrapped up in R packages—written and documented in various ways by R users. There are plenty of analogies in computing that draw comparisons between programming and culinary arts: recipe structures, coding cookbooks, and the like. To conceptualize packages, imagine you are the chef, base R is the kitchen, and packages are the special gadgets which allow you to cook and bake new recipes. “R package” *technically* refers to a collection of R functions in a certain structure. Thus, the few tools that come with your kitchen (e.g., `stats`) are indeed R packages, while “R extension” describes the tools you add to your kitchen. It is important to recognize that “package” is not synonymous with “extension”; however, since we don’t want to keep distinguishing a kitchen sink from a mixer, we use “package” in the colloquial sense. In essence, R packages are coding delectables that enable the user to perform practical tasks and solve problems with interesting techniques.

Are there R packages for wrangling and cleaning data frames, designing interactive applications for data visualization, or performing dimensionality reduction? Yes! How do you *find* an R package that will help you train regression and classification models, assess the beta diversity of a population, or analyze gene expression microarray data? This answer is not as simple; there are tens of thousands of R packages. As a natural consequence of the open source nature of R, there is considerable variation in the quality of R packages and nontrivial differences among those that provide similar tools.

Packages are essential to venturing beyond base R and, thus, quickly become an integral aspect of advancing your R skills. Those who have used R packages may know that, although leveraging existing tools can be advantageous, the initial challenge of finding a suitable package for a given task can obstruct potential benefits. The advanced R user—having developed an intuition for their workflow—may be relatively confident when searching for and selecting packages. By contrast, new R users who are unfamiliar with the structure and syntax of the language may be hindered by the process of finding packages because they do not know where to search, what to look for, or how to sift through options. An obstacle that characterizes learning R at the outset is the struggle to (1) find a package to accomplish a particular task or solve a problem of interest and (2) choose the best package to perform that task. Even so, some obscure and complicated recipes make it difficult for an experienced chef to select the best tools.

In both coding and life, we endeavor to make choices that optimize outcomes. Just as one may go about shopping for shoes, deciding which graduate program to pursue, or conducting a literature review, there is a science behind selection. We inform our decisions by assessing, comparing, and filtering options based on indicators of quality such as utility, association, and reputation. Likewise, choosing an R package requires attending to similar details. We outline ten simple rules for finding and selecting R packages, so that you will spend less time searching for the right tools and more time coding delightful recipes.

Rule 1: Consider your purpose

Usually, there are several ways to accomplish a task while programming, albeit some more elegant and efficient than others. Before looking for a package to use for a task, consider whether you need one. Consider your purpose by first identifying your goal and defining the scope and tasks to achieve it. For some tasks, coding your own recipe with existing tools is practical, while other tasks benefit from new tools.

If the scope of your task is simple or reasonable, given your knowledge and skills, using an R package may not be appropriate. There can be advantages of coding in base R to complete your task or solve your problem. First, when you code from scratch, you know precisely what you are running; thus, your script may be easier to decipher and maintain over time. Conversely, packages require you to rely on shared code with features or underlying processes of which you may not be aware. Second, while base R is relatively stable and slow to change, many R packages are rapidly evolving, and changes in packages you depend on can "break" parts of your code when they change.

Packages are favorable in the same sense as kitchen tools: when a task has a broad or complex scope beyond what you can (or desire to) attempt from scratch. While there are ways to cook up an algorithm using `for` loops and conditionals in base R, a relevant package may accomplish the goal with less code and fewer bugs. The more reasonable it is for a given task to be abstracted away from its context, the more likely someone has generalized its themes, developed efficient algorithms, and intuitively organized them to share with other R users. Extensive tasks justify sophisticated frameworks with several functions that form a cohesive package or even a suite of related packages. Data manipulation is one such common task that has been streamlined by packages such as `dplyr` and `tidyr` [11,12], both part of the `tidyverse` suite (see Table 1). Nevertheless, don't be discouraged if you have a task that seems too unusual for a package. There are indeed packages for seemingly singular tasks, which you may favor over coding from scratch. For example, there is a package for converting English letters to numbers as on a telephone keypad [13].

If you do decide you need a package, next ask yourself: Which functionalities in base R are restrictive in the context of your task? Which new functionalities would expand what you can do? If you identify limitations of your current toolbox before searching for new tools, you be primed to recognize what you do and do not need. List domain-specific keywords on what you are trying to do and how you could do it, so you can narrow your search. Identify the type of inputs you have and envision working with them; contemplate the desired outputs and corresponding format. For instance, suppose you are using Bioconductor packages in analyses and have outputs you want to visualize;

you must consider that the inputs may be of a certain class—namely, S4 objects—which impose restrictions when creating graphics [14]. When you pick a package to visualize the data, you want to pick one that addresses such restrictions.

Rule 2: Find and collect options

Tips and leads on R packages exist in a variety of places online, in print, and elsewhere. While long-time R users have found some favorite starting spots, new R often don't know where to start looking for a good package for a certain task. You can discover new packages any time you learn R-related topics, browse the internet, or tap into the international community of R users.

Learn

When learning how to program in R, you are typically introduced to some of the most common packages, which tend to have more general purposes (**Table 1**). In addition, reputable online tutorials, courses, and books are helpful resources for acquiring knowledge about packages that are versatile and reliable—many of which are short, accessible, and either affordable or offered at no cost to the learner. We recommend online R programming courses such as those through Coursera and Codeacademy for interactive learning and R book series including the RStudio books and Springer titles for further reading.

Browse

The solution-seeking tactics we employ for many tasks nowadays may lead you to think that finding R packages relies heavily on internet search queries. Indeed, search engines such as Google return ample pages related to anything `"...in R"`. However, this approach can lead to frustration and confusion when attempting to find a package tailored to your purpose (see Rule 1). You can also do a more directed search by searching through package lists for repositories including Comprehensive R Archive Network (CRAN) and Bioconductor, or through packages available in code-sharing sites like GitHub and GitLab, all of which will be further discussed in Rule 3.

In many cases, you may find it more helpful, though, to start from a curated list of R packages on a particular topic. There are several available. First, CRAN Task Views are concentrated topics from certain disciplines and methodologies related to statistical computing that categorize R packages by the tasks they perform (e.g., Econometrics, Genetics, Optimization, Spatial). In the HTML version, you can browse alphabetized subcategories within each Task View and read concise descriptions to find tools with specific functions. Alternatively, you can access Task Views directly from the R console with `ctv::CRAN.views()`. To date, there are 41 Task Views that collectively contain thousands of packages which are curated and regularly tested. Moreover, CRAN Task Views provide tools that enable you to automatically install all packages within a targeted area of interest. Ultimately, by providing task-based organization, easy simultaneous installation of related packages, meta-information, ensured maintenance, and quality control, CRAN Task Views address several major user-end issues that have arisen due to the sheer quantity of available packages [15].

[Add paragraph on Bioconductor Task View-equivalent]

Other curated collections or topically-linked lists of packages are also available. CRANberries, for example, is a hub of information about new, updated, and removed packages from the CRAN network. Another place to find well-maintained tools is through rOpenSci packages. These filterable and searchable R packages are organized by name, maintainer, description, and status (i.e., activity, association, review), and all have passed a peer review process (see Rule 6).

Use the community

An inclusive and collaborative community is an overlooked, yet integral, aspect of a software's success [16]. A defining feature of R is the enthusiasm of its users and developers alike. The R community has a widespread internet presence across various platforms; however, members are markedly active on Twitter, a place where R users seek help, share ideas, and stay informed on `#rstats` happenings, including releases of new packages [17]. R-related blogs serve as another informal, up-to-date, and more detailed avenue for communicating and promoting R-related information (**Table ?**). For example, Joseph Rickert, Ambassador at Large for RStudio, writes monthly posts on the R Views blog highlighting interesting new R packages. Rickert also features special articles about recently released packages and lists of top packages within certain categories, including Computational Methods, Data, Machine Learning, Medicine, Science, Statistics, Time Series, Utilities, Visualization.

You can also attend—or re-watch—conferences to learn about recent R package developments and applications. Two large annual R conferences are rstudio::conf for industry and useR! (not exclusively) for academia. Talks and presentations at both these conferences are often recorded and made available online for playback if you can't attend in person or wish to revisit past conferences. Conferences in your field may foster connections with fellow scientists who use R for similar tasks and help you collect information about packages related to your expertise.

Finally, a developer of an R package may intend for it to be private (exclusively for personal or professional use) or public (at no cost and available for use by anyone) [18]. If your task is specific to a line of research, consult colleagues to see if they have a relevant (private) package they would be willing to share.

Rule 3: Check how it's shared

Packages can be shared through a variety of platforms. Repositories are a primary way in which developers share packages for both public and private use, but there are alternatives. As far as R packages are concerned, a repository is essentially a warehouse for tools before they enter your kitchen; in computing terms, a repository is analogous to a cloud because it is a central location in which data is stored and managed. Some repositories impose vetting mechanisms that tame unwieldy aspects of the R ecosystem by regularly checking underlying code and managing corresponding webs of dependencies. Two traditional repositories for R packages are CRAN and Bioconductor; however, there are lesser-known remote repositories that have unique properties. Developers can also share their R packages through large code-sharing sites, including public version control platforms like GitHub and GitLab; these have fewer restrictions on the format or content of shared code compared to repositories. At the least public level, in lieu of making packages accessible to everyone via internet repositories, some developers share their code in zipped files directly with collaborators (see Rule 2).

The CRAN package repository is the most established and primary source from which you can install R packages. CRAN hosts a huge collection of R packages ([current # of CRAN packages] as of August 2020) on almost any conceivable topic, from [example] to [example]. Due to its longevity and historical role, Rickert asserts that "CRAN is the greatest open source repository for statistical computing knowledge in the world" [18]. A key advantage of picking a package from CRAN is that the repository integrates very easily with a user's base R installation. You can simply use the `install.packages()` function to install a package from CRAN; source and/or binary code are automatically saved to your computer in a designated package library [19]. When you want to use a particular package, you load it to your R session via the `library()` function from base R. The R Foundation manages CRAN and imposes strict regulatory practices for the selection and maintenance of the packages they host. A package must pass a series of stability tests in accordance with the CRAN Repository Policy before obtaining publication privileges [20]. As such, CRAN only considers packages that make a substantial contribution to statistical computing and graphics. CRAN maintainers actively monitor source and contributed packages to ensure they are compatible with the latest version of R and modify or remove packages that do not uphold publication quality.

If you are looking for tools related to high-throughput genomic data, the Bioconductor repository is a more specialized repository than CRAN that is worth investigating. The Bioconductor project was motivated by a need for transparent, reproducible, and efficient software in computational biology and bioinformatics and supports the integration of computational rigor and reproducibility in research on biological processes [6]. Bioconductor packages facilitate the analysis and comprehension of biological data and help users solve problems that arise when working with high-throughput genomic data such as those related to microarrays, sequencing, flow cytometry, mass spectrometry, and image analysis. The R environment and its package system is fundamental to the implementation of Bioconductor's interoperable and object-oriented (S4) infrastructure. Bioconductor software is in the form of coordinated, peer-reviewed R packages. Bioconductor boasts a modularized design, wherein data structures, functions, and the packages that contain them have distinct roles that are accompanied by thorough documentation. Similar to CRAN, Bioconductor has strict criteria for package submissions: a package must be relevant to high-throughput genomic analysis, interoperable with other Bioconductor packages, well-documented, supported in the long-term, exclusive to Bioconductor, and comply with additional package guidelines [21].

There are also more specialized, smaller repositories that share R packages. The non-profit organization, rOpenSci, for example, runs a repository as part of their commitment to promote open science practices through technical and social infrastructure for the R community [22]. The repository only includes packages that have passed their open review process and that are within a scope that ROpenSci defines and that focuses on . . . . R package maintainers can even create and host their own personalized repository using the `drat` package [add citation to drat package]. In some cases, this might be done to host a package that does not meet certain restrictions from other repositories (e.g., to share data through a package that is larger than CRAN's usual size limit on packages [add citation to "Hosting Data Packages via drat: A Case Study with Hurricane Exposure Data" paper by me and Dirk Eddelbuettel]). There are other platforms strictly for the development, rather than distribution, of R packages such as R-Forge and Omegahat, which are beyond the scope of this paper [23,24].

The rapid uptick in package development and subsequent inter-repository dependencies has sparked an ongoing debate on whether regulated repositories such as CRAN and Bioconductor are preferable to other distribution platforms, namely public version control systems like GitHub [18,25,26]. While there are practical downsides to their restrictive practices, the benefits of exclusive repositories are evident. Nevertheless, there are considerable advantages to hosting a package on GitHub [18,25]. GitHub is a popular online user interface and multi-purpose development platform that is also effective in distributing R packages. An increasing number of packages are hosted on GitHub during the development stages; if developers choose not to distribute their package through GitHub, the stable release versions of such packages are often published on CRAN or Bioconductor [27]. GitHub provides R users with open access to package code, a timeline of help resources (see Rule 4), a direct line of communication to developers, and permits discovery of up-and-coming packages (see Rule 2). You can install the latest version of packages from GitHub via `devtools::install_github()`; however, the decentralized nature of GitHub is not conducive to a tool that automatically locates and installs corresponding dependencies [28]. For developers, GitHub provides a convenient means by which anyone can share and contribute public or private code without barriers to entry. Authors collaborate within a version-controlled system to develop and distribute packages, including those with dependencies that are not on CRAN or Bioconductor. Further, the `drat` (Drat R Archive Template) package enables developers to design individual repositories and suites of coordinated repositories for packages that are stored in and/or distributed through GitHub [29,30]. Both R users and package developers benefit from interactive feedback channels through GitHub Issues and the Star rating system. Further, GitLab is git-based version control and collaborative cloud for package production and deployment. It is an alternative to GitHub for production of large-scale packages that require continuous integration and continuous deployment for testing data and code to ensure a stable end-product.

Finally, a single package is often available in multiple places.

Rule 4: Explore the availability and quality of help

There has been a call for the development of centralized resources in statistical computing to enable a common understanding of software quality and reliability: software information specified in publications, domain-specific semantic dictionaries, and a single metadata resource for statistical software [10]. No such resources have been consolidated to serve these purposes and, given the decentralized nature of today's information society, it is questionable whether they will emerge. Current sources of information related to R packages are dispersed and plentiful. On one hand, this allows users to explore diverse solutions and discover new tools; on the other, not knowing where to find help can lead to inefficient and ineffectual roundabouts. Clearly, not all package resources share the same level of quality and the fact that there are many resources in aggregate does not imply that every package is associated with the same availability of resources. While all R packages warrant some minimal standard of documentation, beginners and users of complex packages might desire more.

You can access information about R packages along with an index of help pages from the console via `help(package = "...")`. Package information will vary; ideally, packages should have thorough documentation, but at minimum, every R package should include a `DESCRIPTION` file with metadata. The `DESCRIPTION` is a succinct record of the package's purpose, dependencies, version, date, license, associations, authors, and other technical details. The help pages feature information about the structure of functions within the package and contain executable examples to demonstrate the relationship between various inputs and outputs. If the `DESCRIPTION` and help pages alone leave you wanting, the package likely does not have further (quality) documentation and therefore should not be your first choice, if comparable options exist. In short, if the developer cannot initially communicate how their tool works, then you may not want to use it in your kitchen (read: if the instruction manual is useless, do not use the blender).

Fortunately, plenty of R packages include additional documentation beyond mere descriptions. The documentation that accompanies functions within packages is critical; the fact that anyone can read the documentation anytime and use it to guide their own work facilitates extensibility. RDocumentation is a searchable website, package (`install.packages("RDocumentation")`), and JSON API for obtaining integrated documentation for packages that are on CRAN, Bioconductor, and GitHub. This is a subsidiary reason why packages shared on these platforms tend to be superior (see Rule 3). RDocumentation may include: an overview, installation instructions, examples of usage, functions, guides, and vignettes. Most software documentation is rather technical and extraneous to new users whereas a vignette is a practical type of documentation in the form of a tutorial. A vignette is a detailed, long-form document that describes the problems an R package can solve, then illustrates applications through clear examples of code with coordination of functions and explanations of outcomes. Packages can have multiple vignettes; you can view or edit a specific vignette or obtain a list of all vignettes for a package of interest via the `vignette()` function.

Some packages are branded quite well and include a comprehensive set of resources. Implicitly, this indicates that the authors are at least serious about their package development, which may lead you to infer that they know what they (and their package) are doing. Exemplary documentation can signify an exceptional package. For instance, some packages have websites and/or books. One popular method that developers use to publish books about their package is through `bookdown`, a relatively new extension of R Markdown that is structured in such a way that integrates code, text, links, graphics, videos, and other content in a format that can be published as a free, open, interactive, and downloadable online book [31]. The `bookdown` package itself has an online book that details usage of the package [32]. `Rccp` is another package with notable documentation and first-rate help resources; the developers maintain both a main and additional website with a wealth of organized information about the package and resources, including examples, associations, publications, articles, blogs, code, books, talks, a mailing list, and links to other resources with `Rccp` tags. Aside from the documentation and resources from the developer, further information about some R packages is available in video tutorials, webinars, and code demonstrations (i.e., "demos"). As of RStudio v1.3, you can access tutorials powered by the `learnr` package from the Tutorial pane in the IDE [33]. Finally, keep in mind that RStudio creates cheatsheets of concise usage information for popular packages through code and graphics organized by purpose. Cheatsheets can be accessed directly via the RStudio Menu (Help > Cheatsheets) or from the RStudio website on which you can subscribe to cheatsheet updates and find translated versions.

While using a package, anticipate complications beyond the scope of documentation. In this case, you will use resources that involve *asking* for help—should the occasion arise, you want to be assured that you will find a satisfactory answer. In the past, the antiquated R-help mailing list was the only way to seek assistance; since, the R community has formed, with inclusion and creative problem-solving as hallmarks of its online presence [34]. The modern R-help mailing list to which you can subscribe and send questions is moderated by the R Core Development Team and includes additional facets for major announcements about the development of R and availability of new code (R-announce) and new or enhanced contributed packages (R-packages) [35]. Certain packages have independent listservs; `statnet` is an example of a suite of packages that has its own community listserv. If a package has a development repository on GitHub, check the Issues to verify that the maintainer is responsive to posts and fixes bugs in a timely manner. In addition, you can search discussion forums such as Stack Overflow, Cross Validated, and Talk Stats to assess the activity associated with the package in question. Analyses of the popularity of comparable data analysis software in email and discussion traffic suggest that R is rapidly becoming more prevalent and is the leading language by these metrics [8,36]. When you encounter a problem, it is good practice to first update the package to see if the problem is due to a bug in a previous version—if the problem persists, seek help by finding or posting a reproducible example [37]. Overall, avoid using a package if the quality and quantity of related resources is lacking.

Rule 5: Quantify how established it is

Consulting data to inform comparisons is never a bad idea; numerical data associated with R packages will give you an impression of how regarded the tool is and whether it has stood the test of time. Since there are tens of thousands of R packages, you may be wondering how they stack up in terms of popularity. On GitHub, a large number of Stars, Forks, and Watchers associated with a package implies a substantial following and widespread usage [38]. Likewise, the number of Google Scholar citations is a metric of a package's impact on scientific research and utility in research contexts (see Rule 2). RDocumentation (see Rule 4) is rich with stats on R packages. RDocumentation hosts a live Leaderboard with trends including the number of indexed packages and indexed functions, most downloaded packages, most active maintainers, newest packages, and newest updates. What's more, each package is assigned a percentile rank—featured on its RDocumentation page—that quantifies the number of times a package has been downloaded in a given month. A ranking algorithm computes the direct, user-requested monthly downloads by accounting for reverse dependencies (indirect downloads) so packages that are commonly depended upon, and hence frequently downloaded, do not skew the calculation [39]. You can research stats on corresponding dependencies for a more holistic picture. To further determine if a package is well-established in the R community, refer to the number of versions and updates (more is better) as well as the date of the most recent versions and updates (newer is better).

Rule 6: Seek evidence of peer acceptance and review

Peer review is an important aspect of scientific research, not least because it establishes scholarly credibility. You can research information about an R package in different forms of literature and determine the extent to which it has been validated by the scientific community. Some journals publish articles about R packages themselves while others feature work that used a particular package. Literature in your field may either introduce R packages developed to solve a unique data science problem or mention packages used during the research process. The former may be published in the *Journal of Statistical Software*, *The R Journal*, or *BMC Bioinformatics*, for example, and search queries that include `"R package"` with domain keywords will narrow results. The latter requires identifying authors who used R in their analyses; useful packages may be mentioned in the Methods and/or References sections. You can search for packages directly by name in Google Scholar: the `Cited by` link displays the number of times a package has been cited and connects to a page with those publications.

These packages are technically sound and have made a substantial contribution to their fields and/or a common data science problem. In response to the rising number of researchers creating tools and software to work with their data, GitHub has granted developers the ability to obtain a Digital Object Identifier (DOI) for any GitHub repository archive so that code can be cited in academic literature [40]. If a package has such a DOI, you can explore the network of research associated with that package. Many R packages are associated with content in books and series from scientific publishers such as Springer. More directly, rOpenSci, is a unique example of an ecosystem of open source tools with peer reviewed R packages (see Rule 3) [22].

Rule 7: Find out who developed it

Just as research is a library of shared insight, open source software is a collection of shared tools. We care about who writes the articles we read; we should also care about who creates the tools we use. Although R is grounded in statistical computing and graphics, there is variation R users' backgrounds and skills, and the same is true for R developers. That said, the R community prides itself on embracing newcomers at all levels of involvement. This Rule does not imply that worthwhile packages are exclusively written by well-known authors. Rather, associations and reputation can be a proxy for quality; in this way, the process of evaluating and comparing R packages is no different than other decisions. In fact, as you become more immersed in the R community, you will find that name recognition is a factor, among many, that helps you establish trust in certain tools more quickly than others [38].

You can assess the credibility and commitment of R package developers through direct and indirect signals. Who made the package? Consider whether expertise in a certain domain is vital to the design and creation of the tool. Research the authors' associations in academia, industry, and/or laboratories and gauge the extent to which they have a primary role in R development. Further, you can learn more about their experience, active contributions to the R community, and history related to package development by exploring their profiles on GitHub, Google Scholar, Research Gate, Twitter, or personal or package websites. If an author has such a history, peruse their portfolio of packages to see if any are highly regarded or recognizable. Frequent commits and effective resolutions of GitHub Issues can reveal the authors' priorities and commitment. If the package was developed by multiple authors, research each of them to evaluate the robustness of the team. By extension, these indicators of developer involvement and reputation will help you discern whether a package is worthy of your trust or requires evaluation based on other Rules.

Rule 8: See how it's developed

You do not need to be a software engineer to identify strong package development. Scientific software developers sometimes neglect best practices; indeed, these shortcomings are evident in the tools they create [41]. There are concrete ways to measure a tool's robustness beyond whether it works for those who did not create it. R packages often depend on other R packages; you should check the reputations of such *dependencies* when selecting a package—quality packages will rely on a solid web of quality packages. What's more, like other types of software, well-maintained R packages have multiple versions corresponding to iterative releases to indicate that the package is compatible with dependencies and loyally updated (e.g., bug fixes, general improvements, new functionality) [1,19]. You can explore the version history of a package to see if it is up-to-date. As a user, there are two additional development protocols that you can further investigate to assess the underlying stability and utility of a package: unit tests and version control.

A responsible developer with a consistent and reproducible workflow will implement formal testing on their code to examine expected behavior via an automated process called unit testing [19,42]. Although inconvenient at the outset, the developer—and by extension, the package user—will benefit from unit testing, which results in fewer bugs, a well-designed code structure, an efficient workflow, and robust code that is not sensitive to major changes in the future [19]. To alleviate the burdens of unit testing, `testthat` is a popular, integrative R package that helps developers create reliable functions, minimize error, and visualize progress through automatic code testing [43]. Developers are also interested in quantifying the amount of code in their package that has been tested. Test coverage, a measurement of the proportion of code that has undergone unit testing, is an objective metric for package developers, contributors, and users to evaluate code quality. Many developers use the `covr` package to generate reports and determine the magnitude of coverage on the function, script, and package levels [44]. Relatedly, developers who host their packages on GitHub, post status badges in the overview (`README`) section of the repository webpage. GitHub badges are a common self-imposed method to signal use of best practices and motivate developers to produce a product that is high in quality and transparency [45]. You may see, for example, license, dependency, or style badges, all of which are good indicators of package caliber; however, particular to this Rule, you should look for code coverage (`codecov`) badges which reveal the percentage of test coverage.

As we mentioned in Rule 3, version control has an essential role in package development and computational literacy more broadly [19,46]. Version control is like a time capsule for your workflow because it monitors and tracks changes to files as a project evolves, and stores them as previous versions to be recovered if necessary. In other words, "version control is as fundamental to programming as accurate notes about lab procedures are to experimental science" [[46]; p6]. Git is a decentralized open source version control system that is useful regardless of whether a project is independent or collaborative [47]. GitHub works in conjunction with Git to provide a powerful structured system to organize and manage components of a project for others and your future self. A growing number of scientists have research programs based in GitHub, which has become a revolutionary tool for productive team science and distributed development efforts [1,48]. As you may expect, Git coupled with GitHub is the version control duo of choice among serious R package developers [19]. Thus, if the package you are interested in using is among the thousands hosted on GitHub, this is evidence that the developer is at least committed to a logical, open, and reproducible workflow, suggestive of more time spent designing their tool.

Rule 9: Put it to the test

If you are unable to decide whether to use a package based on prior Rules, test it out. Similarly, if you have narrowed your options, work with each to highlight differences. Exploring the package and engaging in trial and error using your skills in context of your goal will illuminate technical details and solidify any doubts. Note, in the case that the package you want to try has been shared as a zipped file, you can use a GitHub mirror of CRAN via `devtools::install_github("username/reponame")` as an alternative to downloading a large or potentially corrupted zipped file.

At this point, what you have learned about the package should be quite helpful. If the development and documentation are sound, the package should come with a test script or working example that you can run after installation [41]. Vignettes include many common data science problems with solutions; you can run the code examples, tweak them, and compare the outputs. In general, it is essential to know the behavior of different functions within a package, how they interact, and how outputs respond to changes in inputs. Suppose you are testing a package with sparse documentation such that function descriptions often include "`...`" and the argument descriptions seem incomplete. This will be problematic if making a reasonable change to an argument results in an incomprehensible error for which you cannot find help. When this happens, you may not want to use the package for your task.

Sometimes packages do not interact well with other packages; a recipe prepared with an odd combination of tools will not turn out. If you are interested in working with a certain package but are already using other packages in your workflow, you will need to verify that they work together. More precisely, you should check the *interoperability* of all the packages you want to use. A given package may be highly specialized and incompatible with certain packages in general, or simply have a few tolerable quirks for which you can develop workarounds. There are some packages that are masterful at doing what they are made to do, yet incongruous with other packages. Such packages might, for example, use S3 or S4 objects, which are two main approaches developers use to implement object-oriented programming in R. Many packages for spatial analysis as well as those from Bioconductor tend to use S4 objects to represent data [6]. On the other hand, the `tidyverse`, a unified suite of packages with an grammatical structure employed within a "pipeline", expects data frame objects [49]. Thus, when you are working in the tidyverse, you cannot incorporate S3 and S4 objects into the framework unless their corresponding functions are the final step in the pipeline. The broom package, and the bioinformatics analog, biobroom, aim to alleviate these disruptions by converting untidy objects into tidy data, thereby making it easier to integrate statistical functions into the structure of the tidyverse workflow [50,51]. Furthermore, the caret package facilitates interoperability for machine learning packages by providing a uniform interface for modeling with various algorithms from different packages that would otherwise have independent syntax [52].

Rule 10: Develop your own package

Alternative solutions can be sought when a package to solve your data science problem is nonexistent. An R package is the fundamental unit of shareable code; rather than exclusively being a user of packages, you can create them—more easily than you may think [19]. The reasons why you might want to create a package are abundant, including necessity, innovation, standardization, automation, specialty, containment, organization, sharing, collaboration, and extensibility. The essence of an R package is a self-contained piece of statistical knowledge that can be used in combination with other self-contained pieces of statistical knowledge of different shapes and sizes; the uniquely structured functions within a package help us implement that knowledge and weave it into novel scientific work.

Whatever your motivation, packages are simply tools; you can create a package out of any collection of specialty functions. Packages need not be formal nor entirely cohesive. For instance, personal R packages such as Hmisc and broman, are comprised of miscellaneous functions which the creator has developed and frequently uses [53,54]. Functions are necessary for efficiency and warranted when you repetitiously copy and paste your code while making slight modifications after each iteration [37]. The concept of personal R packages demonstrates a unique purpose for packages beyond the conventional. R packages are not solely reserved for specific tasks with comprehensive methods; rather, package development can help you learn how to apply proper coding techniques to writing functions and documentation with reproducibility and collaboration in mind [55].

Although you may not anticipate that anyone else will use your tools, following best practices for package development will yield more favorable outcomes. As a consumer of shared packages, you know the inherent benefits of robust software development relative to the quality of code, data, documentation, versions, and tests [41]. Similarly, creating a valuable package for personal use requires consideration for your future self and anticipation of distributing your code, should the need arise. Use version control and take advantage of existing resources. Indeed, there are R packages that aid in package development (e.g., devtools, usethis, testthat, roxygen2, rlang, drat) [28,29,56–59]. In the case of collaboration, the R project within RStudio IDE, is compatible with distributed development—a feature that couples well with version control. There is no lack of effective organizational frameworks to reference in the open source R community; in fact, repositories for many exemplary packages are available on GitHub. We recommend consulting resources authored by expert R developers including *R Packages* by Hadley Wickham and the official manual, *Writing R Extensions*, from CRAN [19,60].

Conclusion

R packages are a defining feature of the language insofar as many are robust, user-friendly, and richly extend R's core functionality. Some of the most prominent R packages are a result of the developer abstracting common elements of a data science problem into a workflow that can be shared and accompanied by thorough descriptions of the process and purpose. In this way, R packages have effectively transformed how we interact with data in the modern day in, perhaps, a more impactful manner than several revered contributions to theoretical statistics [4]. Packages greatly enhance the user experience and enable you to be more efficient and effective at learning from data, regardless of prior experience.

Nonetheless, the sheer quantity and potential complexity of available R packages can undermine their collective benefits. Finding and choosing packages, particularly for beginners, can be daunting and difficult. R users often struggle to sift through the tools at their disposal and wonder how to distinguish appropriate usage. These ten simple rules for navigating the shared code in the R community are intended to serve as a valuable page in your computing cookbook—one that will evolve into intuition and yet remain a reliable reference. May searching for and selecting proper tools no longer spoil your appetite and dissuade you from discovering, trying, creating, and sharing new recipes.

Table 1 (general packages)

```
library(kableExtra)
library(knitr)
```

```
# general packages data
gen_pkgs <- data.frame(
  Package = c("readr",
              "dplyr",
              "tidyr",

              "broom[note]",
              "purrr",
              "caret",

              "ggplot2",
              "kableExtra",
              "rmarkdown"),

  Description = c("Read rectangular data (e.g., csv, tsv, and fwf)",
                  "Grammar of tidy data manipulation",
                  "Tidy messy data",

                  "Tidy model output; convert statistical objects into tidy tib
                  "Functional programming tools for functions and vectors",
                  "Framework for predictive modeling",

                  "System for data visualization",
                  "Build complex tables and manipulate styles",
                  "Authoring framework for data science and reproducible resear
  Year = c("2015",
           "2014",
           "2014",

           "2014",
           "2015",
           "2007",

           "2007",
           "2017",
           "2014"),

  Author = c("Wickham et al.",
             "Wickham et al.",
             "Wickham & Henry",

             "Robinson & Hayes",
             "Henry & Wickham",
             "Kuhn",

             "Wickham et al.",
             "Zhu",
             "Allaire et al."),

  Documentation = c("https://readr.tidyverse.org/",
                    "https://dplyr.tidyverse.org/",
                    "https://tidyr.tidyverse.org/",

                    "https://broom.tidymodels.org/",
                    "https://purrr.tidyverse.org/",
                    "https://topepo.github.io/caret/index.html",

                    "https://ggplot2.tidyverse.org/",
                    "https://haozhu233.github.io/kableExtra/",
                    "https://rmarkdown.rstudio.com/lesson-1.html")
)
```

```
# general packages table
kable(gen_pkgs, format = "latex", booktabs = TRUE) %>%
  # scale
  kable_styling(latex_options = "scale_down") %>%
  # separate rows by category
  pack_rows("Data Manipulation", 1, 3) %>%
  pack_rows("Statistical Modeling", 4, 6) %>%
  pack_rows("Data Visualization", 7, 9) %>%
  # column wrap
  column_spec(1, width = "10em") %>%
  column_spec(2, width = "20em") %>%
  # bold column names
  row_spec(0, bold = T) %>%
  add_footnote("See the biobroom analog in Bioconductor",
               notation = "symbol")
```

Package	Description	Year	Author	Documentation
Data Manipulation				
readr	Read rectangular data (e.g., csv, tsv, and fwf)	2015	Wickham et al.	https://readr.tidyverse.org/
dplyr	Grammar of tidy data manipulation	2014	Wickham et al.	https://dplyr.tidyverse.org/
tidyr	Tidy messy data	2014	Wickham & Henry	https://tidyr.tidyverse.org/
Statistical Modeling				
broom[*]	Tidy model output; convert statistical objects into tidy tibbles	2014	Robinson & Hayes	https://broom.tidymodels.org/
purrr	Functional programming tools for functions and vectors	2015	Henry & Wickham	https://purrr.tidyverse.org/
caret	Framework for predictive modeling	2007	Kuhn	https://topepo.github.io/caret/index.html
Data Visualization				
ggplot2	System for data visualization	2007	Wickham et al.	https://ggplot2.tidyverse.org/
kableExtra	Build complex tables and manipulate styles	2017	Zhu	https://haozhu233.github.io/kableExtra/
rmarkdown	Authoring framework for data science and reproducible research	2014	Allaire et al.	https://rmarkdown.rstudio.com/lesson-1.html

[*] See the biobroom analog in Bioconductor

```
## trying to separate color; striped by group
#  row_spec(1:3 - 1, extra_latex_after = "\\rowcolor{gray!6}")
#  row_spec(0:3, extra_latex_after = "\\rowcolor{orange!6}") %>%
#  row_spec(4:6, extra_latex_after = "\\rowcolor{gray!6}") %>%
#  row_spec(7:11, extra_latex_after = "\\rowcolor{gray!6}")

## QUESTIONS
# Code font for package names in " "? \texttt{}?
# How do you repeat same symbol on multiple items with one footnote?
# How do you separate colors and stripe by group?
# Add title
# Add caption
# Cite packages in bib and add references in table?
# Embed url link to package documentation? Do we want to link cheatsheets?
# How do you add link/reference to Table 1 in text in the template?
# How do you hide code for table in knitted pdf...include=FALSE errors?
# Title for column 2: description/purpose/usage?
# Length of description/purpose/usage for each package?
```

Supporting information

Do we need to include any supporting information?

Acknowledgements

[Acknowledgment of people who have helped; suggestions from colleagues, etc.]

References

1. Perez-Riverol Y, Gatto L, Wang R, Sachsenberg T, Uszkoreit J, Veiga Leprevost F da, et al. Ten simple rules for taking advantage of git and github. PLoS computational biology. Public Library of Science; 2016;12.

2. Peng RD. Reproducible research in computational science. Science. American Association for the Advancement of Science; 2011;334: 1226–1227.

3. Goodman SN, Fanelli D, Ioannidis JP. What does research reproducibility mean? Science translational medicine. American Association for the Advancement of Science; 2016;8: 341ps12–341ps12.

4. Donoho D. 50 years of data science. Journal of Computational and Graphical Statistics. Taylor & Francis; 2017;26: 745–766.

5. Stodden V, Miguez S. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. Available at SSRN 2322276. 2013;

6. Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, et al. Bioconductor: Open software development for computational biology and bioinformatics. Genome biology. Springer; 2004;5: R80.

7. Holmes S, Huber W. Modern statistics for modern biology [Internet]. Cambridge University Press; 2018. Available: `https://web.stanford.edu/class/bios221/book/index.html`

8. Robinson D. The impressive growth of r [Internet]. Stack Overflow; 2017. Available: `https://stackoverflow.blog/2017/10/10/impressive-growth-r/`

9. Team RC. The r project for statistical computing [Internet]. The R Foundation; 2020. Available: `https://www.r-project.org/`

10. Hornik K. Are there too many r packages? Austrian Journal of Statistics. 2012;41: 59–66.

11. Wickham H, François R, Henry L, Müller K. Dplyr: A grammar of data manipulation [Internet]. 2020. Available: `https://CRAN.R-project.org/package=dplyr`

12. Wickham H, Henry L. Tidyr: Tidy messy data [Internet]. 2020. Available: `https://CRAN.R-project.org/package=tidyr`

13. Myles S. Phonenumber: Convert letters to numbers and back as on a telephone keypad [Internet]. 2015. Available: `https://CRAN.R-project.org/package=phonenumber`

14. Huber W, Carey VJ, Gentleman R, Anders S, Carlson M, Carvalho BS, et al. Orchestrating high-throughput genomic analysis with Bioconductor. Nature Methods. 2015;12: 115–121. Available: `http://www.nature.com/nmeth/journal/v12/n2/full/nmeth.3252.html`

15. Zeileis A. CRAN task views. R News. 2005;5: 39–40.

16. Smith D. The r community is one of r's best features [Internet]. Revolutions. Microsoft; 2017. Available: `https://blog.revolutionanalytics.com/2017/06/r-community.html`

17. Ellis SE. Hey! You there! You are welcome here [Internet]. rOpenSci. NumFOCUS; 2017. Available: `https://ropensci.org/blog/2017/06/23/community/`

18. Rickert J. What makes a great r package? [Internet]. RStudio; 2018. Available: `https://rstudio.com/resources/rstudioconf-2018/what-makes-a-great-r-package-joseph-rickert/`

19. Wickham H. R packages: Organize, test, document, and share your code. "O'Reilly Media, Inc."; 2015.

20. CRAN repository policy [Internet]. The R Foundation; 2020. Available: `https://cran.r-project.org/web/packages/policies.html#Submission`

21. Package submission [Internet]. Bioconductor; 2020. Available: `https://www.bioconductor.org/developers/package-submission/`

22. Transforming science through open data and software [Internet]. rOpenSci; 2020. Available: `https://ropensci.org/`

23. Theußl S, Zeileis A. Collaborative software development using r-forge. Special invited paper on" the future of r". The R Journal. The R Foundation for Statistical

Computing; 2009;1: 9–14.

24. Lang DT. The omegahat environment: New possibilities for statistical computing. Journal of Computational and Graphical Statistics. Taylor & Francis; 2000;9: 423–451.

25. McElreath R. Statistical rethinking: A bayesian course with examples in r and stan. CRC press; 2020.

26. Decan A, Mens T, Claes M, Grosjean P. When github meets cran: An analysis of inter-repository package dependency problems. 2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner). IEEE; 2016. pp. 493–504.

27. Decan A, Mens T, Claes M, Grosjean P. On the development and distribution of r packages: An empirical analysis of the r ecosystem. Proceedings of the 2015 european conference on software architecture workshops. 2015. pp. 1–6.

28. Wickham H, Hester J, Chang W. Devtools: Tools to make developing r packages easier [Internet]. 2020. Available: `https://CRAN.R-project.org/package=devtools`

29. Carl Boettiger DE with contributions by, Fultz N, Gibb S, Gillespie C, Górecki J, Jones M, et al. Drat: 'Drat' r archive template [Internet]. 2020. Available: `https://CRAN.R-project.org/package=drat`

30. Anderson GB, Eddelbuettel D. Hosting data packages via drat: A case study with hurricane exposure data. R Journal. 2017;9.

31. Xie Y. Bookdown: Authoring books and technical documents with r markdown [Internet]. 2020. Available: `https://CRAN.R-project.org/package=bookdown`

32. Xie Y. Bookdown: Authoring books and technical documents with r markdown. Chapman; Hall/CRC; 2016.

33. Ushey K. RStudio 1.3 preview: Integrated tutorials [Internet]. RStudio; 2020. Available: `https://blog.rstudio.com/2020/02/25/rstudio-1-3-integrated-tutorials/`

34. Chase W. Dataviz and the 20th anniversary of r, an interview with hadley wickham [Internet]. Medium; 2020. Available: `https://medium.com/nightingale/dataviz-and-the-20th-anniversary-of-r-an-interview-with-hadley-wickham-ea24507`

35. Team RC. Mailing lists [Internet]. The R Foundation; 2020. Available: `https://www.r-project.org/mail.html`

36. Muenchen RA. The popularity of data analysis software. URL http://r4statscom/popularity. 2012;

37. Wickham H. Advanced r. CRC press; 2014.

38. Leek J. How i decide when to trust an r package [Internet]. 2015. Available: `https://simplystatistics.org/2015/11/06/how-i-decide-when-to-trust-an-r-package/`

39. Vannoorenberghe L. RDocumentation: Scoring and ranking [Internet]. DataCamp; 2017. Available: `https://www.datacamp.com/community/blog/rdocumentation-ranking-scoring`

40. Smith A. Improving github for science [Internet]. GitHub, Inc. 2014. Available: `https://github.blog/2014-05-14-improving-github-for-science/`

41. Taschuk M, Wilson G. Ten simple rules for making research software more robust. PLoS computational biology. Public Library of Science; 2017;13.

42. Hester J. How does covr work anyway? [Internet]. The R Foundation; 2020. Available: `https://cran.r-project.org/web/packages/covr/vignettes/how_it_works.html`

43. Wickham H. Testthat: Get started with testing. The R Journal. 2011;3: 5–10. Available: `https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf`

44. Hester J. Covr: Test coverage for packages [Internet]. 2020. Available: `https://CRAN.R-project.org/package=covr`

45. Barts C. How to use github badges to stop feeling like a noob [Internet]. freeCodeCamp; 2018. Available: https://www.freecodecamp.org/news/how-to-use-badges-to-stop-feeling-like-a-noob-d4e6600d37d2/

46. Wilson GV. Where's the real bottleneck in scientific computing? American Scientist. 2006;94: 5.

47. Bryan J. Excuse me, do you have a moment to talk about version control? The American Statistician. Taylor & Francis; 2018;72: 20–27.

48. Perkel J. Democratic databases: Science on github. Nature News. 2016;538: 127.

49. Wickham H, Averick M, Bryan J, Chang W, McGowan LD, François R, et al. Welcome to the tidyverse. Journal of Open Source Software. 2019;4: 1686. doi:10.21105/joss.01686

50. Robinson D, Hayes A. Broom: Convert statistical analysis objects into tidy tibbles [Internet]. 2020. Available: https://CRAN.R-project.org/package=broom

51. Andrew J. Bass SL David G. Robinson. Biobroom: Turn bioconductor objects into tidy data frames [Internet]. 2020. Available: https://github.com/StoreyLab/biobroom

52. Kuhn M. Caret: Classification and regression training [Internet]. 2020. Available: https://CRAN.R-project.org/package=caret

53. Harrell Jr FE, Charles Dupont, others. Hmisc: Harrell miscellaneous [Internet]. 2020. Available: https://CRAN.R-project.org/package=Hmisc

54. Broman KW. Broman: Karl broman's r code [Internet]. 2020. Available: https://CRAN.R-project.org/package=broman

55. Parker H. Personal r packages [Internet]. 2013. Available: https://hilaryparker.com/2013/04/03/personal-r-packages/

56. Wickham H, Bryan J. Usethis: Automate package and project setup [Internet]. 2019. Available: https://CRAN.R-project.org/package=usethis

57. Wickham H. Testthat: Get started with testing. The R Journal. 2011;3: 5–10. Available: https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf

58. Wickham H, Danenberg P, Csárdi G, Eugster M. Roxygen2: In-line documentation for r [Internet]. 2020. Available: https://CRAN.R-project.org/package=roxygen2

59. Henry L, Wickham H. Rlang: Functions for base types and core r and 'tidyverse' features [Internet]. 2020. Available: https://CRAN.R-project.org/package=rlang

60. Team RC. Writing r extensions [Internet]. The R Foundation; 2020. Available: https://cran.r-project.org/doc/manuals/R-exts.html