DRY PACKAGE DEVELOPMENT IN R

Indrajeet Patil



"Copy and paste is a design error." - David Parnas

WHY SO DRY

Why should you not repeat yourself?

DON'T REPEAT YOURSELF (DRY) PRINCIPLE

The DRY Principle states that:

Every piece of **knowledge** must have a **single** representation in the codebase.

That is, you should not express the same thing in multiple places in multiple ways.



i It's about knowledge and not just code

The DRY principle is about duplication of knowledge. Thus, it applies to all programming entities that encode knowledge:

- You should not duplicate code.
- You should not duplicate intent across code and comments.
- You should not duplicate knowledge in data structures.
- ...

BENEFITS OF DRY CODEBASE

- When code is duplicated, if you make change in one place, you need to make parallel changes in other places. When code is DRY, parallel modifications become unnecessary.
- Easy to maintain since there is only a single representation of knowledge that needs to be updated if the underlying knowledge changes.
- As a side effect, routines developed to remove duplicated code can become part of general-purpose utilities.

Further Reading

- Thomas, D., & Hunt, A. (2019). The Pragmatic Programmer. Addison-Wesley Professional. (pp. 30-38)
- Understand distinction between DRY and DAMP (Descriptive And Meaningful Phrases)

PLAN

Apply the DRY principle to remove duplication in following aspects of R package development:

- Documentation
- Vignette setup
- Unit testing
- Dependency management
- Exceptions

DOCUMENTATION

How not to repeat yourself while writing documentation.

WHAT DO USERS READ?

What users consult to find needed information may be context-dependent.







Thus, including crucial information **only in one place** makes it likely that the users might miss out on it in certain contexts.

GO FORTH AND MULTIPLY (WITHOUT REPETITION)

Some documentation is important enough to be included in multiple places (e.g. in the function documentation and in a vignette).

How can you document something just once but include it in multiple locations?



CHILD DOCUMENTS

You can stitch an R Markdown document from smaller child documents.



Thus, the information to repeat can be stored *once* in child documents and reused *multiple* times across parents.

STORING CHILD DOCUMENTS IN PACKAGE

Stratagem: You can store child documents in the manual directory and reuse them.

Child documents

```
├─ DESCRIPTION 
├─ man 
├─ rmd-children 
├─ info1.Rmd 
├─ ...
```

info1.Rmd example:

```
1 This is some crucial information to be repeated across documentation. 2 3 ```{r} 4 1 + 1 5 ```
```

USING CHILD DOCUMENTS IN PACKAGE: PART-1

Include contents of child documents in the documentation in multiple locations.

Vignette

```
    DESCRIPTION
    vignettes
    vignette1.Rmd
    web_only
    vignette2.Rmd
    ...
```

README

```
DESCRIPTION
README.Rmd
```

In vignette1.Rmd

```
1 ---
2 output: html_vignette
3 ---
4
5 Vignette content.
6
7 ```{r, child="../man/rmd-children/info1.Rmd"}
8 ```
```

n README. Rmd

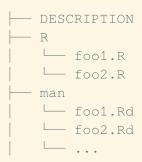
```
1 ---
2 output: github_document
3 ---
4
5 README content.
6
7 ```{r, child="man/rmd-children/info1.Rmd"}
8 ```
```

Source code for the slides can be found here.

USING CHILD DOCUMENTS IN PACKAGE: PART-2

Include contents of child documents in the documentation in multiple locations.

Manual



In foo1.R

```
1 #' @title Foo1
2 #' @section Information:
3 #'
4 #' ``` {r, child="man/rmd-children/info1.Rmd"}
5 #' ```
6 foo1 <- function() { ... }</pre>
```

WHAT ABOUT NON-CHILD DOCUMENTS?

You can include contents from any file in . Rmd, not just a child document!

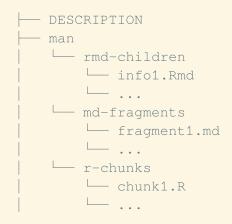




STORING OTHER DOCUMENTATION FILES IN PACKAGE

Like child documents, other types of documents are also stored in man/folder.

Reusable content



fragment1.md example:

```
1 This `.md` file contains
2 content to be included *as is*
3 across multiple locations
4 in the documentation.
```

chunk1.R example:

```
# some comment and code
1 + 1
3
4 # more comments and code
5 2 + 3
```

Source code for the slides can be found here.

USING NON-CHILD DOCUMENTS IN PACKAGE: PART-1

Include contents of various files in the documentation in multiple locations.

Vignette

README

```
DESCRIPTION
README.Rmd
```

In vignette1.Rmd

```
1 ---
2 output: html_vignette
3 ---
4
5 Vignette content.
6
7 ```{asis, file="../man/md-fragments/fragment1.md"}
8 ```
9
10 ```{r, file="../man/r-chunks/chunk1.R"}
11 ```
```

n README. Rmd

```
1 ---
2 output: github_document
3 ---
4
5 README content.
6
7 ```{asis, file="man/md-fragments/fragment1.md"}
8 ```
9
10 ```{r, file="man/r-chunks/chunk1.R"}
11 ```
```

Source code for the slides can be found here.

USING NON-CHILD DOCUMENTS IN PACKAGE: PART-2

Include contents of child documents in the documentation in multiple locations.

Manual

In foo1.R

```
1 #' @title Foo1
2 #' @section Information:
3 #'
4 #' ```{r, file="man/md-fragments/fragment1.Rmd"}
5 #' ```
6 #'
7 #' @example man/r-chunks/chunk1.R
8 foo1 <- function() { ... }</pre>
```



Summary on how to repeat documentation

If you are overwhelmed by this section, note that you actually need to remember only the following rules:

- Store reusable document files in the /man folder.
- When you wish to include their contents, provide paths to these files **relative** to the document you are linking from.
- If it's a child . Rmd document, use the child option to include its contents.
- If it's not an .Rmd document, use the file option to include its contents and use appropriate {knitr} engine. To see available engines, run

```
names(knitr::knit_engines$get()).
```

SELF-STUDY

Example packages that use reusable component documents to repeat documentation.

- {pkgdown}
- {ggstatsplot}
- {statsExpressions}

VIGNETTE SETUP

How not to repeat yourself while setting up vignettes.

SETUP CHUNKS IN VIGNETTES

Another duplication that occurs is in setup chunks for vignettes.

For example, some parts of the setup can be same across vignettes.

```
├─ DESCRIPTION
├─ vignettes
| └─ vignette1.Rmd
| └─ vignette2.Rmd
| └─ ...
```

```
1 ---
2 title: "Vignette-1"
3 output: html_vignette
4 ---
5
6 ```{r}
7 knitr::opts_chunk$set(
8 message = FALSE,
9 collapse = TRUE,
10 comment = "#>"
11 )
12 ```
```

```
1 ---
2 title: "Vignette-2"
3 output: html_vignette
4 ---
5
6 ```{r}
7 knitr::opts_chunk$set(
8 message = FALSE,
9 collapse = TRUE,
10 comment = "#>"
11 )
12
13 options(crayon.enabled = TRUE)
14 ```
```

SOURCING SETUP CHUNKS IN VIGNETTES

This repetition can be avoided by moving the **common** setup to a script, and sourcing it from vignettes. Storing this script in a folder (/setup) is advisable if there are many reusable artifacts.

Option 1

```
├─ DESCRIPTION 
├─ vignettes 
├─ setup.R
```

Option 2

```
    DESCRIPTION
    vignettes
    setup
    setup.R
```

setup.R contents

```
1 knitr::opts_chunk$set(
2 message = FALSE,
3 collapse = TRUE,
4 comment = "#>"
5 )
```

Sourcing common setup

```
1 ---
2 title: "Vignette-1"
3 output: html_vignette
4 ---
5
6 ```{r setup, include = FALSE}
7 source("setup/setup.R")
8 ```
```

Sourcing common setup

```
1 ---
2 title: "Vignette-2"
3 output: html_vignette
4 ---
5
6 ```{r setup, include = FALSE}
7 source("setup/setup.R")
8 options(crayon.enabled = TRUE)
9 ```
```

Source code for the slides can be found here.

SELF-STUDY

Packages in the wild that use this trick.

- {dm}
- {statsExpressions}

DATA

How not to repeat yourself while creating and re-using example datasets.

ILLUSTRATIVE EXAMPLE DATASETS

If none of the existing datasets are useful to illustrate your functions, you can create new datasets.

Let's say your example dataset is called exdat and function is called foo(). Using it in examples, vignettes, README, etc. requires that it be define multiple times.

In examples

```
1 #' @examples
2 #' exdat <- matrix(c(71, 50))
3 #' foo(exdat)</pre>
```

In vignettes

```
1 ---
2 title: "My Vignette"
3 output: html_vignette
4 ---
5
6 ```{r}
7 exdat <- matrix(c(71, 50))
8 foo(exdat)
9 ```</pre>
```

Source code for the slides can be found here.

In README

```
1 ---
2 output: github_document
3 ---
4
5 ```{r}
6 exdat <- matrix(c(71, 50))
7 foo(exdat)
8 ```</pre>
```

SHIPPING DATA IN A PACKAGE

You can avoid this repetition by defining the data just *once*, saving and shipping it with the package.

The datasets are stored in data/, and documented in R/data.R.

Saving data

```
1 # In `exdat.R`
2 exdat <- matrix(c(71, 50))
3 save(exdat, file="data/exdat.rdata")</pre>
```

Directory structure

```
    DESCRIPTION
    R
    data-raw
    l     exdat.R
    data
    l     exdat.rdata
    l     R
    l     l     exdat.rdata
    l     l     exdat.rdata
    l     l     exdat.R
```

REUSABLE DATASET

exdat can now be used in examples, tests, vignettes, etc.; there is no need to define it every time it is used.

In examples

1 #' @examples 2 #' foo(exdat)

In vignettes

```
2 title: "My Vignette"
3 output: html_vignette
4 ---
5
6 ```{r}
```

7 foo(exdat)
8 ```

Source code for the slides can be found here.

In README

```
1 ---
2 output: github_document
3 ---
4
5 ```{r}
6 foo(exdat)
7 ```
```

SELF-STUDY

Examples of R packages that define datasets and use them repeatedly.

- {ggstatsplot}
- {effectsize}

UNIT TESTING

How not to repeat yourself while writing unit tests.

REPEATED TEST PATTERNS

A unit test records the code to describe expected output.



Unit testing involves checking function output with a range of inputs, and this can involve recycling a test pattern.



But such recycling violates the DRY principle. How can you avoid this?

PARAMETRIZED UNIT TESTING

To avoid such repetition, you can write parameterized unit tests using {patrick}.

Repeated test pattern

expect_identical() used repeatedly.

```
1 test_that(
2  desc = "multiplier works as expected",
3  code = {
4   expect_identical(multiplier(-1, 3), -3)
5   expect_identical(multiplier(0, 3.4), 0)
6   expect_identical(multiplier(NA, 4), NA_real_)
7   expect_identical(multiplier(-2, -2), 4)
8   expect_identical(multiplier(3, 3), 9)
9  }
10 )
```

Parametrized test pattern

expect_identical() used once.

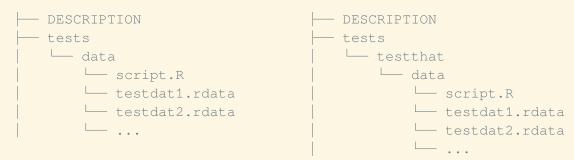
Source code for the slides can be found here.

REPEATED USAGE OF TESTING DATASETS

You have already seen how user-facing datasets — useful for illustrating function usage — can be defined and saved once and then used repeatedly.

Similarly, you can define and save developer-facing datasets - useful for testing purposes - and use them across multiple tests.

Saving datasets in either of these locations is fine.



Save the script!

Always save the script used to create datasets. This script:

- acts as documentation for the datasets
- makes it easy to modify the datasets in the future (if needed)

USING TEST DATASETS

Without stored datasets, you define the same datasets multiple times across test files.

n test-fool.R:

In test-foo2.R:

. . .

```
1 testdat1 <- { ... }
2 foo1(testdat1)
```

```
1 testdat1 <- { ... }
  2 foo2(testdat1)
$ource code for the slides can be found here.
```

SELF-STUDY

Examples of R packages that save datasets required for unit testing.

- {ospsuite}
- {dm}

EXCEPTIONS

How not to repeat yourself while signalling exceptions

SENDING SIGNALS

Exceptions/conditions (messages, warnings, and errors) provide a way for functions to signal to the user that something unexpected happened. Often, similar exceptions need to be signalled across functions.

E.g., for functions that don't accept negative values:

input validation

```
1 foo1 <- function(x) {
2   if (x < 0) {
3     stop("Argument `x` should be positive.")
4   }
5
6   ...
7 }</pre>
```

```
1 foo2 <- function(y) {
2   if (y < 0) {
3    stop("Argument `y` should be positive.")
4   }
5
6   ...
7 }</pre>
```

LIST OF EXCEPTION FUNCTIONS

We can avoid this repetition by extracting exception message strings in a function with an informative name. And then storing them in a list.

```
1 exceptions <- list(
2   only_positives_allowed = function(arg_name) {
3     paste0("Argument `", arg_name, "` should be positive.")
4   },
5
6   ... # you can store as many functions as you want
7 )</pre>
```

REUSABLE EXCEPTIONS: PART-1

We can then use these functions to signal exceptions.

Input validation

```
1 foo1 <- function(x) {
2    if (x < 0) {
3        stop(exceptions$only_positives_allowed("x"))
4    }
5
6    ...
7 }</pre>
```

```
1 foo2 <- function(y) {
2    if (y < 0) {
3        stop(exceptions$only_positives_allowed("y"))
4    }
5
6    ...
7 }</pre>
```

REUSABLE EXCEPTIONS: PART-2

As noted before, you can also move the entire validation to a new function. E.g.

```
1 exceptions <- list(
2    check_only_positive = function(arg) {
3        arg_name <- deparse(substitute(arg))
4        if (arg < 0) {
5            stop(paste0("Argument `", arg_name, "` should be positive."))
6        }
7     },
8     ... # you can store as many functions as you want
9 )</pre>
```

DRY ONCE, DRY MULTIPLE TIMES

Most often the exceptions will be useful only for the package in which they are defined in. But, if the exceptions are generic enough, you can even export them. This will make them reusable not only in the current package, but also in other packages.

That is, DRYing up exceptions in one package does the same for many!

(i) Why a list?

It is not necessary that you store exceptions in a list; you can create individual functions outside of a list and export them.

But storing them in a list has the following advantages:

- Simpler NAMESPACE: There is a single export for all exceptions (e.g. exceptions), instead of dozens (e.g., only_positives_allowed(), only_negatives_allowed, only_scalar_allowed(), etc.), which can overpower the rest of the package API.
- Extendability: You can easily append a list of imported exceptions by adding more exceptions which are relevant only for the current package. E.g. exceptions \$my_new_exception_function <- function() {...}

SELF-STUDY

Example of R package that create a list of exception functions and exports it:

```
{ospsuite.utils}
```

Example of R package that imports this list and appends it:

```
{ospsuite}
```

DEPENDENCY MANAGEMENT

How not to repeat yourself while importing external package functions.

IMPORTS

Instead of using :: to access external package function (rlang::warn()), you can
specify imports explicitly via roxygen directive #' @importFrom.

But if you are importing some functions multiple times, you should avoid specifying the import multiple times, and instead collect all imports in a single file.

Import statements scattered across files: In {pkgname}-package.R file:

```
1 # file-1
                                                               ## {pkgname} namespace: start
 2 #' @importFrom rlang warn
                                                               #' @importFrom rlang warn seq2
                                                               #' @importFrom purrr pluck
 5 # file-2
 6 #' @importFrom rlang warn
                                                               ## {pkgname} namespace: end
                                                            7 NULL
9 #' @importFrom purrr pluck
10 ...
12 # file-3
13 #' @importFrom rlang warn seq2
14 ...
16 # file-4, file-5, etc.
17 ...
                                       Source code for the slides can be found here.
```

SELF-STUDY

Examples of R packages that list the NAMESPACE imports in a single file this way.

- {usethis}
- {lintr}

CONCLUSION

You can use these techniques to avoid repetition while developing R packages, which should make the development workflow faster, more maintainable, and less error-prone.

ADVANCED

Although related to package development at a meta level, these issues are beyond the scope of the current presentation. I can only point to resources to help you get started.

- DRY GitHub Actions workflows
 - https://github.com/krlmlr/actions-sync
 - https://github.com/rstudio/education-workflows
- DRY pkdown website templates for multiple packages in organizations
 - https://github.com/tidyverse/tidytemplate
 - https://github.com/easystats/easystatstemplate

FOR MORE

If you are interested in reading more of my slide decks on related topics, visit this page.

FIND ME AT...

- **Y** Twitter
- in LikedIn
- **GitHub**
- **W**ebsite
- **∑**E-mail

THANK YOU

And Happy (DRY) Package Development! 😊

SESSION INFORMATION

1 sessioninfo::session_info(include_base = TRUE)	
Source code for the slides can be found here.	