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]. The Methods section of scholarly work is not sufficient to convey the complex data analysis methods and powerful tools of our time—data and code must be shared. Some variability in scientific outcomes can be attributed to differences in workflow and, today, the scientific community deems the absence of automation to be irresponsible [4]. Many traditional approaches that produce such inconsistencies also tend to be at odds with productivity and collaboration. 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]. These researchers rely on accessible and robust tools to ensure their computations are reproducible.

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 [6]. 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 [7,8].

R was developed by statisticians and is collaboratively maintained by an international group of core contributors [9]. Unlike several popular proprietary languages such as MATLAB or SAS, R is freely available, open source, and easily extensible software; the user can access, alter, extend, and share code in various applications. Accordingly, a vibrant community of R users has emerged, many of whom develop R packages to complement and extend the functionality of base R. An R package is the fundamental unit of shareable code. 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 numerous analogies in computing 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. The few tools that come with your kitchen (e.g., `stats`) are indeed R packages, whereas "R extension" describes the tools you add to your kitchen. It's 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 to 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, determine whether you need one. Consider your purpose by first identifying your goal then defining the scope and tasks to achieve it. For some tasks and workflows, coding your own recipe with existing tools is practical, while others 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, although base R is relatively stable and slow to change, many R packages evolve rapidly; unbeknown changes made to packages can cause defects in portions of your code affected by these changes.

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 your 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. Surprisingly, you can use an R package to access the Twitter API, send emails from R, and there is even a package for converting English letters to numbers as on a telephone keypad [13–15].

If you decide you need a package, next ask yourself: Which functionalities in base R are restrictive in 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 will be primed to recognize what you do and don't need. List domain-specific

keywords that describe what you are trying to do and how you could do it, so you narrow your search. Identify the type of inputs you have and envision working with them; contemplate the desired outputs and corresponding format. 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 [16]. When you look for a package to visualize the data, you will want to choose one that addresses such restrictions. If, however, you don't quite know (or have a difficult time articulating) what you need, don't hesitate to search the internet for examples that capture your purpose. Sometimes, for instance, it's helpful to search Google Images for plots and figures to identify the terminology that describes what you are trying to do, how people approach similar tasks, and potential challenges that may arise.

Rule 2: Find and collect options

Tips and leads for finding R packages exist in a variety of places online, in print, and elsewhere. If you've searched for packages a lot, you likely have a shortlist of tried and true starting points—a perk of being a long-time R user. On the other hand, new R users who haven't developed an internal compass to navigate the initial ins and outs of finding a package, don't know where to start looking for a package to suit their task. Here are a few directions to head in: You can discover new packages any time you learn R-related topics, browse the internet, or connect with 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"`. Keep in mind, however, this approach sometimes leads to frustration and confusion when attempting to find a package tailored to your purpose (see Rule 1). For a more directed approach, you can search the package lists on repositories including the Comprehensive R Archive Network (CRAN), Bioconductor, and rOpenSci or browse packages available on code-sharing websites, with language filters for finding repositories that heavily rely on R code, like GitHub and GitLab—all of which will be further discussed in Rule 3.

That said, in many cases, you may find it more helpful 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()`, which is compact notation of the form `package::function()`. 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 [17].

The computational biology and bioinformatics analog of CRAN Task Views, biocViews, is a list of packages, available through the Bioconductor website, that provide tools for analyzing biological data. Your keywords from Rule 1 will come in handy here; for instance, you can search "RNA-seq" to find all packages related to high-throughput sequencing methods for analyzing RNA populations. You can also browse the biocViews tree menu to narrow your search based on which general and specific categories describe your task. The list includes each package's name linked to information and documentation, its maintainer, a short description, and its rank relative to other Bioconductor packages.

Other curated collections and topically-linked lists of packages are also available. CRANberries, for example, is a hub of information about new, updated, and removed packages from CRAN. Another place to find well-maintained tools is rOpenSci packages. These filterable and searchable peer-reviewed (see Rule 6) R packages are organized by name, maintainer, description, and status markers based on activity, association, and review.

Embrace the community

An inclusive and collaborative community is an overlooked, yet integral, aspect of a software's success [18]. 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 [19]. 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, and Visualization.

You can also attend—or access content from—conferences to learn about improved, recently released, or up-and-coming R package developments and applications. Two large annual R conferences are rstudio::conf for industry and useR! (not exclusively) for academia. As supportive and innovative subsets of the R community, some boutique industries that use R host smaller conferences: R/Pharma (pharmaceutical development), R/Finance (applied finance), and BioC (open source software for bioinformatics). 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. Talks and presentations at conferences are often recorded and made available online for playback if you can't attend in person or wish to revisit content from past conferences. For instance, R Studio has a webpage devoted to videos of past conferences by year and the talks from useR! are posted on the R Consortium YouTube channel. Of late, the R community and rOpenSci have popularized nontraditional Unconferences and R Collaboratives at which people interact and share ideas in a collegial and unstructured atmosphere; as you may imagine, R packages are a hot topic of conversation at these events.

Don't forget, 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) [20]. 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 are shipped to 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 (not just R 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. Volunteers from the R Core Team (est. 1997) host and maintain a monumental collection of R packages on CRAN (over 16,000 as of August 2020) on almost any conceivable topic, from random forests to multidimensional maps [9,21,22]. Due to its longevity and historical role, Rickert asserts that "CRAN is the greatest open source repository for statistical computing knowledge in the world" [20]. A key advantage of using a package from CRAN is that the repository integrates easily with your base R installation [23]. You can simply use the `install.packages()` function to install a package from CRAN, along with all of the packages it depends on; source and/or binary code are automatically saved to your computer in a designated package library [24]. These features silently enable seamless ease of use. 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 some strict regulatory practices for the selection and maintenance of the packages they host, particularly as standards pertain to the structure of a package. CRAN is more or less mechanical in their approach to accepting and keeping a package as it evolves in the R ecosystem. CRAN is not as concerned with content or substance because they do not have a peer review process (see Rule 3 and Rule 6). At worst, a CRAN package might have a solid framework with help files for each function, but its underlying code might not necessarily do what it claims. A package must pass a series of initial stability tests in accordance with the CRAN Repository Policy before obtaining publication privileges [25]. A number of CRAN packages make impactful contributions to certain subject areas including statistical computing and graphics; however, some CRAN packages, such as `weathermetrics` for converting between weather metrics, are more convenient than substantial [26]. Volunteer CRAN maintainers actively monitor portions of code (e.g., vignettes, help files, test directories) for source and contributed packages to ensure compatibility with the latest version of R and dependencies; they modify or remove packages that do not uphold these publication quality standards.

If you're looking for tools specifically designed for high-throughput genomic data, the Bioconductor repository is a more specialized repository than CRAN that's 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 [7]. 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 [27]. Bioconductor boasts a modularized design, wherein data structures, functions, and the packages that contain them have distinct roles that are accompanied by thorough documentation. Accessibility is a pillar of the Bioconductor project; all forms of documentation, including courses, vignettes, and interactive documents, are curated for individuals with expertise in adjacent disciplines and minimal experience with R (see Rule 4) [7]. Bioconductor packages integrate well with base R by different means than CRAN packages; you should use `BiocManager::install()` to install Bioconductor packages with their corresponding dependencies. Bioconductor has a structured release schedule and thus a unique package management system which automatically loads your Bioconductor packages in versions that are all from the same release cycle and compatible with your current version of R [27]. The R environment and its package system is fundamental to the implementation of Bioconductor's interoperable and object-oriented (S4) infrastructure. The tasks for which Bioconductor packages provide tools generally involve large and complex datasets that need to leverage R's object oriented programming functionalities—a set of specific object classes in particular (e.g., ExpressionSet class) [28]. This helps, among other things, keep metadata properly aligned with sample data. 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, and comply with additional package guidelines [29]. Unlike CRAN, however, to ensure Bioconductor software maintains a centralized content focus, their packages are coordinated and peer-reviewed.

Additionally, there are smaller, specialized 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 [30]. The repository only includes packages that have either been developed by staff to fill a gap in infrastructure or have passed their open review process. These packages are within the scope and aims defined by rOpenSci, including those that focus on data extraction, workflow automation, and field or laboratory reproducibility tools [31]. R package maintainers can even create and host their own personalized repository using the `drat` (Drat R Archive Template) package, which enables developers to design individual repositories and suites of coordinated repositories for packages that are stored in and/or distributed through platforms like GitHub [32]. 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 size limit for packages [33]). 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 [34,35].

While there are evident benefits of exclusive repositories, there are also considerable advantages to hosting a package on online hosting platforms for git repositories like GitHub or GitLab [20,36]. These platforms can be used for production of R packages, including large-scale packages. Package developers can post the directory with all source code for their package as they develop it, from the earliest stages. The platforms allow for continuous integration and continuous deployment for testing data and code to ensure a stable end-product. One particularly popular platform when it comes to sharing R packages is GitHub. An increasing number of packages are hosted on GitHub during the development stages, and so it will often be the first place you can find up-and-coming packages (see Rule 2). Even if developers choose not to distribute their

package through GitHub, the stable release versions of such packages are often published on CRAN or Bioconductor [37], and for these GitHub can be a place to check out planned changes for the next stable version that the developer plans to publish. GitHub provides R users with open access to package code, a timeline of help resources (see Rule 4), and a direct line of communication to developers. It is simple to install the latest version of a package you want to use from GitHub via `devtools::install_github("DeveloperName/PackageName")`; however, the decentralized nature of GitHub is not conducive to a tool that automatically locates and installs corresponding dependencies that are not on CRAN [38]. In fact, the rapid uptick in package development and ensuing inter-repository dependencies has sparked an ongoing debate on whether regulated repositories such as CRAN and Bioconductor are preferable to distribution platforms like GitHub due to this complication [20,36,39]. Further, when a package is installed from GitHub, typically only the source code is available, not binaries pre-built for your computers operating systems. This means that the installation process will need to build the package from source on your computer, which may require certain compilers and other tools (e.g., XCode for macOS, Rtools for Windows). Both R users and package developers benefit from interactive feedback channels through GitHub Issues and the Star rating system.

When you are searching for and selecting packages, it is important to note that a single package is often available in multiple places. Packages that are on CRAN, Bioconductor, or rOpenSci are usually on GitHub. There is an unofficial read-only GitHub mirror at CRAN, METACRAN, in which *all* CRAN packages have their own GitHub repositories with all versions, original dates, and authors. Additionally, package developers often use their personal GitHub accounts to publicize a package while it is being developed and continue to host development versions after it has been published on formal repositories such as CRAN. In this case, CRAN would host a "stable" version of the package, which may simultaneously be in the "master" branch of the developer's GitHub repository for the package; forthcoming package updates would remain in a "development" branch in the same GitHub repository. Although less relevant to new R users, this means that you can access and test upcoming versions of a package through GitHub, which may be particularly helpful if you're developing a package with dependencies and want to know if upcoming changes to a package that your code depends on will cause issues. Lastly, there are some instances where a packages is posted on CRAN—not only because CRAN is its main repository, but also to ease its connection with base R—then also published in a specialized repository like rOpenSci. As part of their review process, rOpenSci asks authors of submitted packages if they plan to publish on CRAN (or Bioconductor) and many times packages become available in both places. If you want a package that more or less guarantees that it does what it says it will do, consider packages that have been peer reviewed. Knowing this, you might start by searching ROpenSci's repository website rather than CRAN Task Views or biocViews.

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.

Help associated with R packages generally falls into one of two categories: help documentation that comes embedded in the source code of the package (i.e., ships with the package) and documentation or other resources that are available more broadly, beyond the package's source code. The documentation that ships with the package can consist of help files, vignettes, `README` files, tutorials, and `pkgdown` documentation. Broader help resources can include webpages or online books whose content is hosted outside the package source, online tutorials, electronic mailing lists, and large and vocal communities of users. Implicitly, if an author has taken the time to provide copious and useful help documentation—within package code, outside of it, or both—this indicates that the authors are at least serious about their package development and adds confidence that they know what they (and their package) are doing. Exemplary documentation can signify an exceptional package, while you'll often want to avoid using a package if the quality and quantity of related resources is lacking.

At the most granular level, every R package should come with a help file for each function that the package maintainer intends for you to use directly (as opposed to smaller help functions meant to solely be used within other package functions). This practice is required for packages on some repositories, like CRAN, but might not be adopted for packages posted on less restrictive platforms, like GitHub. For each function in the package, you can access its help file from the console by typing `?` followed by the function's name (e.g., to get the help file for the `mean` function, type `?mean`). Function help files provide information about how to use the function, including the parameters that can be set, the format to use when specifying arguments for these parameters, a description of the file, and even references to papers or other resources on which the function's underlying algorithm was based. They often contain executable examples to help you get a feel for how to use the function in practice. If you would like to check all help documentation for a package, you can access an index of help pages from the console via `help(package = "...")`. 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. The extent and quality of content in these help files will vary; ideally, packages should have thorough documentation at the function level. If the 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, don't use the blender).

Fortunately, plenty of R packages include additional documentation beyond this minimum. While the function-level help files can be rather technical and extraneous to new users, 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. See, for example, the extensive vignette for the `sf` package[40]. 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 from base R. For a package shared through CRAN or Bioconductor, a list of the package's vignettes can also be found on the CRAN or Bioconductor homepage for the package. Particularly for packages hosted on GitHub, developers will create a mini-vignette in the `README` to provide a high-level overview of their package and its

components, along with a basic introduction and short tutorial for its usage. This mini-vignette, displayed on the package repository's main page, should simply tell you why you should use it, how to use it, and how to install it [24].

Tutorials, most commonly those powered by the `learnr` package, are another effective and popular way to facilitate learning about R packages. They are either conceptual and exploratory or specific and structured in their content and can include a range of features: narratives, figures, videos, illustrations, equations, code exercises, quizzes, and interactive Shiny elements. You can run `learnr::run_tutorial(name = "...", package = "...", shiny_args = "...")` to deploy the `learnr` tutorial included within a package, if one exists [41]. Alternatively, as of RStudio v1.3, you can access `learnr` tutorials for packages you've installed, directly from the Tutorial pane in the IDE [42]. Some developers publish their `learnr` tutorials as Shiny apps, which can be called from within the `learnr::run_tutorial` function and accessed outside of the RStudio interface.

Several thousand developers have been using `pkgdown` to create static HTML documentation [43]. This tool allows developers to collect dispersed forms of documentation for their package such as help files and vignettes, then build a single package website with this information plus more. The `pkgdown` website itself as well as those for `naniar`, `valr`, and `TransPhylo` are examples of `pkgdown` documentation in action for R packages [44–47]. Sometimes, you can find a link to a package's corresponding `pkgdown` website in the "URL" section of its CRAN page, which is often listed alongside its GitHub link.

So far, we've only mentioned types of help documentation that are incorporated in the source code of a package; you can investigate them by looking through a package's underlying code. However, there are easier avenues for accessing help documentation encoded in a package. RDocumentation, for example, is a searchable website, package (`install.packages("RDocumentation")`), and JSON API for obtaining integrated documentation for packages that are on CRAN, Bioconductor, and GitHub [48]. RDocumentation may include: an overview, installation instructions, examples of usage, functions, guides, and vignettes. This is a subsidiary reason why is may be useful to look for packages shared on these platforms (see Rule 3).

Some packages have a comprehensive set of resources beyond those encoded in the package source code. For instance, there are online and/or print books associated with certain packages. 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 [49]. The `bookdown` package itself has an online book that details usage of the package [50].

Other help documentation that is shared outside of the package's code includes online courses, galleries, and cheatsheets. Some packages have corresponding online courses, which might be taught by the maintainer. *GAMs in R*, for instance, is a free interactive course on Generalized Additive Models in R using the `mgcv` package [**???**]. Every now and then, an online course will cover one or a few select packages that revolve around a central theme such as a types of modeling. A suitable example, *Supervised Machine Learning Case Studies in R*, features several `tidy` tools with a primary focus on `tidymodels` [**???**]. Galleries, as the name would imply, are sites that display works, articles, and/or code examples associated with certain packages. As you might expect, galleries can be valuable sources of inspiration, particularly for packages that provide tools for data visualization tasks. See, for example, the galleries for `ggplot2` extensions, `dygraphs` for R, `Shiny`, and `htmlwidgets` for organized displays of outputs from these packages [**???**]. The `Rccp` gallery is an instance of a gallery that

contains collections of articles and code examples for the `Rcpp package` [???]. RStudio hosts cheatsheets of concise usage information for popular packages through code and graphics organized by purpose, including cheatsheets made by RStudio's team and those contributed by others. 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. Packages with their own cheatsheets include `caret` (statisical modeling and machine learning), `data.table` (data manipulation), `devtools` (package development),`dplyr` (data transformation), `ggplot2` (data visualization), `leaflet` (interactive maps), `randomizr` (random assignment and sampling), `sf` (tools for spatial data), `stringr` (character string manipulation), and `survminer` (survival plots) [???]. Further information about some R packages is available in video tutorials, webinars, and code demonstrations (i.e., "demos").

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. The 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) [51]. In the early days, the R-help mailing list was the only way to seek direct assistance; since, the R community has evolved to make asking for help more widespread and accessible, with inclusion and creative problem-solving as hallmarks of its online presence [52]. Certain packages have independent mailing lists; `statnet` is an example of a suite of packages that has its own community mailing list. 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 [6,53]. 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 [54].

One example of a package—mentioned a few times already—with notable documentation and first-rate help resources is `Rccp`. The developers maintain both a main and additional website with a wealth of organized information about the package and resources, including a gallery of examples, associations, publications, articles, blogs, code, books, talks, and links to other resources with `Rccp` tags. The package also has a dedicated mailing list, for developers using the package and the package is thoroughly described in a print book by its maintainer [55] and in chapters or sections of other books (e.g., [54]).

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 popular a 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. There are metrics that you can use to determine how often an R package has been downloaded, how long it's been around, and how often (and recently) it's updated through new versions.

You can find several of these metrics by visiting the site(s) on which a package is shared (see Rule 3). If a package is shared through a GitHub repository, the homepage for the repository will list the number of GitHub Stars, Forks, and Watchers for that

particular repository. As proxies for a repository's overall following, and possibly indirect measures of quality and impact, Stars indicate that people like or are interested in a project, Forks reveal the number times a repository has been copied, and Watchers represent GitHub users who are subscribed to updates on a repository's activity [56]. Although not a definitively unbiased metric, a large number of Stars, Forks, and Watchers associated with a package implies a substantial following and widespread usage [57]. If a package is posted on CRAN or Bioconductor, its page on that site will include the date on which its latest version was released as well as a link to "Old sources"—previous versions of the package and the release date of each. The page also provides a link to a `NEWS` file, where you can see a list of all published versions as well as the major changes made from one version to the next. To further determine if a package is well-established in the R community, refer to the number of versions and updates (more is better) in addition to the date of the most recent versions and updates (newer is better). You can also see, on a package's CRAN or GitHub page, the number of reverse dependencies it has. When a package has numerous reverse dependencies, the code underlying many other R packages incorporates functions from that package, which can indicate that other R developers find that package useful.

RDocumentation (see Rule 4) is rich with stats on R packages, uniting many of the previously discussed metrics, while also providing cross-package summaries. 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 used within other packages, and hence frequently downloaded as automatic downloads with those packages, do not skew the calculation [58]. You can research stats on corresponding dependencies for a more holistic picture.

Most of the above-mentioned resources will help you learn more about packages on an individual basis. Wouldn't it be nice if you could compare related packages that may have features that overlap? Wouldn't it be even nicer if you could use an R package to easily do so? You're in luck; rOpenSci developed `packagemetrics`: A Package for Helping You Choose Which Package to Use [59]. Fun fact: the `packagemetrics` project was born at an Unconference [60]. You can install `packagemetrics` via `devtools::install_github("ropenscilabs/packagemetrics")` and run the `package_list_metrics()` function where the input is a vector of packages of interest, by name. Then, you can use `metrics_table()` to obtain a table of information that includes each package's respective number of downloads, GitHub Star rating, `tidyverse` compatibility, and several more indicators to help you decide which package is most established and widely used (**Figure of table output [BA: That could be great, or even use a Box to show an example of running the code you just described and the resulting output.]**).

An important caveat to consider: several of the metrics introduced in this Rule are based on how often a package is downloaded; however, it is not uncommon for users to download a package to explore its functionality and ultimately decide against using it for their task. Therefore, these suggested metrics can provide a first pass at identifying popular packages—and will, in particular, help you distinguish popular packages for common general tasks, like data wrangling and visualization—but may be less helpful when you are selecting a package for a very specialized task. In this case, evidence of peer acceptance and review will usually be more instructive, which brings us to our next Rule.

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. While scientific articles undergo a peer review process that is fairly standardized across disciplines, R packages have a broader range of possibilities for peer review or other evidence of acceptance by members of the scientific community. 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.

First, some maintainers of select packages have written an article describing their package and its algorithms; such articles must go through the traditional peer-review process in order to be published. Certain journals publish articles about R packages themselves while others feature work that used a particular package. Journals with a targeted and technical focus on R packages and other open-source scientific software include the *Journal of Open Source Software*, the *Journal of Statistical Software*, and *The R Journal*. Discipline-specific journals like *BMC Bioinformatics*, may also publish articles to introduce R packages that have proven to be useful for aspects of the bioinformatics research process, for example. Try searching various websites for prominent journals in your field with queries that include `"R package"` to find these types of articles. Note, the number of Google Scholar citations for a package or, similarly, an article about a package, serves as a metric for a package's impact on scientific research and utility in research contexts (see Rule 5).

For an R package featured in a peer-reviewed journal article, you can be assured that the ideas and principles behind the package have been evaluated by outside reviewers. However, such a review may not have included a fine-grained peer review of the underlying code in the package itself. There is an expanding movement toward direct peer review of R packages at the code level. More specifically, rOpenSci is a unique example of an ecosystem of open source tools with peer reviewed R packages (see Rule 3) [30]. The rOpenSci organization oversees peer review of R packages at a highly detailed level that is increasingly desired by scientists; packages published through their repository have passed external review of the code. They limit their scope to packages that help scientists implement reproducible research practices and manage the data lifecycle [31]. Their review process is conducted openly on GitHub and, as a result, the scientific community can view the discussions between package maintainers and peer reviewers that form this transparent process.

You can also seek evidence of peer acceptance of a package by finding out how often researchers have used it within scientific papers describing research, rather than package-focused papers. Useful packages may be mentioned in the Methods and/or References sections of such papers. Packages hosted on CRAN or Bioconductor have a standardized format for citation, and a call to `citation("PackageName")` will print the suggested BibTex citation for the package. Further, 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 [61]. 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 base on citations using these formats. Further, if a package has such a DOI, you can explore the network of research associated with that package. While it is best practice to cite R packages using these citation formats, occasionally R packages will be mentioned in the Methods section of a paper without a formal citation.

Finally, many R packages are associated with content in books and series from scientific publishers. For example, Springer has a *Use R!* series with books covering specialized topics from finance to marketing, from chemometrics to brain imaging, and more. Additionally, consider reading titles from *The R Series* at Chapman & Hall/CRC.

The books in this series span topics related to R in terms of its applications (e.g., epidemiology, engineering, social sciences), uses (e.g., statistical methods), and development (e.g., creating packages).

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 in 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 [57].

Information about the package authors can be found in the metadata of the package. If you are looking at the source code of the package, there should always be 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. In the "Author" field of this file, you can identify both the "Maintainer" of the package (a similar role to that of the corresponding author for an article) as well as other authors or contributors. For packages hosted solely on a platform like GitHub, this might be the only way to discover a package's author. On the other hand, for packages hosted on repositories, this package data is usually also posted by the repository somewhere online. Packages on CRAN have a page of metadata that is accessible through a standard web address. For example, to see the metadata page for the `ggplot2` package, you can go to `https://CRAN.R-project.org/package=ggplot2`; replace the package name at the end of the web address with the name of any other CRAN package to access analogous package-specific metadata. Here, you will notice various abbreviations associated with the names in the "Author" and Maintainer" fields that denote roles such as author `[aut]`, creator/maintainer `[cre]`, contributor `[ctb]`, and funder `[fnd]`. Similarly, Bioconductor packages each have their own webpage, with some metadata at the top of the page, including the maintainer and other authors. For example, the page for the package `DESeq2` can be found at `https://bioconductor.org/packages/release/bioc/html/DESeq2.html`; you can view pages corresponding to other Bioconductor packages by altering the web address as previously described.

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. If the package was developed by multiple authors, research each of them to evaluate the robustness of the team. 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. Frequent commits and effective resolutions of GitHub Issues can be a testament to the authors' priorities and commitment. If an author has a history of package development, peruse their portfolio of packages to see if any are highly regarded, recognizable, or impressive at first glance. When you use a great tool in your kitchen, you consider buying more tools from that

brand. If, in the past, you found a package that you like to use and is helpful for your work, it can be worthwhile to research other packages made by that author and/or maintainer. You might benefit from making a habit of paying attention to who makes tools that work well for your needs and are designed in a way that seems intuitive to you—take note of these "favorite" package authors.

As you may know, the R community values collaboration—it is not uncommon for packages to be created and maintained by more than one person and, in many cases, large teams. Roles within these teams can shift as the package evolves. As an example, `ggplot2` has no small team of authors and its original maintainer is not the same as its current one. Oftentimes, existence of a large team can give you clues (although not guaranteed) about how the package is evolving in the R ecosystem. For one, it could indicate that a package is very useful and has generated rich community involvement, as is the case for `ggplot2`—many of its authors were added after the first release. This might reveal that a package is part of a larger-scale and more organized project. Also, a sizeable team could imply that the package grew out of a substantial collaboration, like a funded grant with a multi-person team; this happens a lot for specific methods packages. The `flowCore` package is one example of such a case [62,63]. This might suggest that a package has a certain level of review and investment from several R programmers (the authors), not just one, as in a solo-authored project. It's also useful to examine the full list of authors (even if it's long) because you might not recognize the maintainer's name, but you might recognize another author's name. This can happen if the maintainer is a student or someone else that's up-and-coming; sometimes, more senior authors do not assume the role of maintainer, even though they are quite invested in the project. It's worth looking to see, for example, if someone who is well-known for excellence in methods development is one of the non-maintaining authors, which may help you gauge the quality of the fundamental algorithms and techniques in the package code. By extension, the indicators of developer involvement and reputation mentioned in this Rule 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 don't 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 [64]. There are concrete ways to measure a tool's robustness beyond whether it works for those who did not create it. As a user, there are four main relevant package development protocols—in addition to some related indicators—that you can investigate to assess the underlying stability and utility of a package: dependencies, unit testing, version control, and continuous integration.

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) [1,24]. You can explore the version history of a package to see if it is up-to-date (See Rule 5). Package dependencies are not only important to establish stability, but might also give you a hint as to whether a package will fit well into your workflow. This is an especially critical consideration if you're using a "tidy" workflow or working with special object types in, for example, a Bioconductor workflow. You can review the list of dependencies for a package in question to check if it depends on some or many of the packages in your workflow. If this is the case, you can be fairly confident that the package uses the same (or at least

similar) style and conventions as those you want to use.

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 [24,65]. This helps assure the developer (and you) that the package's underlying units of code do what they intend to do. 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 [24]. 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 [66]. 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 [67]. You can look at the source code for a package to confirm it has a testing suite; for instance, if you are on a package's GitHub page, you can navigate to the main Code page and look for a folder labeled `tests` or something similar.

Relatedly, developers who host their packages on GitHub, post status badges, most often, in the overview (`README`) section of the repository webpage for the master branch. GitHub badges are a common self-imposed method to signal use of best practices and motivate developers to produce software that is high in quality and transparency [68]. They are easily visible, interactive, and increase readability of the `README`. You may see, for example, license (`license`), version (`release`), dependency (`dependencies`), code coverage (`codecov`), or build status (`build`) badges, all of which are good indicators of package caliber. What's more, badges have dynamic status tags to indicate the current status of the particular part of a workflow to which the badge refers; as an example, a `build` badge with a `passing` status tag could be telling you that the package is passing continuous integration tests, which you will learn more about soon [56]. Badges are also clickable buttons that link to a page that describes and displays detailed aspects of the metric associated with the badge. For instance, when you click on a `codecov` badge, you'll likely end up on a page with a timeline, graphics, and information about the percentage and distribution of test coverage for the package. You can refer to special badges like `CRAN`, `Bioconductor`, or `rOpenSci` to determine which repository (or repositories) a package is shared on (See Rule 3).

As we mentioned in Rule 3, version control has an essential role in package development and computational literacy more broadly [24,69]. 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" [[69]; p6]. Git is a decentralized open source version control system that is useful regardless of whether a project is independent or collaborative [70]. 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 [1,71]. As you may expect, Git coupled with GitHub is the version control duo of choice among serious R package developers [24]. 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.

Lastly, developers often implement continuous integration (CI) in conjunction with

using a version control hosting platform like GitHub. CI refers to a process by which developers frequently update and commit their contributions to a shared repository and routinely merge all authors' working copies/branches of a project into the master (main) branch to monitor compatibility—without permanently merging everything. Just as you are happy to know that a manufacturer guarantees their kitchen tool is a unified realization of many working parts, you should be impressed when a package is set up to use CI. This is particularly important for packages with a large development team. Use of CI is a good sign that the components of the package are regularly checked to ensure that code passes tests and the package builds without failures [72].

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.

At this point, what you have learned about the package should be quite helpful. If the package has a vignette, this is a great place to start experimenting with code. Vignettes include many common data science problems with solutions; you can run the code examples, tweak them, and compare the outputs. First, try some examples with the example data provided by the package, then see if you can translate that to working with your own data. Sometimes, package functions will require inputs to be in a specific data type (e.g., a dataframe, or a certain type of S4 object) or to have a specific format (e.g., if it's in a dataframe, to have certain columns with certain names). By moving from the example data to your own, you can gauge the effort it will take to transform your data to fit these formats. You can also be on the lookout for whether certain characteristics of your data (e.g., fields with missing values) cause complications when using the package's functions.

You can continue on to looking through the collection of help files for the package. Help files often include executable examples, and you can again start by testing these with the example data provided by the package, then move on to trying them with your own data and within your own workflow. 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 [7]. On the other hand, the `tidyverse`, a unified suite of packages with an grammatical structure employed within a "pipeline", expects dataframe objects [73]. Because of this, when you are working in the `tidyverse`, you may have to take extra steps to extract components

from S3/S4 objects to re-frame the data into a dataframe. Some packages already exist to help with this interfacing task between S3/S4 objects and `tidyverse` tools. The `broom` package, and the bioinformatics analog, `biobroom`, aim to alleviate these disruptions by converting "untidy" objects in S3/S4 objects into "tidy" data in dataframes, thereby making it easier to integrate statistical functions into the structure of the `tidyverse` workflow [74,75]. 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 [76].

Rule 10: Develop your own package

What if you can't find a package to solve your data science problem? Though you may not be the best cook, we all come upon times—of creativity or necessity—that prompt us to experiment in our kitchens. Rather than exclusively being a user of packages, you can create them—more easily than you may think [24].

If you decide to write your own R code to perform your task (see Rule 1), a natural next step is to wrap this code into a function that can be re-used; an R package is basically just a collection of one or more of these functions. The essence of an R package is a self-contained set of algorithms—encoded as functions—that can be used in combination with other self-contained sets of algorithms of different shapes and sizes, including the set of algorithms shared through base R. The uniquely structured functions within a package help us implement that knowledge and weave it into novel scientific work. The reasons why you might want to create a package are abundant: necessity, innovation, standardization, automation, specialty, containment, organization, sharing, collaboration, extensibility, and more.

Whatever your motivation, packages are simply sets of tools; you can create a package out of any collection of specialty functions. A key tenant of efficient programming is the "DRY" principle—Do not Repeat Yourself. Functions help you replace copied and pasted code with a defined, compact piece of code in a single place in a script, which you ca call throughout in shorthand. They help with efficiency and are warranted when you repetitiously copy and paste your code while making slight modifications after each iteration [54]. Since R packages are just sets of these functions, they are not solely reserved for large-scale, comprehensive methods; rather, package development can also help you learn how to apply proper coding techniques to writing functions and documentation with reproducibility and collaboration in mind [77]. 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 [78,79]. They're like the collection of little tried-and-true tricks and tools you've made to cook your own recipes in your kitchen.

While personal R packages are virtually unbounded, we'll describe some of their immediate uses. If you often do the same specific, discipline-oriented data analysis tasks—for which either a package doesn't exist or you don't want to rely one that does (See Rule 1)—a personal R package could help you automate some redundant aspects of your work. You can even connect components of other tools you're using, that wouldn't otherwise play nicely together. For example, if you're work largely involves packages that output S3/S4 object types, but you want to use standard `ggplot2` tools to visualize the output, you will encounter a problem because functions in `ggplot2` expect a dataframe as the standard input, not these more complex object types. In this case, you might want to write some of your own functions that extract what you need from the S3/S4 objects and convert them into a dataframe input type so they are compatible with `ggplot2`. Interestingly, this type of occurrence is the precise motivation behind the

widely used `broom` package [80]. You can also create visualization tools that are tailored to your research and include them in your personal R package. If you frequently design a certain type of plot with your data, you can encode that process into a function, even if it relies heavily on functions from other packages. For instance, this could happen if you implement particular tweaks whenever you make scatterplots with your data; many R users build custom themes within `ggplot2` for purposes like personal or professional branding, which can be wrapped into functions and contained in a personal R package [72].

Although you may not anticipate that anyone else will use your tools, following best practices for package development is still a good idea. You wouldn't want to craft a useful cooking hack without having a good way to replicate it when you need to cook the same (or similar) recipe again. Even if you're the only person who will use your package, be kind to your future self—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 [64]. There's also no reason not to take advantage of existing resources when coding your package; all your newfound special tools paired with others' shared tools can help you make exquisite recipes. There are R packages that aid in package development (e.g., `devtools`, `usethis`, `testthat`, `roxygen2`, `rlang`, `drat`) [32,38,81–84]—while these packages can provide industrial-strength support for developing large and complex packages, they also prove useful even in developing small, private packages. They make it easier and more efficient, for example, to write help documentation and unit tests for functions in your package. RStudio's IDE has a framework called "Projects", which can help you establish an organized framework of directories within an R Project (`.Rproj`)—specifically designed for creating R packages—that is straightforward to navigate throughout the package development process. Tips on using many of these package development tools are available through resources authored by expert R developers, like *R Packages* by Hadley Wickham [24], while the official resource on writing packages to share through CRAN is the manual, *Writing R Extensions* [85].

Regardless of whether you're writing a package to share, or one for your own use that you don't plan to publish through a repository, consider using version control. Among the numerous reasons why version control will positively impact your workflow, you'll be able to easily share your project with collaborators, at any point in time. If you decide to collaborate on a package, even if you just intend for the package to remain a resource among your group of collaborators, the Project framework is compatible with distributed development—a feature that couples well with version control and online platforms for version controlled repositories, like GitHub. Version control also helps you maintain a clean directory of code without limiting the process of making changes to files. All changes are trackable and rewindable so earlier versions are restorable. You can create offshoots of your main project (branches) to try new things and incorporate (some or all of) the changes into the main branch. As you work, keep in mind that within the open source R community, there is no lack of effective organizational frameworks to reference; in fact, repositories for many exemplary packages are readily available on GitHub.

Conclusion

The open source movement is the heartbeat of R; it sustains and enlivens the growth and improvement of the R ecosystem. Anyone, whether they are motivated to propel a research program or interested in learning from data in other contexts, can be an R user—without significant barriers to entry. A number of R users don't stop there, however. They become R makers by adding their own contributions so not only they,

but new and existing users and makers, can do more in R. This plays an essential role in the lifecycle of R's beloved shared tools.

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

7. 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.

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

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. Gentry J. TwitteR: R based twitter client [Internet]. 2015. Available: https://CRAN.R-project.org/package=twitteR

14. Premraj R. MailR: A utility to send emails from r [Internet]. 2015. Available: https://CRAN.R-project.org/package=mailR

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

16. 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

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

18. 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

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

20. 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/

21. Liaw A, Wiener M. Classification and regression by randomForest. R News. 2002;2: 18–22. Available: https://CRAN.R-project.org/doc/Rnews/

22. Morgan-Wall T. Rayshader: Create maps and visualize data in 2D and 3D [Internet]. 2020. Available: https://CRAN.R-project.org/package=rayshader

23. Broman K. Getting your r package on cran [Internet]. 2020. Available: https://kbroman.org/pkg_primer/pages/cran.html

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

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

26. Anderson GB, Bell ML, Peng RD. Methods to calculate the heat index as an exposure metric in environmental health research. Environemtental Health Perspectives. 2013;121: 1111–1119. Available: http://ehp.niehs.nih.gov/1206273/

27. Bioconductor: Open source software for bioinformatics [Internet]. Bioconductor; 2020. Available: https://www.bioconductor.org/

28. Falcon S, Morgan M, Gentleman R. An introduction to bioconductor's expressionset class. 2007.

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

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

31. rOpenSci, Anderson B, Chamberlain S, Krystalli A, Mullen L, Ram K, et al. Ropensci/dev_guide: Fourth release [Internet]. Zenodo; 2020. doi:10.5281/zenodo.3749013

32. 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

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

34. 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.

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

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

37. 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.

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

39. 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.

40. Pebesma E. Simple Features for R: Standardized Support for Spatial Vector Data. The R Journal. 2018;10: 439–446. doi:10.32614/RJ-2018-009

41. Schloerke B, Allaire J, Borges B. Learnr: Interactive tutorials for r [Internet]. 2020. Available: https://CRAN.R-project.org/package=learnr

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

43. Wickham H, Hesselberth J. Pkgdown [Internet]. RStudio; 2018. Available: https://pkgdown.r-lib.org/

44. Wickham H, Hesselberth J. Pkgdown: Make static html documentation for a package [Internet]. 2020. Available: https://CRAN.R-project.org/package=pkgdown

45. Tierney N, Cook D, McBain M, Fay C. Naniar: Data structures, summaries, and visualisations for missing data [Internet]. 2020. Available: https://CRAN.R-project.org/package=naniar

46. Riemondy KA, Sheridan RM, Gillen A, Yu Y, Bennett CG, Hesselberth JR. Valr: Reproducible genome interval arithmetic in r. F1000Research. 2017; doi:10.12688/f1000research.11997.1

47. Didelot, Xavier, Fraser, Christophe, Gardy, Jennifer, et al. Genomic infectious disease epidemiology in partially sampled and ongoing outbreaks. Molecular Biology and Evolution. 2017;34: 997–1007. doi:10.1093/molbev/msw275

48. RDocumentation [Internet]. DataCamp Inc. 2020. Available: https://www.rdocumentation.org/

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

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

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

52. 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

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

54. Wickham H. Advanced r. CRC press; 2019.

55. Eddelbuettel D. Seamless r and c++ integration with rcpp. Springer; 2013.

56. GitHub docs [Internet]. GitHub, Inc. 2020. Available: https://docs.github.com/en

57. 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/

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

59. Firke S, Krouse B, Grand E, Shepherd L, Ampeh W, Frick H. Packagemetrics: A package for helping you choose which package to use [Internet]. 2017. Available: https://github.com/ropenscilabs/packagemetrics

60. Becca Krouse HF Erin Grand. Packagemetrics - helping you choose a package since runconf17 [Internet]. rOpenSci; 2017. Available: https://ropensci.org/blog/2017/06/27/packagemetrics/

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

62. Ellis B, Haaland P, Hahne F, Le Meur N, Gopalakrishnan N, Spidlen J, et al. FlowCore: FlowCore: Basic structures for flow cytometry data. 2019.

63. Hahne F, LeMeur N, Brinkman RR, Ellis B, Haaland P, Sarkar D, et al. FlowCore: A bioconductor package for high throughput flow cytometry. BMC bioinformatics. BioMed Central; 2009;10: 1–8.

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

65. 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

66. 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

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

68. 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/

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

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

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

72. Mastering software development in r [Internet]. 2017. Available: https://rdpeng.github.io/RProgDA/

73. 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

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

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

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

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

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

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

80. Robinson D. Broom: An r package for converting statistical analysis objects into tidy data frames. arXiv preprint arXiv:14123565. 2014;

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

82. 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

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

84. 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

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