R Development Guide

R Contribution Working Group

2022-10-20

Contents

Chapter 1

Acknowledgement

This guide draws on documentation and articles written by the R Core Team. Initial chapters of the guide were developed by Saranjeet Kaur Bhogal, in a project funded by the R Foundation, mentored by Heather Turner and Michael Lawrence. This initial version was upgraded in a Google Season of Docs 2022 project with Saranjeet Kaur Bhogal and Lluís Revilla Sancho working as technical writers managed by Nicolas Bennett and overseen by a Steering Committee including representatives from R Core and the wider R community.

This guide has benefited and continues to be benefit from varied contributions by several contributors.

Chapter 2

Introduction

This guide is heavily influenced by the Python Developer Guide, and is a comprehensive resource for contributing to R Core – for both new and experienced contributors. It is maintained by the R Contribution Working Group. We welcome your contributions to R Core!

How to contribute to this guide?

This guide is built using bookdown which makes editing it easier, provided you have a GitHub account (sign-up at github.com). After you log-in to GitHub, click on the 'Edit' icon highlighted with a red ellipse in the image below. This will take you to an editable version of the the source R Markdown file that generated the page you are on:

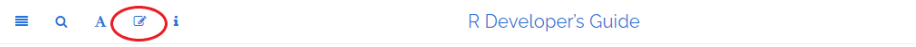

Figure 2.1: Screenshot of the toolbar in the HTML version of the guide, with the Edit button (pencil and paper icon) circled in red.

Use the issue tracker to raise an issue about the guide's content or to make a feature request.

Maintainers and contributors are requested to follow this project's code of conduct.

Chapter 3

Getting Started

These instructions cover how to install R in Windows. The tools required to build R and R packages in Windows are also discussed.

3.1 General instructions

1. If you install the latest version or R-patched or R-devel, it will not overwrite the previous installation(s) in your Windows machine.
2. R uses a 'major.minor.patchlevel' version numbering scheme. Accordingly there are three main releases of R available to install:
    - The official release (`r-release`),
    - The patched release (`r-patched`), and
    - The development release (`r-devel`).

The `r-devel` is the next minor or eventually major release development version of R. Mostly, bug fixes are introduced in `r-patched`, while `r-devel` is for introducing new features.

3.2 Installing R

1. The binary builds of R for Windows can be downloaded and installed from here. Along with the link to the latest stable release, this page also contains links to the binary builds of r-patched and r-devel.
2. Click on the download links to download an executable installer.

3. Select the language while installing, read the GNU general public license information, and select the destination location to start the installation. You will be prompted to select components at this stage: `User installation`, `32-bit User installation`, `64-bit User installation`, or `Custom installation`. The default option may be chosen for the questions from this step onwards to complete the installation.

3.3 Building R and R packages

3.3.1 What tools do you need to build R from source on Windows?

1. RTools is the toolchain bundle that you can use to build R base and R packages containing compiled code, on Windows.

2. You also need a distribution of LaTeX installed for building R and checking packages. The MiKTeX distribution of LaTeX that is used on CRAN can be downloaded from https://miktex.org.

3.3.2 How to setup `RTools`?

1. The latest version of `RTools` can be downloaded from https://cran.r-project.org/bin/windows/Rtools/ and run in the Windows-style installer. You will need to know if you have a 32-bit or 64-bit Windows machine (right-click `This PC` in Windows Explorer and check the properties if you are unsure).

2. Don't forget to add `RTools` to the path as documented on the download page.

3.3.3 How to build R?

To build R for Windows using `RTools` follow the instructions in this README file. There are two options available to build R. One is the quick development build and the other option is the full installer build.

For development and testing, you need only the quick development build. The quick build avoids building the manuals, which are generally not needed for development and testing.

However, even for the quick build there are some default requirements. For instance, MikTeX is to be installed in `C:/Program Files` and you have 64-bit R. If necessary, these defaults can be customised. The installation path of

MikTex can be customised here whereas the Windows bit can be customised here.

If you are a maintainer of the Windows CRAN releases then, the full installer build is available for building the complete installer as it appears on CRAN. It will build both the 32-bit and 64-bit R, the pdf manuals, and the installer program. You will use this to create the binary builds and not when building R from the source yourself.

3.4 How to download the R sources directly or from the svn repository?

- To download the R sources on Windows, you can use `tar` from the RStudio terminal.
- If you want to checkout the sources from svn, it is probably best to install an SVN client. Either TortoiseSVN (https://tortoisesvn.net/, command line tool, and Windows Explorer integration) or SlikSVN (https://sliksvn.com/download/, just the command line tool) is recommended. They have simple Windows installers and you can use svn straight-away from Windows cmd or RStudio terminal.

3.5 See also

1. CRAN official website
2. R installation and administration manual
3. R for Windows FAQ
4. Rtools40 manual for Windows
5. R FAQ

Chapter 4

Bug Tracking

4.1 What is a bug in R?

You may find a bug in R if:

1. The R session terminates unexpectedly, or there is a segmentation fault, it might be a bug in R, unless you have written your own call to compiled code or an internal function (via `.C` or `.Internal`). The error may look like this:

```
##  *** caught segfault ***
## address (nil), cause 'memory not mapped'
```

2. If the code does not do what the documentation says it should, then either the code or the documentation is wrong. Report either of which needs to be fixed.

Note: When you are in doubt that there is a bug: (which should be most of the time!)

1. Make sure whether the bug appears in a clean session of R. Many times, there are variables/commands/functions stored in history which might cause issues. Hence, check if the issue happens in a clean session. To do so, launch R from the command line with the `--vanilla` option.

2. At times the code that is written is very complicated, has numerous package and file dependencies, has many function calls, etc.. In such scenarios it is quite common that the code throws an error and you are not able

to solve it. You may tend to think that there is a bug that needs to be reported. Before doing so, try to produce a minimum working example of the code for the section where the error occurred. Add only those package and files which are required by that section, and see if the error appears still. Using this approach shall solve most of the errors.

3. Install R-devel, which is the most recent version of R from svn / git or daily Windows build, and see if your bug still exists in R-devel (it may have been fixed very recently).

4. Search on R-devel email list for messages with keywords related to your possible bug. If you find some related messages then read them to see if they clarify whether or not it is a bug. If you do not find any related messages, then please post a new message to R-devel. Your message should include (1) a brief description of the bug including current and expected behavior, (2) a minimal reproducible example.

4.2 What condition might not be a bug?

1. In case the code is doing something unexpected, it may not necessarily be a bug. Carefully review the documentation of the function being called, and check whether the behaviour being exhibited on calling this function is the same as it was designed to do.

2. Issues with *seemingly* identical numbers not being equal (especially floating point numbers) are usually not bugs.

3. If R is running slower than expected, then also it may not be a bug. Ask someone else to review your code in such a case.

4. If some function is working, but it is not defined in the best generalised way, then consult someone to look over your code. This may perhaps not be a bug; instead, it might be an alternative way of writing the function.

4.3 Checking if a bug is already reported

The first step before filing a bug report is to see whether the problem has already been reported. Checking if the bug is reported will:

1. Save time for you and the developers

2. Help you see if the bug is already fixed for the next release

3. Lead you to learn what needs to be done to fix it

4. Determine if any additional information is needed

The sections that follow discuss where to check whether a bug is already reported.

4.4 Levels of contributing to bug / What do you do when you find a bug?

1. Report the bug (if it is not already reported).
2. Test the bug.
3. Fix the bug: Fixing a bug might require relatively more time. You may start a conversation about it on BugZilla. This would require engagement with R Core team.

4.5 What are some places where you may find a bug?

You may find a bug in:

1. In the R-Core supported packages, their documentations, and/ or in the R language implementation.
2. In packages and/or their documentations which are not supported by the R-Core.

4.6 How to report a bug?

Once you confirm a bug exists, you need to submit a bug report so that it gets fixed.

4.6.1 Bug in the R-Core supported packages, their documentations, and/ or in the R language

1. Packages that are supported by the R-Core are labelled with `Maintainer: R Core Team <R-core@r-project.org>`. One simple way to get the information from R is by running the `maintainer("package_name")` command.

2. The bug report for R-Core supported packages, their documentations, and/ or a bug report for the R language itself can be submitted either to R-devel, see posting guide, or to Bugzilla. In the future, we hope to have an option to report an issue to the GitHub Mirror of R svn server.

3. If you want to submit the bug report using Bugzilla, please ensure that you have a Bugzilla account. To get a Bugzilla account, please send an e-mail to `bug-report-request@r-project.org` from the address that you want to use as your login. In this e-mail, briefly explain why you need an account. A volunteer shall then create a Bugzilla account and add you to R's Bugzilla members.

4. Please ensure whether the bug is already fixed (in the upcoming changes in R) or reported (search for it from those already reported on Bugzilla, either on search existing bug reports, may use the advanced search option here, or show open bugs new-to-old).

4.6.2 Bug in the non R-Core supported packages and/or their documentations

For packages that are not maintained by the R-Core, the bug reports can be submiited at, perhaps, an issues tracker url on GitHub/GitLab/R-Forge. To find if such an issues tracker is available, you can look at the package `DESCRIPTION` file first (e.g. using `packageDescription("package_name")`) to check if a url is provided in the `BugReports` field. If that is not available, then the package maintainer can be contacted (using `maintainer("package_name")`). In R running the function `bug.report(package = "package_name")` shall direct you to either the GitHub issue tracker of the package, or to the bug tracking web page, or towards composing an e-mail to the package maintainer. This function `bug.report` is disabled in RStudio, by default. However, if you use `utils::bug.report(package = "package_name")` then it works on RStudio as well. Please ensure that the bug is not already reported or fixed before reporting it in any of the ways suggested above.

4.7 Good practices in reporting bugs / Expectations of a good bug report

If you follow the practices given below, you will come up with a good bug report which might make it easier for the maintainer(s) to fix the bug.

1. Include a minimal reproducible example of the bug in your report. The maintainer should be able to quickly reproduce the bug on using the minimal example that you provide. Here is a community wiki post on how to make a minimal reproducible example.

2. Mention the software architecture on which the bug occurred.
3. Use inbuilt data sets as far as possible.

In addition to the above, here are the bug writing guidelines on Bugzilla. The bug reporting documentation in R also the discusses practices to write a good bug report.

Once you have successfully reported a bug, you will likely receive an update each time an action is taken on the bug. On Bugzilla, the report may be given one of the following status: New, Assigned, Confirmed, Reopened, Unconfirmed.

4.8 Disagreement with a resolution on the bug tracker

As humans, there might be differences of opinions from time to time. What needs to be considered here is that, being respectful of the fact that care, thought, and volunteer time has gone into the resolution of the issue or bug.

If you take some time, then on reflection, the resolution steps may seem more reasonable than you initially thought. If you still feel that the resolution is incorrect, then raise a thoughtful question to the person who resolved it. If the issue was carefully thought about in the first place then is less likely to win any conversion of thought.

As a reminder, issues closed by a core developer on Bugzilla have already been carefully considered. Please do not reopen a closed issue. Although one can comment on a closed issue, if necessary. Every comment on an issue generates an email to every R-core member (unless they have the notifications disabled). So it would be best to be considerate while commenting on issues, especially in case of the closed issues or when you are commenting in pure agreement without adding anything beyond that to a discussion (the +1 type posts which are perfectly acceptable in other contexts).

4.9 Examples of Bug reports submitted on Bugzilla or R-devel mailing list

If you like to see how bugs are reported on Bugzilla, here are some examples:

1. A bug report with a reproducible example, a patch, and a review.
2. A bug report submitted by Kara Woo which was promptly fixed via the R-devel mailing list. (More information about the R-devel mailing list can be found here).

3. A substring bug reported by Toby Dylan Hocking and fixed by Tomas Kalibera, Feb 2019 via the R-devel mailing list.

4. A gregexpr bug report and patch submitted by Toby Dylan Hocking and merged by Tomas Kalibera, Feb 2019 via the R-devel mailing list.

4.10 See also

1. Reporting a bug

2. R FAQ on bugs

3. Bugzilla guidelines of reporting a bug

Chapter 5

Reviewing Bugs

5.1 How you can help to review bug reports?

After understanding where bugs are reported in R (Bugzilla) or in other projects (GitHub/GitLab/R-Forge), a great way to contribute is reviewing bug reports.

Around the clock, new bug reports are being submitted on Bugzilla or the bug trackers (for instance, GitHub issues) of R packages and existing bug reports are being updated. Every bug report needs to be reviewed to make sure various things are in proper order. You can help with this process of reviewing bugs.

5.1.1 Preparing to review bug reports

If you want to review bug reports on Bugzilla, you are required to have a Bugzilla account. To get a Bugzilla account send an e-mail to `bug-report-request@r-project.org` from the address you want to use as your Bugzilla login. Briefly explain why you want a Bugzilla account and a volunteer will add you to R's Bugzilla members. More details on how you can review a bug report are available on this blog

5.2 Classifying bug reports

A good bug report is the one which:

1. Explains clearly how to reproduce the bug.
2. Includes the version of R, the machine architecture, and the operating system platform on which the bug occurred.

Relevant details should be a part of a good bug report. You can help with the following tasks once you have some R programming experience:

1. Reproducing the bug: If you see a bug report which does not clearly explain how to reproduce it, you can try reproducing the bug and eventually make things easier for the core developer(s) and/or package maintainer(s).

2. Checking different binary builds: Check whether the bug occurs on a different binary build of R. It is helpful to know whether the bug is affecting: `r-patched`, `r-devel`, or `r-release` binary builds of R.

3. Writing a unit test: If the bug report lacks a unit test that should be a part of R's test suite, then you can help with providing it.

These helpful tasks allow the Core developers and/ or maintainers to classify a bug report properly, so that the bug can be handled in a timely fashion.

5.3 How to find a bug report or an issue to review?

1. You may search old bug reports or issues that could be closed. Old bug reports may no longer be valid or may include a patch that is ready to be committed, but no one has had the time to review and commit.

2. You might also want to search for issues in topics in which you have a working knowledge. When on Bugzilla you can use the advanced search to find specific topics. Bug reports are by default public on Bugzilla (unless the defaults are changed to avoid security vulnerability).

5.4 Example of a bug review submitted on Bugzilla

If you would like to see how bugs are reviewed on Bugzilla, here is an example where an old bug report is being reviewed. It is tested to see if it was still an issue and a few ways are proposed to resolve the issue.

Note:

There is a `#bug-reporting` channel on the R Contributors slack where you can share your bug report(s) for review/feedback before submitting to Bugzilla. This can help with checking that it really is a bug, that you have included the important information and excluded redundant information.

5.5 See also

1. Reviewing bug reports: Blog

Chapter 6

Finding the Source

This chapter discusses how you can have an overview of the R codebase. For instance, where to find the implementation of a base function written in R and where to find a primitive implementation written in C. You may want to find the source code of a function just out of curiosity or maybe to gain more insight into what a particular function is actually doing. Whatever be the case, reading the source code will help you to learn a lot about any function.

6.1 Finding R source code

1. Find the R function with the code of interest. You will always be able to print the top-level function (or use `View(function_name)` in RStudio). Looking at the code for the body of this function will reveal what you need to do next:

- Can already see code of interest: stop here or skip to step 3 to find the corresponding file in the R sources.
- Code of interest is in nested R function: go to step 2.
- Top-level function is an S3 generic, identified by a call to `UseMethod()`. Use `methods(function_name)` to see available methods, then go to step 2.
- Code of interest is in compiled code, identified by a call to `.C()`, `.Call()`, `.Fortran()`, and `.External()`, or `.Internal()` and `.Primitive()`: go to section on compiled code.

2. Nested functions or S3 methods may not be exported by the package they are in. If this is the case, the simplest way to view the code is to use

`getAnywhere()` or `getS3method()`. Now you can keep looking at nested R functions till you find the code of interest or hit a call to compiled code.

3. Find an R function in the R sources. Two options here:

- Search on the internet: For R Core packages, search on the GitHub mirror (https://github.com/r-devel/r-svn); for recommended packages, use the CRAN mirror (https://github.com/cran) - this will link to the source on GitHub if available, e.g. https://github.com/cran/survival. Note that GitHub search ignores wildcard characters

```
. , : ; / \ ` ' " = * ! ? # $ & + ^ | ~ < > ( ) { } [ ]
```

but this does not include `-` so you can search for a function or S3 method as follows:

```
"body <- function" extension:R
"quantile.ecdf <- function" extension:R
```

- Search in the R sources using grep: The Getting Started chapter discusses how to download the R sources directly or from the svn repository. Now if the sources are in `~/R-devel`, you can search as follows:

```
grep -R "body <- function" ~/R-devel/src
grep -R "quantile <- function" ~/R-devel/src/library
```

Note: The above procedure does not cover S4, R6 or RC generics or methods. Refer accessing R source for further details.

6.2 Finding C source code

1. If `.Internal()` or `.Primitive()`, find entry point in `names.c` as described in the Jenny Bryan's post of accessing R source. For all other calls to compiled code, you can find the entry point from within R. For instance, the body of complete.cases() is

```
.External(C_compcases, ...)
```

`C_compcases` inherits from class "NativeSymbolInfo" and we can extract the name of the entry point via

```
stats:::C_compcases$name
```

We know that it is in the stats package as we see that when we print complete.cases or look at the help file. This shows us that the entry point is simply "compcases" and in fact that is the general convention in R code, that you simply remove the `C_` prefix (sometimes `.F_` for Fortran code) in the name of the object passed to the first argument of the call.

2. Once you have the entry point, search as for R code. In the case of searching on GitHub, restrict the search to files with the relevant extension

```
compcases extension:c
lowesw extension:f
```

similarly for grep

```
grep -R --include=*.c "compcases" ~/R-devel/src/library/
```

Note:

1. Many editors (like RStudio, ESS) support ctags for code browsing, making it easy to jump to definitions of functions. `R CMD rtags` can generate ctags for any R code (Credit: Deepayan Sarkar).
2. A more sophisticated system is called GNU GLOBAL, which also supports finding all references (calls) to a function.
3. GitHub has a code navigation feature via the library tree-sitter. Unfortunately, it does not have R support yet. An R driver for tree-sitter made by Jim Hester is available.

6.3 See also

Read the R source blogpost.

Chapter 7

Lifecycle of a Patch

7.1 Introduction

R uses a workflow based on patches. A patch is the set of differences (additions and deletions) between two versions of code. So you can create a patch defining a bug fix or a proposed update to the R codebase and submit it through your official Bugzilla account to the core developer(s). Be clear in your communication as it is the key to contributing to any project, especially an open source project like R.

7.2 When do you submit a patch?

There might be a situation where you come across a bug in R, which you may have an idea of how to fix. This can turn out to be an opportunity for you to submit a patch. By submitting a patch or a bug fix, you are helping to reduce the workload on the R developers in addition to yourself being a contributor to R!

When you submit a patch, you are helping the developer(s) and maintainer(s) so that they do not have to write the entire code from scratch. Instead, they can test and tweak your patch, if necessary.

7.3 What tools are required to submit a patch?

To submit a patch, you need:

1. SVN installed on your machine.

2. The latest developer version of R.

7.4 How to prepare a patch?

Refer to the guidelines given here for the process to prepare a patch against R-devel.

7.5 Making good patches

When creating a patch for submission, there are several things that you can do to help ensure that your patch is accepted:

1. Make sure to follow R's coding standards (R is a GNU project and there are GNU coding standards). The coding style of the patch you submit should largely match with the codebase it is being applied to. If your patch has one or two minor discrepancies, then those may be fixed by the core developer who will eventually test your patch. However, if there are systematic deviations from the style guides your patch will be put on hold until you fix the formatting issues. There is no comprehensive official R style manual, however some nearly universal standards are summarised in this article.

2. Be aware of backwards-compatibility considerations. While the core developer who eventually handles your patch will make the final call on whether something is acceptable, thinking about backwards-compatibility early will help prevent having your patch rejected on these grounds. Put yourself in the shoes of someone whose code will be broken by the change(s) introduced by the patch. It is quite likely that any change made will break someone's code, so you need to have a good reason to make a change as you will be forcing someone to update their code. This obviously does not apply to new functions or new arguments. New arguments should be optional and have default values which maintain the existing behaviour. If in doubt, discuss the issue with experienced developers.

3. Make sure you have proper tests to verify that your patch works as expected. Patches may not be accepted without the proper tests.

4. Make sure the entire test suite runs without failure because of your changes. It is not sufficient to only run whichever test seems impacted by your changes, because there might be interactions unknown to you between your changes and some other part of the interpreter.

5. Proper documentation additions/changes should be included.

6. Each bugfix should ideally be addressed by a single patch. In particular, do not fix more than one issue in the same patch (except, if one code change fixes all of them) and do not do cosmetic changes to unrelated code in the same patch as some bugfix.

7.6 Submitting your patch for review

1. Patch in response to a pre-existing issue or bug report: In this case, you should attach the patch to the existing issue or bug report on Bugzilla with a brief comment.

2. Patch in response to an unreported issue or bug report: Assuming you already performed a search on Bugzilla for a pre-existing issue or bug and did not find the issue or bug reported, you need to create a new bug report and include your patch with it. Please fill in as much relevant detail as possible to prevent reviewers from having to delay reviewing your patch because of lack of information. Include (mostly as the first sentence), a to-the-point explanation of what the purpose of the patch is. This sentence should not be in the descriptive form, rather an imperative form will be more suitable here. If this is not enough detail for a patch, a new paragraphs(s) can be added to explain in proper depth what has happened. The details should be good enough that a core developer reading it understands the justification for the change.

7.7 Getting your patch reviewed

To begin with, please be patient. There are many more people submitting patches than there are people capable of reviewing your patches. Getting your patch reviewed requires a reviewer to have the spare time and motivation to look at your patch. We cannot force anyone to review patches and no one is employed to look at patches.

There is a `#patches-for-review` channel on the R Contributors slack where you can share your patch(es) for review/feedback before submitting to R-Core/Bugzilla. This can help with checking that you have included the important information and excluded redundant information.

If your patch has not received any notice from reviewers (i.e., no comment made) after one month, comment/message on the `#patches-for-review` channel to remind the members that the patch needs a review.

When someone does manage to find the time to look at your patch they will most likely make comments about how it can be improved. It is then expected that you update your patch to address these comments, and the review process will thus iterate until a satisfactory solution has emerged.

7.7.1 How to review a patch?

One of the bottlenecks in the R development process is the lack of code reviews. If you browse Bugzilla, you will see that numerous issues have a fix, but cannot be merged into the main source code repository, because no one has reviewed the proposed solution. Reviewing a patch can be just as informative as providing a patch and it will allow you to give constructive comments on another developer's work. This guide provides a checklist for submitting a patch review. It is a common misconception that in order to be useful, a patch review has to be perfect. This is not the case at all. It is helpful to just test the patch and/or play around with the code and leave comments in the patch or on Bugzilla.

If a bug report or an issue has a patch attached that has not been reviewed, you can help by making sure that the patch:

- follows the style guides;
- is a good solution to the problem it is trying to solve;
- includes proper tests; and
- includes proper documentation changes.

Also refer to Making good patches for more ideas. Doing all of this allows the core developer(s) and/ or maintainer(s) to more quickly look for subtle issues that only people with extensive experience working on R's codebase will notice.

7.8 Leaving a patch review on Bugzilla

When you review a patch, you should provide additional details and context of your review process and leave comments. For example:

1. If you tested the patch, report the result and the system and version tested on, such as 'Windows 10', 'Ubuntu 16.4', or 'Mac High Sierra'.
2. If you request changes, try to suggest how or attach an updated patch.
3. Comment on what is 'good' about the patch, not just the 'bad'. Doing so will make it easier for the patch author to find the good in your comments.

7.9 Dismissing review from another core developer

A core developer can dismiss another core developer's review if they confirmed that the requested changes have been made. When a core developer has assigned

the patch to themselves, then it is a sign that they are actively looking after the patch, and their review should not be dismissed.

7.10 Acceptance or rejection of your patch

Once your patch has reached an acceptable state, it will either be applied or rejected. If it is rejected, please do not take it personally. Your work is still appreciated regardless of whether your patch is applied. Balancing what does and does not go into R is tricky and everyone's contributions cannot always be accepted.

But if your patch is accepted and applied it will then go on to be released with the next patched release and eventually the next major release of R. It may also be backported to older versions of R as a bugfix if the core developer doing the patch acceptance believes it is warranted.

It may take longer before your patch is accepted and applied or rejected, sometimes even months or years. Nonetheless, it is appreciated that you submitted a patch.

7.11 Examples of patch reports on Bugzilla

7.12 Examples of reviewing a patch

7.13 See also

1. Submitting patches

Chapter 8

Documenting R

The R language has a substantial body of documentation, much of which is contributed by various authors. The help files for R functions are written in 'R documentation' (.Rd) file format. It is a simple markup language with close resemblance to LaTeX. The .Rd file format can be further processed into a variety of formats, including LaTeX, HTML, and plain text.

This chapter describes the styleguide for R's documentation, how to document for R, report and review bugs (and suggest corrections for them) in the existing documentation. If you are interested in contributing to R's documentation, your contributions are more than welcome.

8.1 Introduction

R's documentation has long been considered to be good for a free programming language. The core developers have been committed to providing good documentation on the language and its packages. The continuing involvement of the user community in providing assistance for creating and maintaining documentation has also supported a lot.

The involvement of the community takes many forms, from authoring to bug reports to raising an issue when the documentation could be more complete or easier to use.

This chapter is aimed at authors and potential authors of documentation for R. More specifically, it is for people contributing to the standard documentation and developing additional documents using the same tools as the standard documents. Any time you feel that you can clarify existing documentation or provide documentation that is missing, your contribution will be welcome and highly appreciated. If you find it difficult to deal with the markup language,

you can ask for help for that part too. Please do not let the material in this chapter stand between the documentation and your desire to help out.

8.2 Guidelines for writing the system help files (in .Rd format)

Following are the suggested guidelines while creating the system help files in the .Rd format. These are intended for writing the core documentations but may also be useful for package writers. (share documentation of a simple base function here?)

There are three main parts of an .Rd file:

1. **Header**: This part is for the basic information of the document/file. For instance, the name of the file, the topics documented, a title, a short textual description, and R usage information for the objects documented.
2. **Body**: This part includes further information on the function's arguments and return value.
3. **Footer**: This part is optional. Usually the keyword information is included here.

All the above information is included in a .Rd file within a series of sections with standard names (user-defined sections are also allowed). These sections are discussed below:

1. `\title` section:
    - Capitalize each word.
    - Do not end in a period.
    - Avoid use of markup language (because markup language need not be suitable for various hypertext search systems).
2. `\usage` and `\examples` sections:
    - Line length of 65 characters is advised.
    - Use `TRUE` instead of `T` and `FALSE` instead of `F`.
    - Add spaces around binary operators.
    - Add spaces after commas in the argument lists.
    - Use `<-` rather than `=` for assignments.
    - Add spaces around the `<-` operator.
    - Do not use tabs to indent (as these do not render correctly on all possible pagers).
    - Use 4 spaces to indent the (example) code.

- Make sure the examples are directly executable.
- The examples should be system-independent.
- The examples should not require special facilities (for instance, Internet access or write permission to specific directories).
- Examples should also not take longer than necessary to run, as they are run when checking a build of R.

3. `\synopsis` section: The `\usage` section can have the function definition, or the actual definition can be included in the `\synopsis` section.

4. `\source` and `\references` sections:

   - Author(s) names should be written in the form `Author, A. B.`.
   - Author(s) names should be separated by a comma or `and` (but not both).
   - Separate paragraphs (separated by a blank line) should be used for each reference.
   - Give a date immediately after the author(s) names.
   - Do not put a period after the date.
   - Titles of books and journals (not articles) should be enclosed in `\emph{...}`.
   - Volume numbers for journals are to be enclosed in `\bold{...}` and followed by a comma.
   - Use `--` for page ranges.
   - For giving an address for a publisher use the format `New York: Springer-Verlag`.

More guidelines for writing .Rd files can be found here.

The language used in the documentations should follow these basic rules:

1. Affirmative tone should be used to describe what the function does and how to use it effectively. Rather than creating worry in the mind of a reader, it should establish confident knowledge about the effective use of the particular function/feature.

2. More documentation is not necessarily better documentation. Long descriptions full of corner cases and caveats can create the impression that a function is more complex or harder to use than it actually is. Be succinct but exhaustive.

3. Short code examples can help in understanding better. Readers can often grasp a simple example more quickly than they can digest a formal description. Usually people learn faster with concrete, motivating examples that match the context of a typical use case.

4. Giving a code equivalent (or approximate equivalent) can be a useful addition to the description provided. You should carefully weigh whether the code equivalent adds value to the document.

5. The tone of the documentation needs to be respectful of the reader's background. Lay out the relevant information, show motivating use cases, provide glossary links, and do your best to connect-the-dots. The documentation is meant for newcomers, many of whom will be using it to evaluate the R language as a whole. The experience needs to be positive and not leave the reader with worries that something bad will happen if they make a mistake.

Extensive details of writing R documentation files can be found here.

8.3 Helping with documentation

Maintaining the accuracy of R's documentations and keeping a high level of quality takes a lot of effort. Community members, like you, help with writing, editing, and updating content, and these contributions are appreciated and welcomed.

Looking at pre-existing documentation source files can be very helpful when getting started.

If you look at documentation issues on Bugzilla, you will find various documentation problems that may need work. Issues vary from typos to unclear documentation and items lacking documentation.

If you see a documentation issue that you would like to tackle, you can leave a comment on the issue saying you are going to try to solve the issue and mention roughly how long you think you will take to do so (this allows others to take on the issue if you happen to forget or lose interest).

8.4 Proofreading

While an issue filed on Bugzilla means there is a known issue somewhere, that does not mean there are not other issues lurking about in the documentation. Proofreading a part of the documentation can often uncover problems.

If you decide to proofread, read a section of the documentation from start to finish, filing issues in Bugzilla for each major type of problem you find. It is best to avoid filing a single issue for an entire section containing multiple problems; instead, file several issues so that it is easier to break the work up for multiple people and more efficient review.

8.5 Helping with the Developer's Guide

The Developer's Guide (what you are reading now) uses the same process as the main R documentation, except for some small differences. The source lives in a GitHub repository and bug reports should be submitted to the devguide GitHub tracker.

Our dev guide workflow uses continuous integration and deployment so changes to the dev guide are normally published when the pull request is merged. How to contribute to this guide from the introduction.

8.6 Instructions for reporting the CRAN policy bugs – discussion in slack (random channel)

Note:

There is a `#core-documentation` channel on the R Contributors slack where you can discuss about the patches for improvements to R's documentation.

8.7 See also

1. Writing R documentation files

Chapter 9

Translations

This chapter covers internationalization in R, i.e., the display of messages in languages other than English. All output in R (such as messages emitted by `stop()`, `warning()`, or `message()`) is eligible for translation, as are menu labels in the GUI. Depending on the version of R that you are using, some of the languages might already be available while others may need work. R leverages the `gettext` program to handle the conversion from English to arbitrary target languages.

Having messages available in other languages can be an important bridge for R learners not confident in English – rather than learning two things at once (coding in R and processing diagnostic information in English), they can focus on coding while getting more natural errors/warnings in their native tongue.

The `gettext` manual is a more canonical reference for a deep understanding of how `gettext` works. This chapter will just give a broad overview, with particular focus on how things work for R, with the goal of making it as low-friction as possible for developers and users to contribute new/updated translations.

9.1 How translations work

Each of the default packages distributed with R (i.e., those found in `./src/library` such as `base`, `utils`, and `stats` and which have priority base) contains a `po` directory that is the central location for cataloguing/translating each package's messages.

9.1.1 `.pot` files

The `.pot` file is a snapshot of the messages available in a given **domain**. A domain in R typically identifies a source package and a source language (either

R or C/C++). For example, the file `R-stats.pot` (found in the R sources in `./src/library/stats/po`) is a catalogue of all messages produced by R code in the `base` package, while `stats.pot` is a catalogue of all messages produced by C code in the `stats` package.

There are two exceptions to the basic pattern described above. The first is the domain for messages produced by the C code which is the fundamental backing of R itself (especially, but not exclusively, the C code under ./src/main). The associated `.pot` file is `R.pot` and it is found in `./src/library/base/po`. `R-base.pot` is a normal `.pot` file because base has a normal `R` directory.

The second is the domain for the Windows R GUI, i.e., the text in the menus and elsewhere in the R GUI program available for running R on Windows. These messages are stored in the `RGui.pot` domain, also in the `po` directory for `base`, and are most commonly derived from C code found in `./src/gnuwin32`. One reason to keep this domain separate is that it is only relevant to one platform (Windows). In particular, Windows has historically different character encodings, so that it made more sense for Windows developers to produce translations specifically for Windows, since it is non-trivial for non-Windows users to test their translations for the Windows GUI.

9.1.1.1 Generating `.pot` files

For outside contributors, there's no need to update .pot files – translators will typically take the R `.pot` files as given and generate `.po` files. These will be sent along to a language-specific translation maintainer, who then compiles them to send to the R Core developer responsible for translations, who finally applies them as a patch.

To emphasize, this section is almost always not needed for contributing translations – it is here for completeness and edification.

9.1.2 `.po` files

.po files are the most important artifacts for translators. They provide the (human-readable!) mapping between the messages as they appear in the source code and how the messages will appear to users in translated locales.

9.1.2.1 Singular messages

Most messages appear as `msgid`/`msgstr` pairs. The former gives the message as it appears in the code, while the latter shows how it should appear in translation. For example, here is an error in German (locale: `de`) informing the user that their input must be of class `POSIXt`

```
msgid "'to' must be a \"POSIXt\" object"
msgstr "'to' muss ein \"POSIXt\" Objekt sein"
```

See this in context in the `R-de.po` source file.

The same message can also be found in `R-it.po` giving the translation to Italian:

```
msgid "'to' must be a \"POSIXt\" object"
msgstr "'to' dev'essere un oggetto \"POSIXt\""
```

9.1.2.2 Plural messages

Some messages will have different translations depending on some input determined at run time (e.g., the `length()` of an input object or the `nrow()` of a `data.frame`). This presents a challenge for translation, because different languages have different rules for how to pluralize different ordinal numbers[^See the relevant section of the `gettext` manual]. For example, English typically adds `s` to any quantity of items besides 1 (1 dog, 2 dog`s`, 100 dog`s`, even 0 dog`s`). Chinese typically does not alter the word itself in similar situations (, , ,); Arabic has *six* different ways to pluralize a quantity.

In `.po` files, this shows up in the form of `msgid_plural` entries, followed by several ordered `msgstr` entries. Here's an example from `R-de.po`:

```
msgid "Warning message:\n"
msgid_plural "Warning messages:\n"
msgstr[0] "Warnmeldung:\n"
msgstr[1] "Warnmeldungen:\n"
```

The two entries in English correspond to the singular and plural messages; the two entries in German correspond similarly, because pluralization rules in German are similar to those in English. The situation in Lithuanian (`R-lt.po`) is more divergent:

```
msgid "Warning message:\n"
msgid_plural "Warning messages:\n"
msgstr[0] "Įspėjantis pranešimas:\n"
msgstr[1] "Įspėjantys pranešimai:\n"
msgstr[2] "Įspėjančių pranešimų:\n"
```

This corresponds to the 3 different ways to pluralize words in Polish.

What do `0`, `1`, and `2` correspond to, exactly? Ideally, this will be clear to native speakers of the language, but for clarity, it is the solution to a small arithmetic problem that can be found in the language's metadata entry. Look for the `Plural-Forms` entry in the metadata at the top of the `.po` file; here it is for Lithuanian:

```
"Plural-Forms: nplurals=3; plural=(n%10==1 && n%100!=11 ? 0 : n%10>=2 && (n"
"%100<10 || n%100>=20) ? 1 : 2);\n"
```

`nplurals` tells us how many entries correspond to each `msgid_plural` for this language. `plural` tells us, for the quantity `n`, which entry to use. The arithmetic is C code; most important if you really want to parse this and are only familiar with R code is C's ternary operator: `test ? valueIfTrue : valueIfFalse` is a handy way to write R's `if (test) valueIfTrue else valueIfFalse`.

Parsing, we get the following associations:

- the `0` entry corresponds to when a number equals 1 modulo 10 (i.e., 1, 11, 21, 31, …) *except* numbers equaling 11 modulo 100 (i.e., 11, 111, 211, 311, …). Combining, that's 1, 21, 31, …, 91, 101, 121, 131, …, 191, …
- the `1` entry corresponds to numbers at least 2 modulo 10 (2, 3, …, 8, 9, 12, 13, 14, …) and *either* below 10 modulo 100 (0, 1, …, 9, 100, 101, …, 109, …) *or* exceeding 20 modulo 100 (21, 22, …, 99). Combining, that's 2, 3, …, 9, 22, 23, …, 29, 32, 33, … 39, …, 102, 103, …, 109, 122, 123, …
- The `2` entry corresponds to all other numbers, i.e. 0, 10, 11, 12, …, 19, 20, 30, …, 90, 100, 110, 111, 112, …

9.1.3 `.mo` files

`.po` files are plain text, but while helpful for human readers, this is inefficient for consumption by computers. The .mo format is a "compiled" version of the .po file optimized for retrieving messages when R is running.

In R-devel, the conversion from .po to .mo is done by R Core – you don't need to compile these files yourself. They are stored in the R sources at `./src/library/translations/inst` in various language-specific subdirectories.

9.2 How to contribute new translations

9.3 Current status of translations in R

https://contributor.r-project.org/translations/

9.4 Helpful references

- Statistical terms glossary

Chapter 10

Testing Pre-release R Versions

This chapter is inspired from the blog on testing R before release and discusses how you can help with testing of pre-release versions of R.

10.1 Where to test?

Whenever possible use a fresh package library for testing, even better would be to use virtual machines for the testing. This would ensure that you do not damage your existing R installation.

10.1.1 Virtual machine

A free Windows 10 virtual machine is provided by Microsoft (with a 90-day limit) for building, testing, and checking R packages and R itself. Package maintainers who work on Linux and MacOS can use it to test their packages on Windows. Read the instructions on how to automatically set up the machine to check R packages. Tomas Kalibera describes the details of using virtual machine in the blog Virtual Windows machine for checking R packages.

10.2 What can you test?

You can test:

- Your own programs.

- Your own workflows.
- Your special ways of installing or setting up R.
- Things that interact with external libraries.
- Interactive R packages.

Details of performing testing on various operating systems:

- Windows
- macOS
- Linux
- Solaris

10.3 Writing tests for R

Writing tests for R is much like writing tests for your own code. Tests need to be thorough, fast, isolated, consistently repeatable, and as simple as possible.

When you are adding tests to an existing test file, it is also recommended that you study the other tests in that file; it will teach you which precautions you have to take to make your tests robust and portable. We try to have tests both for normal behaviour and for error conditions. Tests live in the `tests` directory.

10.4 Benchmarks

Benchmarking is useful to test that a change does not degrade performance.

Chapter 11

R Core Developers

This page lists the former and current members of the R Core team who have write access to the R source.

- Brian Ripley (present)
- Deepayan Sarkar (present)
- Douglas Bates (present)
- Duncan Murdoch (up to September 2017)
- Duncan Temple Lang (present)
- Friedrich Leisch (present)
- Guido Masarotto (up to June 2003)
- Heiner Schwarte (up to October 1999)
- John Chambers (present)
- Kurt Hornik (present)
- Luke Tierney (present)
- Martin Maechler (present)
- Sebastian Meyer (present)
- Martin Morgan (up to June 2021)
- Martyn Plummer (present)
- Michael Lawrence (present)
- Paul Murrell (present)
- Peter Dalgaard (present)
- Robert Gentleman (present)
- Ross Ihaka (present)
- Seth Falcon (up to August 2015)
- Simon Urbanek (present)
- Stefano lacus (up to July 2014)
- Thomas Lumley (present)
- Tomas Kalibera (present)
- Uwe Ligges (present)

View the affiliations of R Core members. We've left it up to the individual core developers to list areas of expertise (or things they are willing to maintain) if they wish.

The Contributors page on the R Project website also lists contributors, outside the R Core team, who provided invaluable help by donating code, bug fixes, and documentation.

Chapter 12

Where to Get Help

If you are working on R it is possible that you will come across an issue where you would need some assistance to solve it. If you require help, there are options available to seek assistance or get some feedback which are discussed in this chapter. If the question involves process or tool usage then please check the rest of this guide first as it should answer your question. Please make sure to search the documentation and resources to see if your question has already been addressed. If not, then ask for assistance in the appropriate forum. Many developers are volunteers and please be polite, patient, and thoughtful when requesting for feedback or help.

12.1 Slack

You can discuss issues related to the development of R and learn about the process of contributing to R on the R Contributors slack. There are a number of experienced developers on this slack who can answer questions and/or provide feedback. The following channels are available on the R-devel slack for help and feedback with specific areas:

- #bugreports-for-review: Share bug reports for review/feedback before submitting to Bugzilla.
- #core-dev-help: Getting help on anything related to R Core contribution.
- #core-documentation: Discuss patches/improvements to R's documentation.
- #core-translation: Discuss translating R messages, warnings, and errors into non-English languages.

- #patches-for-review: Share patches for peer review before submitting to R Core.

Note: You may not be able to access the history of these channels, so it cannot be used as a knowledge base of sorts.

12.2 Mailing lists

There are quite a few mailing lists for getting help with R:

- R-devel:
  - Questions and discussion about development *of* R vs. *with* R.
  - Getting help with technical programming issues, e.g. interfacing R with C/C++.
  - Proposals of new functionality for R.
  - Pre-testing of new versions of R.
  - Enhancements and patches to the R source code and the R documentation.
  - Posting examples and benchmarks.
- R-help:
  - Discussions about problems and solutions using R.
- R-package-devel:
  - Getting help about package development in R.
  - Learning about the package development process.
  - Discussing problems developing a package (or problem in passing the R CMD check).

Please avoid cross-posting to both the R-package-devel and the R-devel mailing lists.

12.3 File a bug

If you strongly suspect you have come across a bug (be it in the build process, or in other areas), then report it on Bugzilla.

Chapter 13

News and Announcements

Here are some resources that can be useful to keep up with the developments in R:

13.1 Blogs

The R project maintains a list of public blogs mainly written by the R Core Development Team. Daily news about R-devel can be found here.

13.2 Conferences

Updates about conferences actively supported or endorsed by The R Foundation can be found here. These conferences are organised by members from the R community.

13.3 Journal

The R Journal is an open access and refereed journal featuring short to medium length articles that should be of interest to users or developers of R. It also has a news section where information on, changes in R (new features of the latest release), changes on CRAN (new add-on packages, manuals, binary contributions, mirrors, etc.), upcoming conferences, and conference reports is provided.

13.4 Mailing lists

- R-announce: A moderated mailing list used for announcements by the R Core Development Team. Major announcements about the development of R and the availability of new code are made here.
- R-packages: A moderated mailing list for announcements about contributed R packages (typically on CRAN) and similar R project extensions.

13.5 Twitter

Follow @R_dev_news on Twitter for the R development updates and blog post announcements.

Chapter 14

Developer Tools

This chapter lists resources and tools which R developers may use. Here we will go over some commonly used tools that are relevant to R's workflow. As there are several ways to accomplish these tasks, this chapter reflects methods suitable for new contributors. Experienced contributors may desire a different approach.

14.1 Subversion (svn) client

Subversion (svn) is a version control system that tracks any changes made to files and directories. You can install either the TortoiseSVN (https://tortoisesvn.net/, command line tool, and Windows Explorer integration) or the SlikSVN (https://sliksvn.com/download/, just the command line tool) client. They have Windows installers and can be used from Windows cmd or RStudio terminal.

Some resources for learning subversion commands:

1. Apache Subversion quick start guide
2. TortoiseSVN commands
3. SlikSVN basics
4. Subversion book

14.2 Globally search for a regular expression and print matching lines (grep)

grep is a command line utility for searching plain text data sets for lines that match a regular expression. Refer the grep manual for more commands.

14.3 Git

Git is also a version control system for tracking changes in any files and directories. View git documentation for learning git commands.

14.4 GitHub

Some resources that are useful while using GitHub are:

1. Creating a pull request
2. Opening an issue from code
3. Resolving a merge conflict on GitHub