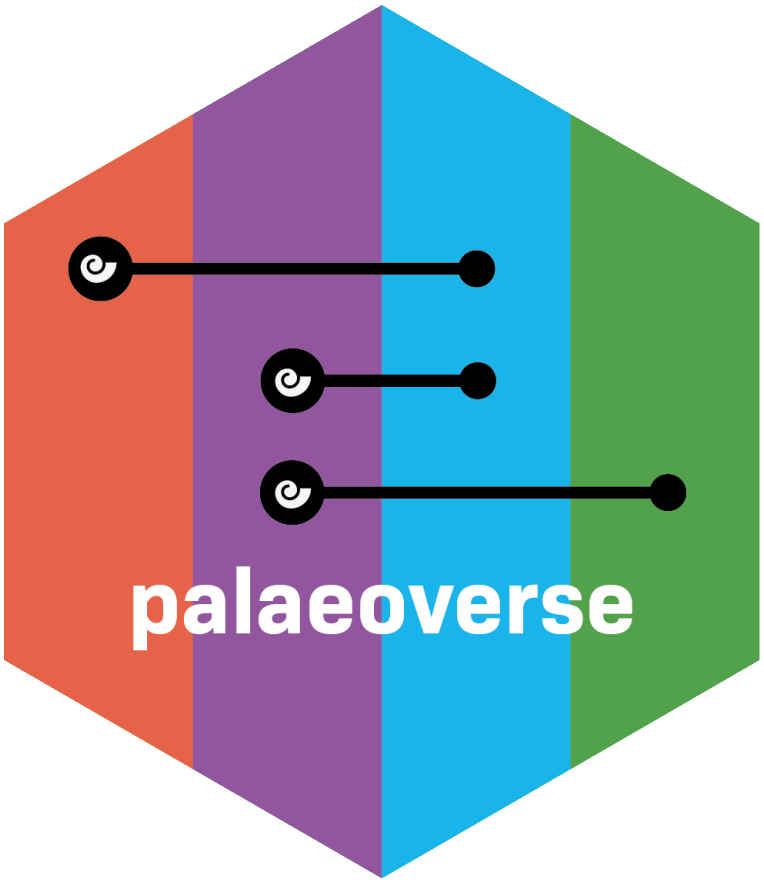
Introduction to palaeoverse: structure and standards



Authors: Lewis A. Jones, William Gearty, Kilian Eichenseer, and Bethany Allen

Reviewed by: Christopher D. Dean and Sofía Galván

Last updated: 22 July, 2022

1 Introduction

The `palaeoverse` *R* package is a community-driven software library providing generic tools for palaeobiological analysis. The core principles of palaeoverse are to: (1) streamline analyses, (2) enhance code readability, and (3) improve reproducibility of results.

This document describes the essential structure and conventions of the `palaeoverse` R package. Naturally, there are always disagreements regarding best practices and conventions, and `palaeoverse` is no exception. Despite this, all the essentials in `palaeoverse` are encouraged to make the lives of both the developer, and the user, easier. It is worth noting that the core structure and conventions adopted in `palaeoverse` are heavily influenced by Hadley Wickham and Jenny Bryan's R Packages and the tidyverse style guide, which is currently also Google's guide.

"Good coding style is like correct punctuation: you can manage without it, butitsure-makesthingseasiertoread." – tidyverse style guide

2 Files

File names should always be concise, meaningful and end in `'.R'`. Avoid using special characters whenever possible (i.e. stick to letters and numbers), and use `'_'` or `'-'` instead of spaces in file names. The use of lowercase is also strongly encouraged, and **never** have file names that only differ in capitalization. **Note:** the preferred separator is `'_'` to consist with the 'lowercase snake_case' convention for developing functions (more on that later!).

```
# Nein!
Sup3r Aw3sum Functi0n.r

# Besser
super_awesome_function.R

# Das ist gut!
time_bins.R
```

3 R style guide

We recommend using the following conventions to make your code easier for us (and the community) to understand and use:

Assigning: use arrows (<-) rather than equals sign to assign.

Comments: please explain your processes comprehensively using comments (#), as this will help us to understand your intentions and to proof-read the code.

Packages: please use as few external packages (i.e. dependencies) as possible, as the package is more likely to break when alterations are made to these; if your code uses packages, load them all at the start of the script. This is more transparent for the user than having various packages sprinkled throughout the code.

Progress: if your code may take a long time to run (more than a minute), include a progress bar such as `'txtProgressBar()'` in your function.

Sections: use lines of '-' or '=' to indicate section breaks. It is hard to provide exact details on how long a section break should be. However, we tend to think of them as paragraphs in our code. **Tip:** In RStudio, you can use `'# SECTION NAME ###'` to make a section which is interpreted by the outline feature. You can also make subsections by adding more `'#"` before the section name.

Spacing: add spaces to your code to make it more human-readable.

Objects: use object names that are short but relevant to their contents, ideally in 'lowercase snake_case'.

Wrapping: break long commands into multiple lines, conventionally no line should be longer than ~75 characters.

Language: use plain English (British spelling preferred, but not required) for your code and documentation.

```
# Nope!
x = 1
y<-x+1

# Awesome!
x <- 1
y <- x + 1
```

4 Data

Often we will want to include data in palaeoverse. This might come in the form of example datasets for testing functions (e.g. fossil occurrences), reference datasets such as the Geological Timescale 2020, or even data that is fundamental for a function to run. In palaeoverse's structure, we currently recognize four main ways of including data in the package depending on its usage.

- Raw data
- Internal data
- Exported data
- External data (preferred option)

However, for the sake of file size efficiency, please only include data and variables in your files which will be used by your function. **Note:** in order to be released on CRAN, R packages must be less than 5 MB in size. Please only include data that is absolutely vital for your function. The preferred solution to including data into palaeoverse functions is calling the data via an API or download URL (i.e. external data). We have our own data repository for including data, so please get in touch if you want to include data.

4.1 Raw data

Raw data should always be included in `inst/extdata`. If you want to include cleaned data in `data/`, it is generally a good idea to include the code used to process the raw data. If you ever need to reproduce or update your cleaned data, this will save you precious time. The code for processing your data should be included in `data-raw/`. Strictly speaking, raw data does not need to be documented. However, it is a good idea to include the original source and version (including a download date) in your code.

4.2 Internal data

Data you do not wish to directly make available to the user should be saved as `R/sysdata.rda`. This is the best option for pre-computed data tables that are needed for a function to run. Strictly speaking raw data does not need to be documented. However, it is a good idea to document the internal data in the function documentation.

4.3 Exported data

Package data you wish to make available to the user should be stored in `data/`. Each file in this directory should be either a `'.rda'` or `'.RData'`. This file type is fast, small and explicit. The most appropriate way to include exported data is to use `usethis::use_data()`.

When using large datasets, we want to ensure that our files are not bloated and taking up too much space on our users' machines. As such, you may want to experiment with the compression settings in `usethis::use_data()`. Generally, `xz` and `gzip` can create smaller files than the default `bzip2`. You can also implement several 'hacks' to generate smaller files which you may want to consider for large datasets (depending on whether your data is sensitive to such changes). Data with many decimal places consume a lot of memory, consider how many significant figures are relevant for your data, and `round()` accordingly. You can also experiment with your file size by multiplying your data by X (e.g. 1,000) to remove decimal places altogether. **Note:** Remember to undo any transformations when calling or working with the data.

4.4 External data

External data is the preferred approach for contributors to include data into `palaeoverse` and will be required in almost all cases. This is to ensure that the package does not become unnecessary bloated to all users when the data can just be called by a download link. Currently, external data are stored on a private Dropbox account. It is likely that in the future this will be migrated elsewhere but for all intent and purposes, Dropbox does a good job for now. If you wish to include external data, please get in touch with one of the palaeoverse developers, and we can store it on the Dropbox account and provide a static link.

```
#generate temp directory
files <- tempdir()

#download files
download.file(url = "www.myfiles.com", destfile = paste0(files, "/my-download.csv"))

#run some kind function using the download

#REMEMBER: remove downloaded files
unlink(x = paste0(files, "/", list.files(files)))
```

4.5 Data documentation

Objects in `data/` are always exported by default, and should be documented accordingly. In order to properly document data, you must document the name of the dataset and save it in `R/data`. For example, the documentation block used to document GTS2020 is saved as `R/data.R` and is similar to the following (simplified here):

```
#' Geological Time Scale 2020
#'
#' A dataset of the Geological Time Scale 2020. Age data from:
#'  \url{https://stratigraphy.org/timescale/}.
#' Supplementary information is also included in the dataset for
#' plotting functionality (e.g. GTS2020 colour scheme).
#'
#' @format A data frame with 189 rows and 20 variables:
#' \describe{
#'   \item{index}{Index number for the order of all intervals in the dataset}
#'   \item{stage_number}{Index number for stages}
#'   \item{series_number}{Index number for series}
#'   \item{system_number}{Index number for system}
#'   \item{interval_name}{Names of intervals in the dataset}
#'   ...
#' }
#' @section References:
#' Gradstein, F.M., Ogg, J.G., Schmitz, M.D. and Ogg, G.M. eds. (2020).
#' Geologic time scale 2020. Elsevier.
```

```
#' @source Compiled by Lewis A. Jones. See item descriptions for details.
"GTS2020"
```

4.6 What does it mean to document your data?

Documenting your data is to provide a thorough description of the data and any information relevant to understanding it. Good documentation is key to reproducible science, and will also help us to ensure that we acknowledge all data collators who have provided data for the `palaeoverse` package. When providing us with datasets, please give the following information:

Author(s): Who collected the data, and prepared its current format? Please provide citations if relevant. Have the authors given permission for the dataset to be included in the `palaeoverse` package?

Description: Brief description of the dataset.

Provenance: When and how was the data collected? When was the dataset finalised in its current form?

Size: Please state the full, uncompressed size of the file.

Variables: Describe each of the columns in your dataset, providing as much information as you can on the full name of the variable, data type (e.g. continuous, discrete, categoric, etc.), units, and how it was collected.

5 Functions

5.1 Documentation

Functions are saved as `'.R'` files in the `'R'` folder. The name of the file needs to correspond to the function, e.g. the file `time_bins.R` contains the function `time_bins()`. For documentation, we use `roxygen2`. The title, a brief description, and every argument (including the input class and default input) and the output of the function need to be documented in `roxygen2` style, for example:

```
#' An exemplary function
#' This function is used to demonstrate the documentation of functions.
#' @param example \code{character}. Arguments are the function inputs.
#' @param another_example \code{logical}. All arguments need to be documented.
#' @return A \code{list} is returned as output in in this example function.
#' @details Describe more details if necessary, and list sources if applicable.
#' @section Developer:
#' Your name
#' @section Reviewer:
#' Name(s)
#' @examples
#' #Show off the example function
#' example_function(example = "documentation", another_example = TRUE)
#' @export
```

Add the `'@export'` namespace tag to make the function available.

To get started with `roxygen2`, set your working directory to your package directory, or to the directory where you store your function as a `'.R'` file. The R command `devtools::document()` creates a 'man' folder in the directory, which contains a `'.Rd'` file corresponding to your documented function. Opening that file in RStudio, you can create a preview to see what the documentation looks like. After you have implemented changes, rebuild your documentation file with Ctrl+Shift+B or `devtools::document()`.

5.2 Efficiency

When possible, you should make coding decisions which will ensure that your code is maximally efficient - this could make a big difference to users who want to apply your function to a large dataset. A few general examples include:

- Using functions from the `'apply'` family rather than for-loops
- Storing objects as lists, or lists of lists, rather than data frames
- Vectorise, when possible
- Avoiding using `rbind()` and `cbind()` to compile objects row-wise or column-wise within for-loops; specifying the row or column number using the iteration number is usually a faster alternative.

However, please don't let this deter you - we welcome submissions from R users of all experience levels, and our team of in-house code evaluators can help you with any concerns about efficiency.

5.3 Error messages

To ensure that the functions are used appropriately, error messages should be generated when the function is receiving input it is not designed for. Error messages consist of a brief description of what went wrong. Sometimes it makes sense to specify where it went wrong, and what kind of input was expected. Optionally, error messages can include hints to guide the user towards correct input.

Examples of error messages include

- input of the wrong format, e.g. `"Error: 'x' must be a numeric vector."`
- input of the wrong dimension, e.g. `"Error: 'x' must be a data.frame with two columns."`
- input without mandatory names, e.g. `"Error: 'x' does not have a column 'stage'. 'x' must be a data.frame with the columns 'stage' and 'age'."`

To implement error messages in R, the `stop()` function can be used:

```
#generate error message
if(!is.numeric(x)) stop("Error: 'x' must be a numeric vector.")
```

5.4 Warning messages

In general, we try to minimise the use of warning messages in `palaeoverse` as these can be easily ignored by the user, or completely skipped in pipeline analyses (see more?). However, as with most things in life, there is a time and a place. So, while the use of `warning()` is generally discouraged if throwing an error will suffice, we do allow the use of warning messages in `palaeoverse`, where appropiate.

To implement warning messages in R, the `warning()` function can be used:

```
#generate error message
if(!is.numeric(x)) warning("Warning: 'x' must be a numeric vector.")
```

5.5 Tests

Testing is perhaps one of the most important parts of developing a function. If you have not already, we recommend reading through Chapter 12–Testing of 'R packages' before continuing here as `palaeoverse` follows this guidance.

Before a function can be added to `palaeoverse`, it needs to go through formal testing. This is required as **hopefully** your function will be very popular, and we need to ensure that it behaves as expected to avoid any issues. To do so, we make use of the R package `testthat`.

The initial setup for function testing with `testthat` is already established in palaeoverse, under the directory `'tests/testthat/'`. The organisation of test files must match that of `'R/'` files in `palaeoverse`. For example, the function `time_bins()` is saved as `time_bins.R` in the `R/` directory, and has an associated

test file of 'test-time_bins.R' in the 'tests/testthat' directory. This ensures that associated function testing is clear.

Tip: the usethis package provides a helpful pair of functions for creating/alternating between these files:

- usethis::use_r()
- usethis::use_test()

Make sure to create enough tests within your .R file to cover all of the possible variants of a function. This includes creating tests that cover most or all optional arguments and the majority of options for those arguments (and the required arguments). Remember, even if you personally would not use a function for a particular reason, you must attempt to cover the majority of edge cases that may arise that are allowed by the function.

Related tests should be bundled within test_that() calls combined with strings of text to identify the broad reason for each bundle of tests (e.g., testing a function works with a particular type of data). Finally, if tests rely on data or packages outside of the palaeoverse community must have, they should be skipped if those data or packages are not available.

Ultimately, we aim to have >90% code coverage, which means 90% of the lines of code in our codebase should be tested by at least one test. Pushing code to GitHub will trigger a code coverage check, which will alert you as to whether you need to write more tests.

```
#test expected outcome is equal to actual outcome
test_that("time_bins() works", {
  expect_equal(nrow(time_bins(interval = c("Maastrichtian"))), 1)
})
```

6 Contributing

6.1 Git and GitHub

We use git via Github (**palaeoverse-community**) to manage our R code and data. If you are not familiar with these tools, there are some excellent free resources available online:

- Setting up git within RStudio (short)
- Collaborating in R with git and GitHub (50 min. youtube video)
- Git and GitHub and R packages (medium length)
- A comprehensive guide to git and R (very long)

6.2 How to contribute?

You (the contributor) should clone the desired repository (i.e. the palaeoverse R package) to your personal computer. Before changes are made, you should switch to a new git branch (i.e., not the main branch). When your changes are complete, you can submit your changes for merging via a pull request ("PR") on GitHub. Note that a complete pull request should include a succinct description of what the code changes do, proper documentation, and unit tests. Only the description is required for the initial pull request and code review (see below), but pull requests will not be merged until they contain complete documentation and tests.

If you are not comfortable with git/GitHub, you can reach out to one of the core developers (see collaborators) via email and they can make a pull request on your behalf. However, you will be expected to respond to any review comments on GitHub (see below).

If you don't feel comfortable implementing changes yourself, you can submit a bug report or feature request as a GitHub issue in the proper repository (e.g., palaeoverse issues).

6.3 Code review

All pull requests must be reviewed by two core developers of palaeoverse (see collaborators) before merging. The review process will ensure that contributions 1) meet the standards and expectations as described above, 2) successfully perform the functions that they claim to perform, and 3) don't break any other parts of the codebase.

Submitting a pull request for one of the palaeoverse R packages will automatically initiate an R CMD check, lintr check, and test coverage check via GitHub Actions. While these checks will conduct some automatic review to ensure the package has not been broken by the new code and that the code matches the style guide (see above), a manual review is still required before the pull request can be merged.

Reviewers may have questions while reviewing your pull request. You are expected to respond to any of these questions via GitHub. If fixes and/or changes are required, you are expected to make these changes. If the required changes are minor enough, reviewers may make them for you, but this should not be expected. If you have any questions or lack the background to make the required changes, you should work with the reviewer to determine a plan of attack.