R Package Validation Framework

phuse.global

Table of Contents

Doc ID: WP-059 | Version: 1.0 | Working Group: Data Visualisation & Open Source Technology | Date: 15-Dec-2021

1: Introduction

In this white paper, we describe a framework to support the validation of user-contributed software extensions for the R programming language. This framework turns the process of validation into a part of the software development life cycle (SDLC), making validation an efficient and user-friendly process where the generation of proof that the software can consistently meet the requirements of the users can be done at the click of a button.

Validation is made user-friendly by breaking the foundations of validation into easy-to-follow steps and modular files that allow for easy updates and modifications without having to redo the entire process. At the end of the process, all the files produced by following this framework get combined into a final report document. By taking this modular approach, the massive amount of rework across multiple files to create the validation packet becomes a thing of the past, increasing flexibility and consistency, without compromising integrity. Storing these elements in a clear and consistent location allows for portability across projects and reduced cognitive load in traversing the validation landscape. Finally, automating the validation report generation creates an environment where iteration and development are encouraged.

This white paper describes the approach specifically for internally developed R packages. However, these same concepts are generally applicable to other works that require validation including external R packages, packages or modules in languages other than R, and entire software development environments.

2: Definitions

R: A free and open-source extensible programming language used to perform statistical computation, data manipulation and generation of figures.
R package: A structured collection of files used to share collections of functions, manuals and instructions between programmers.
CRAN: Comprehensive R Archive Network – a controlled online repository for R packages where users can install R packages that were approved by CRAN maintainers for general consumption.
SDLC: Software Development Life Cycle – the process by which software is updated, tested and released for use.
Working directory: The directory in a file system that the code is executed in and where file paths are referenced from.
Requirements: A clearly defined goal or expectation of behaviour the software is to achieve to be considered complete. Any contextual knowledge for understanding the requirement is either included or a reference is identified.
Specifications: A collection of approved, documented requirements.

3: Background

Validation – When and Why

The purpose of this document is to describe a development process to follow when creating and validating an R package. However, what validation is and when to be concerned with validation are two critical pieces to consider prior to embarking on this venture.

Validation is generating objective proof that the specifications (a set of requirements) meet users' needs and the software can consistently satisfy those requirements.[1] To this end, validation is not simply a box to tick, but a process to be followed to ensure that the software is doing what it set out to do, and that the users' needs are being met. For the purposes of the framework, objective proof is considered to be a document containing:

- detailed requirements
- test cases showing how to prove the requirements have been met
- records of the successful execution of the test code that implements the test cases
- signatures of the individuals involved and key stakeholders approved of this documentation.

Another factor to consider is when to perform validation as not all software nor its inherent risks and impacts are created equal. A prudent approach to validation is that efforts for validation are based on assessed risk of the software.[1,2,3] For example, the word processing software used to write the report does not require validation, but the software calculating the values or generating the tables used in decision-making should.

About R and R Packages

R is a free and open-source programming language and software environment for statistical computing and graphics that is supported by the R Foundation for Statistical Computing. Gaining popularity in recent years for analysis and data science, R compiles and runs on a wide variety of UNIX platforms, Windows and MacOS.

The base R source code and its recommended packages can be considered highly trustworthy. With a small set of programmers approved to make additions or changes to the core language, development through a standardised SDLC that includes thorough unit testing, and millions of programmers across the world using R, risks are assumed to be minimal. The R Foundation has previously released a white paper defining their views on regulatory compliance with the FDA for R core and providing more details around R's SDLC.[4]

One of the most powerful features of R is its extensibility through shared code. This allows users to quickly add new statistical methods or abilities to the R language with minimal barriers or having to write their own code to solve the same problem. Similar to many other programming languages, the fundamental unit of shareable code in R is the "package". A package bundles together code, data, tests, examples and documentation into a single location, making it easy to share with others. Any user can write a package and share it with their

R community. Additionally, users can install packages easily from within R using a few commands. Once the package is installed, it can be easily accessed and used.

This makes R an ideal language for use in the pharmaceutical industry. Consistency across different groups can be assured by creating and sharing internal packages that provide organisation-specific information to how processes are performed. Versioning of packages allows for a historical reference, and updates can be tracked using technology such as version control software.

To this point, there has not been a well-described process for developers to follow when creating packages to capture the information necessary for validation. This is where the R Package Validation Framework comes in, offering a portable framework to allow organisations and developers to create a validation infrastructure. This framework can be integrated into packages or be applied generally to environments and is shareable with the R community.

4: Package Validation

The Validation Framework

This framework has been applied to the development of internal packages at the Statistical Center for HIV/AIDS Research and Prevention (SCHARP) and public packages developed by Atorus Research. The process described here is the refinement of the original idea into the critical elements necessary to apply the framework successfully. In total, there are five steps within the validation framework whose outputs are combined to produce a validated package: Requirements, Package Development, Test Cases, Test Code and the Validation Report.

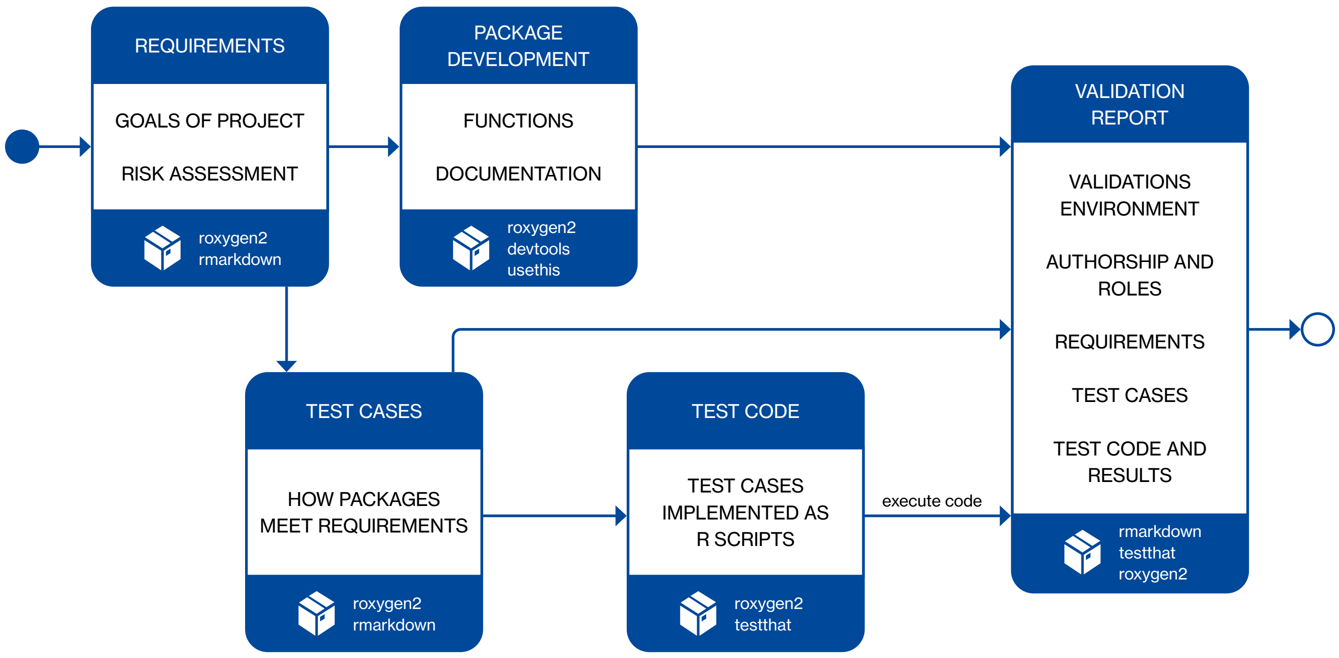
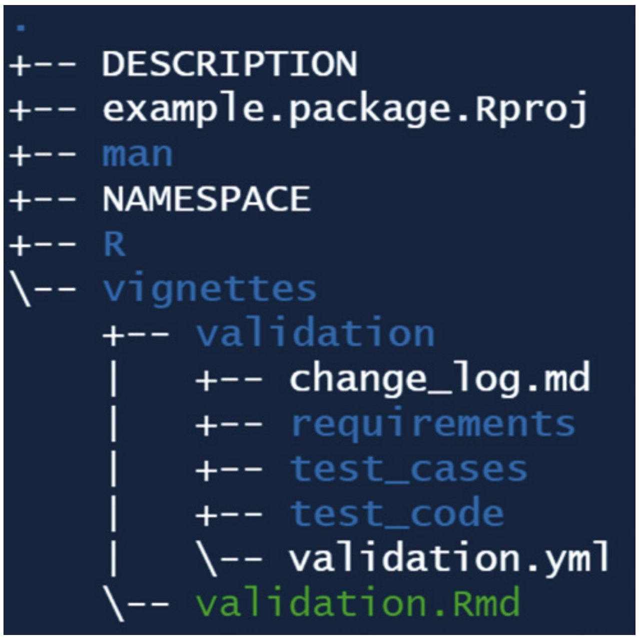


Figure 1 Flow Diagram of the R Package Validation Framework

The value of this framework not only comes in providing a series of clear steps to follow for validation, but a clean and consistent file structure to organise all the files necessary for validation. Below is an example of an R package folder structure with the validation framework applied to it.



```
.
+-- DESCRIPTION
+-- example.package.Rproj
+-- man
+-- NAMESPACE
+-- R
\-- vignettes
    +-- validation
    |   +-- change_log.md
    |   +-- requirements
    |   +-- test_cases
    |   +-- test_code
    |   \-- validation.yml
    \-- validation.Rmd
```

Figure 2 Example R package folder structure with the R Package Validation Framework infrastructure added

Requirements

A requirement is any need or expectation for a system or software.[1] Through requirements, the goals and expected outputs of the software are shaped and are then able to be validated. Without requirements, there is no clearly defined scope to test against or programming to verify, which prevents an effective validation and reduces the quality of the package.

To write requirements, collaborate with subject matter experts (SMEs) and end users to yield clear requirements that capture as much contextual information as possible to convey any nuance. If the information cannot be explained concisely, pointing to external resources to provide additional guidance is an acceptable solution. Before considering a requirement complete, gain approval from the SME and stakeholders to ensure that they meet user needs, and are of consistent quality across the requirements.

Requirements are saved in a file format that is both human- and machine-readable, such as markdown.[5] Choosing a file format that can be opened without special or proprietary software democratises the involvement of the project members. Making the files machine-readable allows the code that generates the validation report to source these files into the output document, allowing it to represent the latest iteration of the requirements without duplication of effort.

Record when each requirement was first written or edited and who performed the editing within the header of the requirements file. By capturing this information within the file, updates to ownership and edit dates can happen at the same time as the changes occurring to the file. Now, any future users who review the requirements and have questions can identify the editor without having to use an external tool. Additionally, the header information will be read by the validation report to assign credit and track roles in validation.

As part of writing the requirements, risk assessments are performed. Risk assessments determine the likelihood of a defect in the software based on requirements, and the impact if it occurs. Record the assessment within the requirement header with the editor and edit date so it too can be read by the code that generates the validation report. These assessments drive later testing as the riskier the requirement, the more thorough the mitigations need to be to reduce risk.

Save all the requirements in the same folder under the working directory for validation. The name of this folder is suggested to be called “requirements” for clarity and distinction from the rest of the validation elements. In an R package, the parent directory of the validation working directory should be the “vignettes” folder, with the working directory being a folder called “validation”. (See Figure 1.) When validation is performed outside of an R package, the working directory is the folder designated to hold the validation contents.

Package Development

This step is only applicable when there is software to write. If there is existing software written by a third party that meets the requirements, this step can be skipped.

During development of an R package that is to be validated, who last edited and when the last edit was made to the function needs to be captured. The purpose of this is to track ownership and roles of the package across time for appropriate attribution of responsibility.

A requirement may be fully met by a single function or by the sum of many modular functions. When writing the software package, it may be best to have a combination of both approaches to support the nature of requirements and writing clean reusable code. Carefully review the requirements and make sure all requirements have been met.

A method for capturing this information is to add headers and comments around each function to document the ownership and any other information about the package. This aligns with self-documentation tools that are common in programming. For R, self-documentation through comments is supported through the package {roxygen2}.

Documenting who edited and when the function was edited next to the function itself has many different layers of benefit. The header is easy to update at the time of editing the function and is documentation that does not require advanced tools such as version control systems. Next, any other programmers who run into difficulties when extending this function have the documentation within the package of who to contact with questions. Finally, it is useful for validation to track the roles of the individuals across the project, and this is another way to make sure people who are involved with writing code are not involved with testing the same code. The actual implementation of ownership in this model assumes that when an edit is made, the owner is responsible for the entire function.

There are many opinions and approaches that can be followed during R package development, each with their own benefits and drawbacks. The purpose of this white paper is not to define good programming practices, but rather how to integrate the validation framework into package development. For resources on good package development and suggestions from these authors, refer to Appendix 1: R Package GPP.

Documentation in both long form (vignettes) and short form (function manuals) that are important to support validation efforts but fall outside the scope of this white paper. Refer to Appendix 2: Package Documentation Types for more information on package documentation.

Test Cases

Once the software has been developed, it now must undergo thorough testing to ensure that software executes correctly outside the developer's environment. This testing is known by the term "User Site Testing"[1] and is essential to validation to prove that the software meets the requirements with the actual hardware and external software on the intended system.

To support the testing, a pre-defined plan describing the testing to be performed may need to be written. The plan is comprised of test cases that demonstrate the software meets the requirements by defining input data, processing steps to follow and exact outputs expected.

Every requirement must have evidence that it has been met, which means every requirement needs to have support from at least one test case. The number of test cases that are written for each requirement are based on the identified risks. The higher the risk, the more test cases may be created to mitigate the risk by showing the ways that the code could go wrong. A well-written test case can support multiple requirements, adding to overall coverage and ensuring the software works well together while reducing overall effort.

A well-written test case confirms the understanding of the software and how its functionality meets the requirements. The test case should be written in such a way that a person with reasonable knowledge of the programming language can implement the test case without internal knowledge of the system being tested. Be specific and write cases that are representative of how a user may utilise the program since that will be the most helpful to uncovering defects and supporting test automation. To ensure that the requirements are met, explicitly indicate the expected output and tests to be performed to show that the test case passes.

Suppose we have a function, 'hello_world()', that has a single argument called 'name'. When the function is called, it will return text stating 'Hello, {name}!', where 'name' matches the argument provided by the user. An example of a test case may be 'Say hello to the user 'Sam' using the function 'hello_world()' by setting the argument 'name' to be 'Sam'.' The value that is returned from this function is a character string of the value 'Hello, Sam!' Note how there is no code provided, but the steps for the programmer to follow to implement the test are provided and the exact expected output is defined.

The successful execution of the collection of test cases proves that every requirement is being met by the code. This collection is reviewed by the key stakeholders and SME to ensure the collection is both representative and complies with their understanding of the requirements to show their needs are met.

Test cases are saved in a human- and machine-readable file format, and it is suggested to use the same file format as was used in the requirements. The editor of the test case, when the last update to the test case was made, and which requirements are being met by which test case are all recorded in the header of the file.

Save all the test cases in the same folder under the working directory for validation, in a separate folder from the requirements. The name of this folder is suggested to be called "test_cases" for clarity and distinction from the rest of the validation elements.

Test Case Considerations

Writing test cases is as much an art as it is a science, when aiming to ensure full coverage and being informative without being prescriptive. It is not feasible to test the complete set of possible inputs for a function, so it is not reasonable to attempt to test every possible set of inputs and test cases. Writing test cases relies on creating representative conditions or explicitly checking for known edge cases that can be used to provide coverage for a set of possible inputs. Good test cases would subtract from the set of assumptions made about the functionality and expand the set of known behaviours.

Test cases cannot prove a package is faultless, rather they are to confirm that the requirements laid out have been satisfied. A suite of tests that fails is often more valuable than one that passes. Test cases contribute to the quality of a system by uncovering problems with documentation, assumptions, underlying dependencies and the code itself. Test cases that are failing can help any developers and testers find where issues are happening and fix them in a timely manner.

Test Code

The test code is the written implementation of the test cases. By writing the test cases out as reproducible snippets of code, unbiased and automated evaluation of the tests and capturing of the results can occur. The code is written as digestible code chunks that demonstrate the functionality of the software and prove the requirements have been met. Well-written test code should have three main goals in mind: simplicity, clarity and repeatability.

Simple test code means only the code necessary based on the test cases is recorded. The goal of the test code is to implement the test case and reliably capture the results. Keeping to using simple functions that may not be efficient but are verbose and clean makes the process of debugging test code failures much easier and faster.

Writing code that makes it clear how the test case maps to the code simplifies review and updates when test cases change. Format the test code into easily digestible chunks that correspond to a test case so that users looking at the documentation can easily follow the process. Add comments in the test code to describe what is happening in the code and how that matches the test case.

The purpose of writing the test code in reproducible scripts allows for user site testing to become automated. The code should be able to be rerun to check that any changes made from version to version did not have any unintended effect on the functionality of the package. There should be no permanent changes to the environment outside of the test being run. Changes may occur within a test, but the environment must revert to the original state at the completion of the test.

An important part of user site testing is the evaluation of the ability of the users to understand and interact with the software.[1] The test code writer was not involved with writing any of the package code or the test cases, and as such is a valuable resource for reviewing the quality of the package. Because they do not have in-depth knowledge of the workings of the package, they can give feedback as a new user to the package documentation and check that assumptions made by the developers hold as true, as they write the test code.

For each test case that the test code writer turns into code, the author of the test code and the date it was written is captured, with the code as a header to each test section. Every time the code is updated, the author and date are updated in the file. This information will be captured by the validation report.

Each test code file is saved as an executable script in the same folder under the working directory for validation, in a separate folder from the requirements or test cases. The name of this folder is suggested to be called “test_code” for clarity and distinction from the rest of the validation elements.

Validation Report

Authoring the Validation Report

The validation report is the objective evidence that the package can consistently meet the requirements, and thus the needs of the users. It is done by compiling all the different files that have been written across this process into a file that can be signed off by the individuals involved.

To create the validation report, we take advantage of the code-executing and document-generating abilities of R Markdown and write the validation report source code. Using basic commands, each file that was created through the prior steps is parsed to extract editor information and evaluate the test code to gather results, combining the raw text from the requirement and test case files to produce a final report.

Because the validation report source code is written in R Markdown, it is infinitely customisable to the organisation’s requirements. For example, the report can be modified to include a description of the testing system, the operating system and version of any dependencies, a table of the validation team and their responsibilities, or record the change history of the validation. Additionally, the template that the rendered document is based on can be customised to include the organisation’s letterhead and logos.

This validation report R Markdown document serves as the scaffold for the final validation report, and validation is not complete until this is compiled into the validation report for the release of the code.

Generating the Validation Report

Once the validation report source code has been written and approved, the next step is the compilation of this document into the final validation report for this version of the package. Each time the software is updated and is up for validation, the version number of the package is incremented, and the report is recompiled and circulated for approval. This ensures that the validation aligns with the version of the software being used.

The preferred method for storing the validation source code is as a vignette. A vignette is an implementation of long-form documentation, in which R Markdown files are placed in the “vignettes” directory of the R package. Files in this directory may be rendered as part of a package release and/or at installation using native R installation tools.

On a system with this R package installed, the user can access the rendered document from within the active workspace. This makes “vignettes/” an ideal location for files that make up the validation report source, capturing requirements, test cases, test code and test code results while rendering everything into a human-readable document.

If the validation files have been copied to “inst” after generating the validation report, the elements as they were to generate the original report are preserved on package build in the bundled package file or binary file. Now, validation of the installed package can be done by executing the validation report source code within the package, with paths updated to reflect the new location.

Validation Types

Here, we consider three different approaches for when an organisation might want to compile the validation report source code to generate the validation report: version release, install, and for re-validation of an installed or binary package.

An important piece to understanding these validation types is to understand the state that a package may be in during or after release and how files and information move within a package as it moves from a source to a bundled, binary, and eventually, installed package.[6]

On Version Release

This is the case where developers generate the validation report and circulate the document for approval as the last step of releasing a particular version of that package. Validation at this level does not consider the details of a user’s environment; however, it does demonstrate proof-of-concept that the validation test cases execute in the developer-specified environment. This approach is closest to current practices for packages distributed via CRAN repositories and means that the vignette is pre-rendered. The user may run the validation on the source code and leave it as is or compile it into a bundle to ensure source code changed pre-installation not post-validation of the code.

On Install

This validation is for cases when the proof of validation needs to be generated within the environment the package is intended to be used in, and the team performing the installation has access to the source code. For cases where validation is performed "on install", whomever is running the validation can get the compiled vignette to acquire the necessary signatures or documentation to meet the organisation's regulatory requirements.

After Installation

The final validation mode is for cases where the package has been installed in the environment it is being used in, but access to the source code is not available or restricted. This may be the case where proof needs to be regenerated because the state of the environment has changed, but the package to be validated has not.

In these cases, once the validation report has been rendered from the vignettes folder, inside the "inst" folder of the R package, create a new "validation" folder, and copy all the elements required for validation into this new folder. If the folder already exists, overwrite the existing contents with the new files. This includes the specification files, test case files, test code files, the roxygen headers of functions, and the source code that generates the validation vignette.

Once all the contents exist inside the "inst"/"validation" folder, the package can be built and installed as with the other packages. The unique case here is that contents within the "inst" folder of an R package are kept with the package but move up a folder level. Now, the contents of "validation" can be accessed from within the installed package directory and rerun to ensure the validated package maintains its behaviour even when the environment changes.

Other Considerations

Expectations Imposed by Distribution Platform

When preparing the R package for release via an official repository, e.g. CRAN, the expectation is that the package passes the R's built-in command line package testing suite with no errors. One of the checks is that all vignette code runs using the code in this release. Note that for CRAN specifically, the rendered vignette document is expected to be provided by the developer, and during testing only the R code chunks in the vignettes are executed to ensure the vignette runs but the document is not regenerated.[19]

A R package can be released in one of several formats: source, bundled or binary. When users install from source or bundled, there is the option of rendering vignettes at the point of installation. If starting with a binary, the vignette is not built at installation and relies on the instance uploaded by the developer.

Future Releases

The process described thus far is a linear start to end. However, no software projects avoid bugs or have zero new feature requests after the initial release.

The framework readily supports this because of the independent structuring of the requirements, test cases and test code. As new features are requested, requirements are updated or added, test cases are written to ensure full coverage still, and testers update the test code.

As requirements are no longer necessary and become deprecated, there can be two choices depending on the organisation. One method is that the requirements, test cases and test code are removed or updated as necessary, and it is recorded in the report that the original requirement has been removed. The other method is to simply mark the requirement as deprecated in the file and mark the test cases and code accordingly to preserve the record.

When the elements have all been updated, the validation report can be recompiled according to the organisation's validation practices and the package continues in its newly validated state.

References

1. *https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-principles-software-validation*
2. *https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-application*
3. *https://www.pharmar.org/presentations/r_packages-white_paper.pdf*
4. *https://www.r-project.org/doc/R-FDA.pdf*
5. *https://rmarkdown.rstudio.com/authoring_pandoc_markdown.html*
6. *https://r-pkgs.org/package-structure-state.html*
7. *https://stat.ethz.ch/R-manual/R-devel/library/base/html/Round.html*
8. *https://stat.ethz.ch/R-manual/R-devel/library/stats/html/quantile.html*
9. *https://r-pkgs.org/r-cmd-check.html*
10. *https://docs.python.org/3/library/doctest.html*
11. *https://r-pkgs.org/vignettes.html#vignettes*
12. *https://devtools.r-lib.org/*
13. *https://roxygen2.r-lib.org/*
14. *https://pkgdown.r-lib.org/*
15. *https://xml2.r-lib.org/*
16. *https://usethis.r-lib.org/*
17. *https://www.tidyverse.org/*
18. *https://www.sphinx-doc.org/en/master/index.html*
19. *https://r-pkgs.org/vignettes.html#vignette-cran*

5: Disclaimer

The opinions expressed in this document are those of the authors and should not be construed to represent the opinions of PHUSE members, respective companies/organisations or regulators' views or policies. The content in this document should not be interpreted as a data standard and/or information required by regulatory authorities.

6: Appendices

1. R Package Good Programming Practices:

Here, the authors of this white paper document some opinions for how to approach developing a well-designed, well-tested package that will serve your organisation well. In addition to this appendix, read "R Packages" by Hadley Wickham for an in-depth review of state-of-the-art package development philosophies and tools.

Design

As with any software project, a design phase is a critical part of deploying an R project. Extensibility and life cycles are built into the R ecosystem and can be taken advantage of if the development stage is thought through. Like all software projects, general advice about designing functions and objects is appropriate. R packages rely on several methods for implementing object-oriented principles and can be written in a functional style.

It is rare for R packages to operate on their own. Most will extend or rely on other packages, or users will use packages together to pursue a solution. The first step in designing an R package is to map out what will work with other packages, allow its own methods and objects to be operated on, and when to encapsulate logic that is intended to be internal.

An important design consideration is the management of NAMESPACE conflicts – packages that load functions with identical names. The order in which packages are loaded into a script matters a great deal in how an R script will run. When packages are loaded, their NAMESPACE is inserted into the environment, meaning if two packages have overlapping function names, the function loaded second will override the behaviour of the package loaded first. This can cause unexpected behaviour, especially when NAMESPACEs include functions such as 'sort' and 'sum'. A common solution for NAMESPACE conflicts is function prefixes, which is prefixing all functions with something relating to the package. Most user functions in the XML2 package are prefixed with 'xml_'; similarly, most functions in the usethis package are prefixed with 'use_'.[15,16]

Life cycles are separate from any functionality in the package, but signal your intentions to maintain and improve a package or your intentions of keeping certain function interfaces stable. As it is important to inform your potential users of your intentions, it is advisable to not rely on any package that is experimental or depreciated.

Development

Documentation

The {roxygen2} package presents an interface to manage documentation and package exports along with code in the R/ directory instead of the manual pages and NAMESPACE file. This process of documentation provides documentation, along with the context of the code itself, and removes the issues of coordinating markdown, namespace and R code.

Tests

Part of package development includes developing a testing suite to prove the contents of the package are behaving as expected from the perspective of the developer. The testing suite is built out of a combination of unit tests and regression tests.

Unit tests are individual tests that check functionality on the smallest "unit" possible. This is typically at the function level. A variety of inputs and checks should be performed, confirming behaviour for both correct and invalid inputs. Regression tests combine multiple units to confirm that changes in one function are reflected in the expectations of any inputs. This wide and deep testing provides the best possible coverage of the package code and is the first line of defence to protect from unintended bugs being added into the system.

Test scripts are found in the "tests/" directory of the R package and can be run a few different ways. The most basic way is to load all the package functions and execute the tests manually. However, R comes with a native package testing suite – R CMD CHECK – which runs all tests in addition to reviewing package contents for conformity with standard package-building conventions.

The testthat package is a popular interface for designing tests with intuitive functions for labelling tests and defining expectations. It has a clear syntax for modularising tests, declaring expectations, and helps automate the testing cycle.

Although both unit tests and test cases/code are used to test the functionality of the software package and its contents, they have distinctly different goals. Unit or regression tests are built to test if specific pieces are working as intended from a programming standpoint. Unit tests may also test that any expected errors and warnings are happening in the intended manner. Test cases on the other hand test that the code meets the requirements of the software in the end user environment and inform the test code that is written by a third party. Test cases should also include tests that include multiple pieces working at the same time if that is the intended functionality.

Integration

The R language download comes with utilities for installing, building and checking packages. During submission to CRAN, a package is reviewed initially using a native R package testing suite to check the package for consistency, complete documentation and tests. However, running these checks frequently during development is an easy way to improve the success of any R package. The 'devtools::check()' function offers an easy R interface without having to work with the CLI directly and returns a non-zero code to allow for any continuous integration system.

Nearly every R package is built on a network of other packages, called dependencies. It may not be feasible to test every possible permutation of every package and every R version, and it may not be necessary. The most straightforward way to test a package is to test the package with the latest version of any dependencies your package has. This may not be realistic on a local computer; however, since this can cause issues with other versions of packages a user is using, a solution for this is to test the package with set dates to test releases by. Containers made by the Rocker organisation make this easy by providing dockerfiles that are frozen to the date of release of R versions. You can pair frozen package states with testing different OS versions. This would result in every specified frozen date being tested on every OS distribution. An example is shown below.

Operating System	R version 4.0.3	R version 4.0.0	R version 3.6.3
MacOS	MacOS running 4.0.3	MacOS running 4.0.0	MacOS running 3.6.3
Ubuntu 18.04	Bionic running 4.0.3	Bionic running 4.0.0	Bionic running 3.6.3
Windows 10	Windows 10 running 4.0.3	Windows 10 running 4.0.0	Windows 10 running 3.6.3

Integration can also include the building and deployment of documentation artifacts. The pkgdown package is designed to turn existing documentation, vignettes, news and readme files into a website that can host these files in an easily digestible and navigable format.

These tools are generally combined and automated into a process called “continuous integration”. The checking of compatibility with other packages, R versions and operating systems is combined with the rendering of documentation to shorten the time it takes from development to deployment. Discussion of specific tools to achieve this is out of scope for this paper; however, there are many functions that integrate these tools into an existing development process.

Deployment

Deployment from a local or development environment can be an unexpected source of issues when the production environment differs in unexpected ways. This can be an especially acute problem for R, where package versions change frequently. R packages can be thought of as a discrete unit during deployment and validation. Deployment for our purposes could be a deployment of an R package to CRAN for public use, deployment to internal users on a private repository, or deployment of an analysis product.

Documentation is a necessary requirement for a validated package. A package that lacks context and functional specifications would be difficult to validate. Consistency between function documentation and function inputs is checked with R CMD CHECK, as is the building of vignettes. The rendering of documentation is a key component of the deployment process. This is generally done by rendering the manual comments from the roxygen tags.

While the release of a package or analysis product was difficult to automate in the past, improvements in R infrastructure as of late have made this a straightforward process in many situations. The devtools package has a 'release' function that can transmit a package to CRAN for submission. The {golem} package has multiple functions that create the necessary files for deploying a shiny application to several hosting solutions.

2. Package Documentation Types

The process and documentation of validation during validation is extremely important, but equally important is the documentation of the package itself. While validation documentation such as requirements specify what the package should do, the package documentation explains to the user how those requirements are implemented and how the package should be used. Overall, package documentation can be broken down into two general levels, applicable to most programming languages: function documentation and long-form documentation.

Function Documentation

Function documentation provides usage information applicable to a single function or a group of related functions. Similar functions can be grouped into the same set of documentation when they are closely related. For example, a group of functions may share the same parameters and may be used in very similar contexts. Instead of duplicating this information in multiple locations, it may make sense to group that documentation together in one place.

Function documentation should answer specific questions about the use and functionality of a function, such as what the function is called, what the inputs should be and what the function will return. Function documentation can be broken down into a few different sections.

Summary

A summary of the function should be provided to give a high-level overview of a function’s purpose, intended usage and intended result. This information should be concise as this is the first place a user should look to see if a function fulfils their needs. Keep this section to a short paragraph with a maximum of no more than three or four sentences.

Parameters

One of the most crucial aspects of function documentation is the documentation of parameters. Every parameter available in a function should be documented. The documentation should explain the expected input to that parameter and the purpose of the parameter. If the parameter is optional, this should be specified and explain the impact of using that optional parameter.

Details

When a function or group of functions warrants further explanation, a details section may be necessary. The details section allows the documentation to elaborate further on information that would not fit within the summary or within the parameter documentation. This could be information on specifics of the implementation of the function or references to relevant literature. For example, the 'round()' function in R details that the IEE 60559 standard was used for rounding off a 5.7 This is critically important information to understand but is too detailed for a summary section. Similarly, the 'quantile()' function details equations of each of the 9 quantile algorithm types available within the function.[8]

Examples

Lastly, great function documentation includes example usage of the function in different scenarios. This is highly advantageous to the user as it gives practical examples of what to call the function, contextual usage of the function, and even code that can be used as a starting point for the user. Furthermore, example code is testable and can be used in automated frameworks like the R CMD Check,9 or the Python doctest10 library. Examples also clearly communicate expectations to a validator. This section is an opportunity for a developer to show a validator what the function was intended to be called and can help reduce back and forth communication.

Long-form Documentation

A sometimes overlooked, but equally important, section of package documentation is the long-form documentation. The R programming language has a special framework for this, called vignettes.[11] With tools like R Markdown, creating vignettes is quite simple, allowing you to put text, code and styled output such as tables and graphs all within the same document.

While function documentation is critical for a user to know what to call a function, packages are usually made up of many functions – and these functions will typically work together in some way. Long-form documentation ties together the bigger picture to explain to a user how functions should be used together contextually and can serve several purposes, from giving basic information to getting a user started on a package to explaining advanced usage scenarios in detail.

While long-form documentation offers more space to provide information, this space should still be used responsibly. It is quite easy for these documents to get long and unruly, which makes specific usage information or particular scenarios being explained difficult to find. Long-form documentation is still best served when the information is concise. When a document is getting too detailed and long, it may be best to split it into multiple documents of sub-topics or consider regrouping the information into smaller related sets. Long-form documentation should still be a reference rather than training, so keep in mind that a user will likely be using this documentation with a specific question in mind. Therefore, build the document to be navigable and intuitive.

Additional Tools

The open-source landscape has a number of tools available to aid in the production of documentation. With the R language, packages like devtools[12] and roxygen2[13] help with package development and documentation. Furthermore, packages like pkgdown[14] are available to take function references and vignettes and build a website out of that content. Resources like this are helpful as it offers a richer user interface than the base R language. All of the tidyverse[15] and many other R packages are produced using these package development and documentation tools. Similarly, Python has tools such as Sphinx, which is a tool originally developed for Python that creates rich documentation and has since expanded into multiple languages.[18] While many of these tools aim to produce websites or contribute documentation to open-source forums, these resources could just as easily be leveraged within internal systems and provide internal users with rich references beyond simple PDF documents.

7: Project Contact Information

- Ellis Hughes
- *ellishughes@live.com*

8: Acknowledgements

This white paper was developed with the assistance, contributions and feedback from the members of the R Package Validation Framework Working Group Project: Aiming Yang, Chris Battiston, Eli Miller, Katherine Ostbye, Marie Vendettuoli, Mike Stackhouse, Nathan Kosiba, Paul Stutzman, Peyman Eshghi, Phil Bowsher, Rafael Kuttner, Shrishaila Patil, Srinivasa Pilarisetty, Stella Guo and Ting Zhang. Without these individuals' diligence, input and imagination, this framework would not be where it is today.