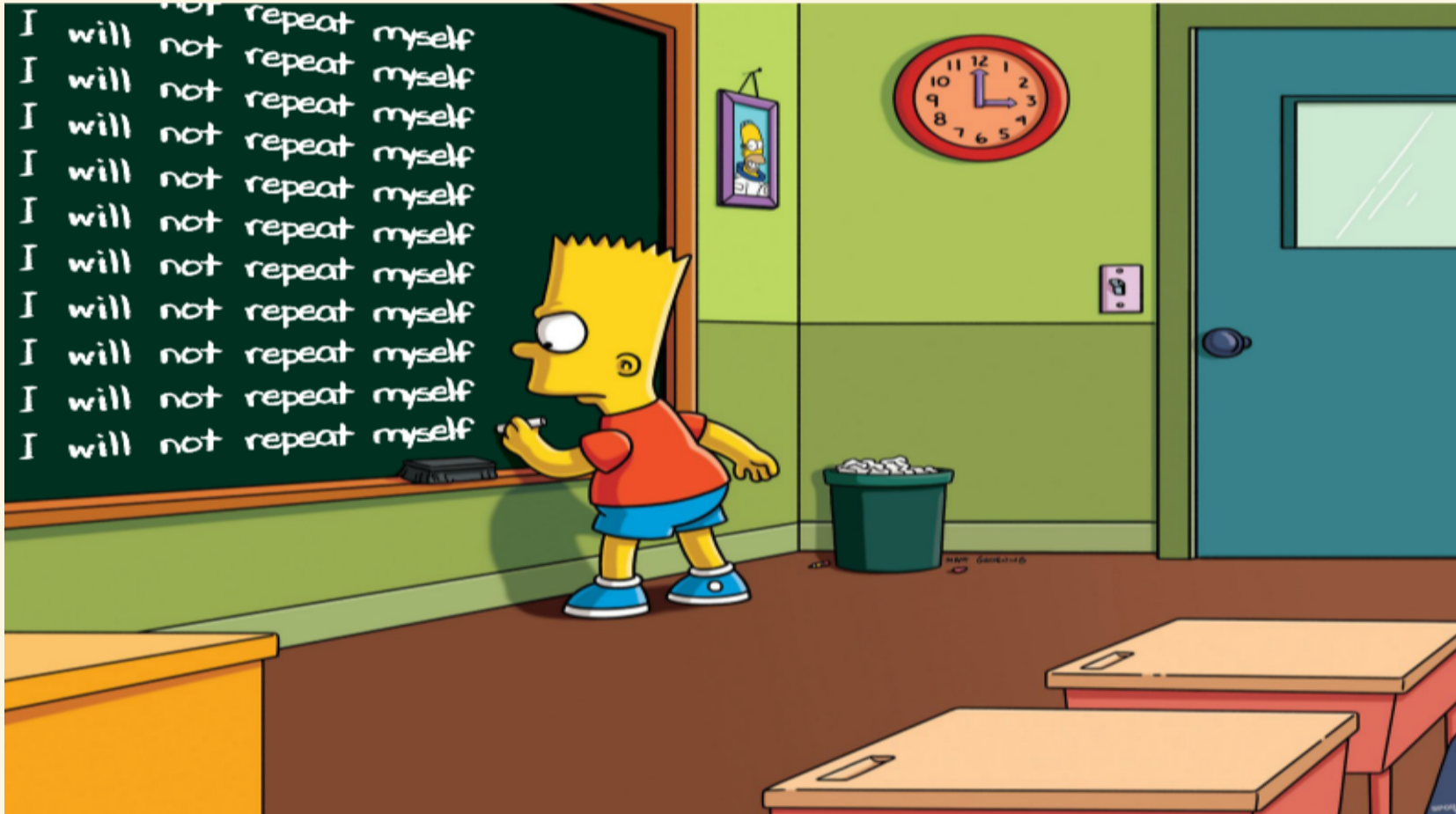# DRY PACKAGE DEVELOPMENT IN R

Indrajeet Patil

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

WHY SO DRY
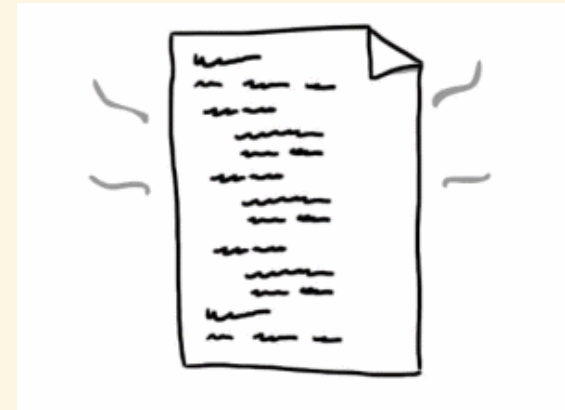
Why should you not repeat yourself?

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***

Source code for the slides can be found here.

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

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.

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.

README: While exploring the package repository.

Vignettes: When first learning how to use a package.

Manual: When checking details about a specific function.

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*

Source code for the slides can be found here.

CHILD DOCUMENTS

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

(parent Rmd) (child Rmd) (result Rmd)

Thus, the information to repeat can be stored *once* in child

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   ```
```

💡 **Tips**

- You can include as many child documents as you want.
- The child document is just like any `.Rmd` file and can include everything that any other `.Rmd` file can include.
- You can choose a different name for the folder containing child documents (e.g. `rmd-fragments`).
- Make sure to include `Roxygen: list(markdown = TRUE)` field in the `DESCRIPTION` file.
- The child documents will not pose a problem either for `R CMD check` or for `{pkgdown}` website.

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  ```
```

In `README.Rmd`

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

USING CHILD DOCUMENTS IN PACKAGE: PART-2

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

Manual

```
├── DESCRIPTION
├── R
│     └── foo1.R
│     └── foo2.R
├── man
│     └── foo1.Rd
│     └── foo2.Rd
│     └── ...
```

In `foo1.R`

```
1  #' @title Foo1
2  #' @section Information:
3  #'
4  #' ```{r, child="man/rmd-children/info1.Rmd"}
5  #' ```
6  foo1 <- function() { ... }
```
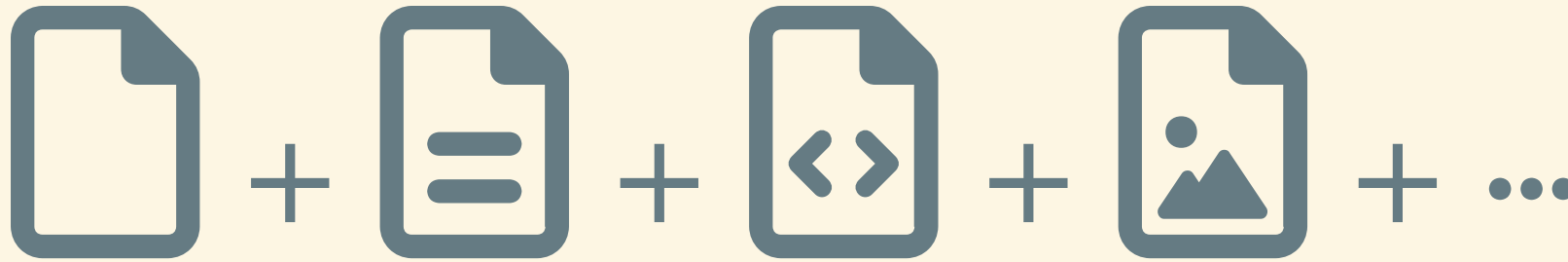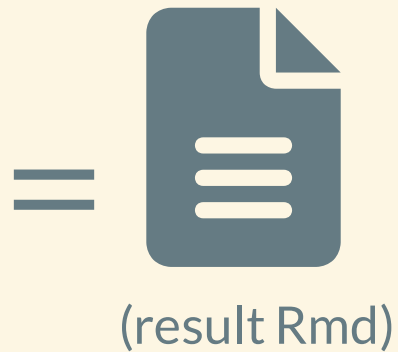
> ⊘ **Important**
>
> The underlying assumption here is that you are using `{roxygen2}` to generate package documentation.

WHAT ABOUT NON-CHILD DOCUMENTS?

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

(parent Rmd) + (child Rmd) + (`.R` + R engine) + (`.md` + `asis` engine) + ...

= (result Rmd)

STORING OTHER DOCUMENTATION FILES IN PACKAGE

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

Reusable content

```
├── DESCRIPTION
├── man
│   └── rmd-children
│       └── info1.Rmd
│       └── ...
│   └── md-fragments
│       └── fragment1.md
│       └── ...
│   └── r-chunks
│       └── chunk1.R
│       └── ...
```

`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:

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

Source code for the slides can be found here.

Folder names

USING NON-CHILD DOCUMENTS IN PACKAGE: PART-1

Include contents of various files 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  ```{asis, file="../man/md-fragments/fragment1.md"}
 8  ```
 9
10  ```{r, file="../man/r-chunks/chunk1.R"}
11  ```
```

In `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  ```
```

USING NON-CHILD DOCUMENTS IN PACKAGE: PART-2

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

Manual

```
├── DESCRIPTION
├── R
│   └── foo1.R
│   └── ...
├── man
│   └── foo1.Rd
│   └── ...
```

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() { ... }
```

> **⚠ Important**
>
> Source code for the slides can be found here.

💡 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  ```
```

How can this repetition be avoided?

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  ```
```

> 💡 **No parallel modification**
>
> Now common setup can be modified with a change in only one place.

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)
```

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  ```
```

In README

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

How can this repetition be avoided?

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")
```

Directory structure

```
├── DESCRIPTION
├── R
├── data-raw
│   └── exdat.R
├── data
│   └── exdat.rdata
├── R
│   └── data.R
```

> 🚧 **Don't forget!**

- For future reference, save script (in `data-raw/` folder) to (re)create or update the dataset.

- If you include datasets, set `LazyData: true` in the `DESCRIPTION` file. Source code for this slide can be found here.

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

```
1  ---
2  title: "My Vignette"
3  output: html_vignette
4  ---
5
6  ```{r}
7  foo(exdat)
8  ```
```

In README

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

> 💡 **No parallel modification**

Note that if you now wish to update the dataset, you need to change its definition only in one place!

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.

(actual) ⟷ (expected)

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

> 🚧 **Not DRY**
>
> But such recycling violates the DRY principle. How can you avoid this?

```r
 1  # Function to test
 2  multiplier <- function(x, y) {
 3    x * y
 4  }
 5
 6  # Tests
 7  test_that(
 8    desc = "multiplier works as expected",
 9    code = {
10      expect_identical(multiplier(-1, 3),   -3)
11      expect_identical(multiplier(0,  3.4), 0)
12      expect_identical(multiplier(NA, 4),   NA_real_)
13      expect_identical(multiplier(-2, -2),   4)
14      expect_identical(multiplier(3,  3),    9)
15    }
16  )
```

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.

```
 1  patrick::with_parameters_test_that(
 2    desc_stub = "multiplier works as expected",
 3    code = expect_identical(multiplier(x, y), res),
 4    .cases = tibble::tribble(
 5      ~x,   ~y,   ~res,
 6      -1,   3,    -3,
 7      0,    3.4,  0,
 8      NA,   4,    NA_real_,
 9      -2,   -2,   4,
10      3,    3,    9
11    )
12  )
```

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.

```
├── DESCRIPTION
├── tests
│   └── data
│       └── script.R
│       └── testdat1.rdata
│       └── testdat2.rdata
│       └── ...
```

```
├── DESCRIPTION
├── tests
│   └── testthat
│       └── data
│           └── script.R
│           └── testdat1.rdata
│           └── testdat2.rdata
│           └── ...
```

> 🚧 **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.

In `test-foo1.R`:   In `test-foo2.R`:   `...`

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

```
1  testdat1 <- { ... }
2  foo2(testdat1)
```

With saved datasets, you define just **once** and load them from test files.

In `test-foo1.R`:   In `test-foo2.R`:   `...`

```
1  testdat1 <- readRDS("testdat1")
2  foo1(testdat1)
```

```
1  testdat1 <- readRDS("testdat1")
2  foo2(testdat1)
```

> (i) **Note**
>
> The exact path provided to `readRDS()` will depend on where the datasets are stored inside the `tests/` folder.

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  }
```

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

unit testing

```
1  expect_error(
2    foo1(-1),
3    "Argument `x` should be positive."
4  )
```

```
1  expect_error(
2    foo2(-1),
3    "Argument `y` should be positive."
4  )
```

How can this repetition be avoided?

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  )
```

> ⓘ **Why not include the entire validation?**
>
> You can move the entire `if()` block to `only_positives_allowed()` and create a new validation function.
>
> But this is not done here to address the most general case where:
>
> - the exception message string can be used outside of an `if()` block
> - it can be used not only as a message, but may be as a warning or an error

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  }
```

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

Unit testing

```
1  expect_error(
2    foo1(-1),
3    exceptions$only_positives_allowed("x")
4  )
```

```
1  expect_error(
2    foo2(-1),
3    exceptions$only_positives_allowed("y")
4  )
```

> 💡 **No parallel modification**
>
> Note that if you now wish to change the condition string, this change needs to be made only in *one* place!

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  )
```

Input validation

```
1  foo1 <- function(x) {
2    check_only_positive(x)
3
4    ...
5  }
```

```
1  foo2 <- function(y) {
2    check_only_positive(y)
3
4    ...
5  }
```

Unit testing

```
1  x <- -1
2  expect_error(
3    exceptions$check_only_positive(x),
4    "Argument `x` should be positive."
5  )
```

Since the validation has moved to a new function, you only need to test it once.

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!

> ⓘ **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:

```
 1  # file-1
 2  #' @importFrom rlang warn
 3  ...
 4
 5  # file-2
 6  #' @importFrom rlang warn
 7  ...
 8
 9  #' @importFrom purrr pluck
10  ...
11
12  # file-3
13  #' @importFrom rlang warn seq2
14  ...
15
16  # file-4, file-5, etc.
```

In `{pkgname}-package.R` file:

```
 1  ## {pkgname} namespace: start
 2  #'
 3  #' @importFrom rlang warn seq2
 4  #' @importFrom purrr pluck
 5  #'
 6  ## {pkgname} namespace: end
 7  NULL
```

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...

 Twitter

in LikedIn

 GitHub

 Website

 E-mail

THANK YOU

And Happy (DRY) Package Development! 😊

SESSION INFORMATION

```r
1  sessioninfo::session_info(include_base = TRUE)
```

```
─ Session info ──────────────────────────────────────────────────
 setting  value
 version  R version 4.2.2 (2022-10-31)
 os       Ubuntu 22.04.1 LTS
 system   x86_64, linux-gnu
 ui       X11
 language (EN)
 collate  C.UTF-8
 ctype    C.UTF-8
 tz       UTC
 date     2023-01-30
 pandoc   2.19.2 @ /usr/bin/ (via rmarkdown)

─ Packages ──────────────────────────────────────────────────────
 package      * version date (UTC) lib source
 base         * 4.2.2   2022-10-31 [3] local
 cli            3.6.0   2023-01-09 [1] RSPM
 compiler       4.2.2   2022-10-31 [3] local
 datasets     * 4.2.2   2022-10-31 [3] local
 digest         0.6.31  2022-12-11 [1] RSPM
 evaluate       0.20    2023-01-17 [1] RSPM
```