Using R Packages for Reproducible Workflows

Michael Dumelle

September 22, 2021

Contents

Overview

Welcome to the 2021 EPA R Workshop titled "Using R Packages for Reproducible Workflows" by me, Michael Dumelle – I am glad to have you here! Before proceeding, let's first start up R and download the devtools package.

```
install.packages("devtools") # if required
```

The workshop's companion R package (Using **R** **P**ackages for **R**eproducible **W**orkflows) is available for download via

```
devtools::install_github(repo = "michaeldumelle/RPRW", ref = "main")
library(RPRW)
```

Here are the sections of this workshop:

1. Building an R Package
2. A Research Compendia for an overview of an effective research compendia
3. Turning An R Package into a Reproducible Research Compendia
4. Extensions
5. Exercise Solutions

Acknowledgements

I would like to thank Charlotte Wickham, Hadley Wickham, Jenny Bryan, and Yihui Xie for the immense impact their work has had on my programming journey. Much of this workshop draws from heavily their inspiration. I would also like to thank everyone who helped me hold this workshop.

Chapter 1

Building an R Package

1.1 What is an R Package?

An R package is a collection of code, data, documentation, and tests with a particular structure that can be shared with others. R packages are commonly downloaded from the Comprehensive R Archive Network (CRAN). You can install them from CRAN with `install.packages("package_name")`, load them in your workspace with `library("package_name")`, and get help by running (`help(package = "package_name")`).

One of the reasons R packages are so useful is because they are the fundamental way to share code in R. If your code is in a package, others can easily download and use it. If they are familiar with R packages, they likely will be familiar with how to use yours! But sharing R code is not the only benefit of creating R packages. Learning how to build an R package will provide several other benefits to future you!

Future you will benefit from creating your own R packages because they enforce a particular structure. This structure

1. Saves you time – you don't need to think about how to organize your files, R packages have a template!
    - This was especially helpful for me because before learning how to create R packages, I would save my R files in all sorts of locations on my computer with all sorts of names. This made it *very challenging* to come back to my work later and find a particular file.
2. Gives you standardized tools – people have created extremely useful tools that work with R packages, so take advantage of them!
    - The R package devtools, which we downloaded earlier, contains many of these standarized tools.
3. Requires documentation – This is especially helpful for future you.

    - Before I started using R packages, when I would come back my old code, I was convinced someone else wrote it – I basically had to rewrite it all to understand it. R packages help prevent this.
4. Is reproducible – R packages are built from R projects (see here and here), so file paths are relative, not absolute!
    - `read_csv("a_fun_csv_file.csv")` works on my machine – and yours!
    - While R projects are not the fundamental focus on today, I highly, highly recommend you use them for every data analysis project that you are not using an R package for.
5. Guides your data analysis – We will talk about this today
    - See Marwick et al. [2018] for more!

1.1.1 Exercises

1. What are some of your favorite R packages?

2. Of those we have talked about so far, what benefits of R packages are most appealing to you?

1.2 Creating an R Package

Together, we will create the companion package for this workshop! If this is your first R package, then an extra special congratulations to you – this is a big milestone! The name of the package is

1.2.1 The Motivating Dataset

Suppose we want to build an R package that summarizes length (in kilometers) and discharge (meters per second cubed) of North American rivers based on the names of the rivers. Below is our data of interest

```r
rivers <- data.frame(
  Missouri = c(3768, 1956),
  Mississippi = c(3544, 18400),
  Yukon = c(3190, 6340),
  Colorado = c(2330, 40),
  Arkansas = c(2322, 1004),
  Columbia = c(2000, 7730),
  Red = c(1811, 852),
  Canadian = c(1458, 174)
)
rownames(rivers) <- c("length", "discharge")
rivers
```

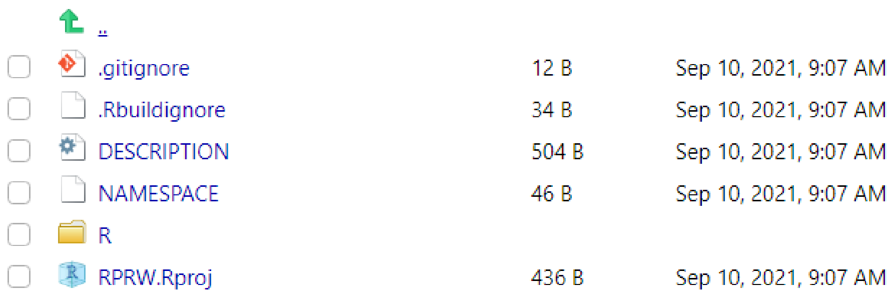
```
#>           Missouri Mississippi Yukon Colorado Arkansas Columbia  Red Canadian
#> length        3768        3544  3190     2330     2322     2000 1811     1458
#> discharge     1956        18400   6340        40      1004      7730   852       174
```

1.2.2 The First Step

```r
devtools::create_package("path_to_RPRW_package/RPRW")
```

```
√ Creating 'path_to_RPRW_package/RPRW/'
√ Setting active project to 'path_to_RPRW_package/RPRW'
√ Creating 'R/'
√ Writing 'DESCRIPTION'
Package: RPRW
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
√ Writing 'NAMESPACE'
√ Writing 'RPRW.Rproj'
√ Adding '^RPRW\\.Rproj$' to '.Rbuildignore'
√ Adding '.Rproj.user' to '.gitignore'
√ Adding '^\\.Rproj\\.user$' to '.Rbuildignore'
√ Opening 'path_to_RPRW_package/RPRW/' in new RStudio session
√ Setting active project to '<no active project>'
√ Setting active project to 'path_to_RPRW_package/RPRW'
```

In the "Files" pane of RStudio (bottom right corner), you will see that your folder has been populated with a few new files and a new folder.



..		
.gitignore	12 B	Sep 10, 2021, 9:07 AM
.Rbuildignore	34 B	Sep 10, 2021, 9:07 AM
DESCRIPTION	504 B	Sep 10, 2021, 9:07 AM
NAMESPACE	46 B	Sep 10, 2021, 9:07 AM
R		
RPRW.Rproj	436 B	Sep 10, 2021, 9:07 AM

We will focus on some of these next.

You will also notice that the "Environment" pane of RStudio (top right corner) now has a "Build" tab – this tab contains some useful tools for your R package.

- R (folder): Where functions in your R package are stored
- Description: This contains metadata for your package
- Namespace: Information about 1) functions from other packages your package uses and 2) what functions in your package you make available to others

1.2.3 The First Function

The R folder is where functions in your R package are stored. Let's create our first function, called `river_medians()`, which finds the median of river length and discharge of desired rivers. Generally, the name of your file should match the name of the function, and you should use separate files for separate functions. More experienced users, it is okay to break this rule every once in a while in certain contexts. To create an R file in the R folder, run

```
use_r("river_means")
```

You will see some output in the console

```
* Modify 'R/river_means.R'
* Call `use_test()` to create a matching test file
```

and the appropriate file now in your R folder.

..
river_means.R 0 B Sep 10, 2021, 10:16 AM

Now let's write our function. We want this function to take the mean length and discharge of all rivers whose names have a common pattern.

```
river_means <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  rowMeans(new_data)
}
```

The first line of code finds the variables in the data whose names match the pattern. The second line of code subsets the data to include only the desired variables. The third line of code finds and returns the mean length and mean discharge of the desired variables. If you would like, you can leave comments in the code (using #) to remind yourself what each line of code does. This is usually good practice, as it can help others (and future you) understand the intent of your code line-by-line.

1.2.4 `load_all()`

So we have written this function, how should we try it out? Well you *could* run the function or source it (`source("R/river_means.R")`), but this puts it in our global workspace, and if you are not careful, global workspaces can get messy. The devtools function `load_all()` emulates the process of building, installing, and loading our R package, making all of your functions available via a single line of code. As your package becomes more complicated, it is much easier to try our your functions using `load_all()` than navigating the global workspace.

```r
load_all()
```

Now let's try out our function! What if we want our summary for the rivers starting with "Mi" (Missiouri and Mississippi)?

```r
river_means(rivers, "Mi")
```

```
#>    length discharge
#>      3656     10178
```

Or what if we want our summary for rivers starting with "Y" (Yukon) or "R" (Red)?

```r
river_means(rivers, "Y|R")
```

```
#>    length discharge
#>    2500.5    3596.0
```

Hooray – our function works! Give yourself a congratulations :)!

1.2.4.1 Exercises

1. How many functions does `river_means()` call within the body of the function? What package are these functions in?

1.2.5 Creating Package Data

So far we have used the `rivers` data from our global workspace. But what if we want this to be data that is included as part of the R packages so that others (or future you) have easy access to it? This is the place for `use_data()`

```r
use_data(rivers)
```

```
✓ Creating 'data/'
✓ Saving 'rivers' to 'data/rivers.rda'
```

You will see that at the root of your R package, there is a folder called "data", and in that folder is a file called rivers.rda, which contains the rivers data. For illustration, let's remove rivers from our global workspace and then load it like we would data from any other package

```r
rm(rivers) # remove the rivers data from our global workspace
load_all() # emulate package building process
data("rivers") # load the rivers data
rivers
```

1.2.6 roxygen Comments

Now that you have written your function, it is time to thoroughly document your function so that others (or future you) can understand how to use it. The documentation for R functions uses special types of comments called *roxygen* comments. The comments use `#'` and have special "tags" associated with them. To insert a template of roxygen comments into `river_means()`, put your cursor somewhere in the `river_means()` function and either

1. In the upper-left toolbar, go to Code -> Insert Roxygen Skeleton
2. Press Ctrl/Cmd + Alt + Shift + R

You will see `river_means()` now looks like

```r
#' Title
#'
#' @param data
#' @param pattern
#'
#' @return
#' @export
#'
#' @examples
river_means <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  rowMeans(new_data)
}
```

The `@` indicates a particular "tag" that controls how functions are documented.

- `@param`: for documenting arguments to a function
- `@return`: for documenting the function output
- `@export`: for making this function available to others downloading your package
- `@examples`: to give examples for how to use the package

Let's fill these in.

```r
#' Means of river lengths and discharges
#'
#' @param data A data frame with two rows. The first row indicates river length and
#'   second row indicates river discharge. The columns of data indicate river names.
#' @param pattern A pattern by which to include only particular rivers
#'
#' @return The mean river length and mean river discharge for the desired rivers
#' @export
#'
#' @examples
#' data("rivers")
#' river_means(rivers, pattern = "Mi")
river_means <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  rowMeans(new_data)
}
```

To view this documentation as you would any other package's function documentation, first run

```r
document()
```

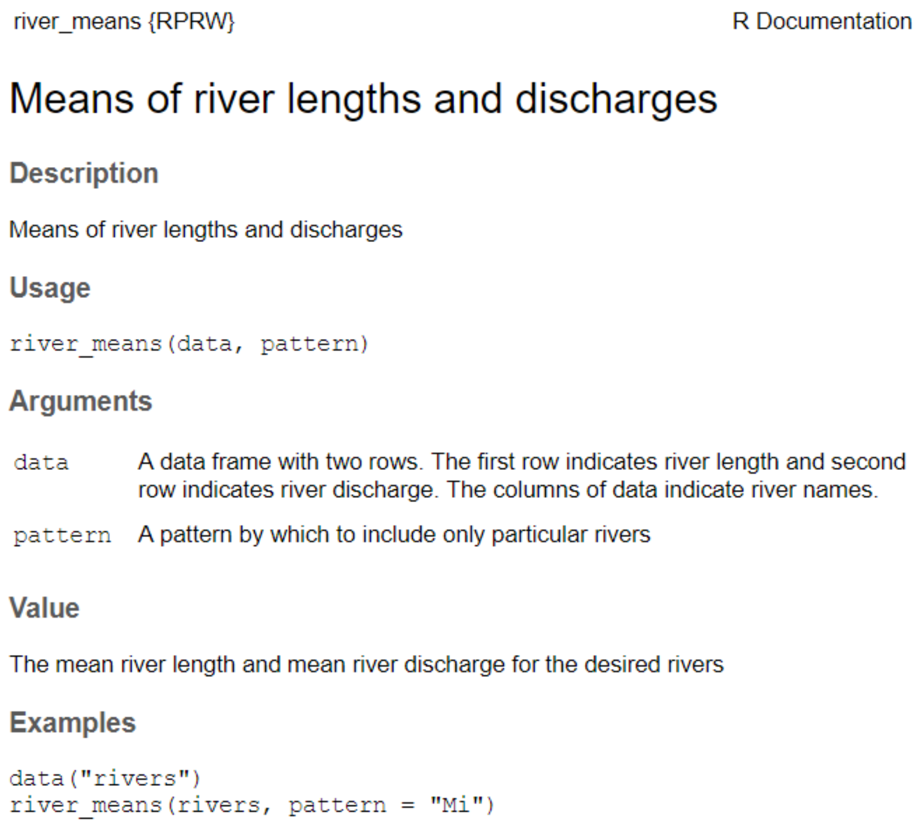
```
i Updating RPRW documentation
i Loading RPRW
Writing NAMESPACE
Writing river_means.Rd
```

You will notice that there is a new folder at the root of the package, called "man", which contains a file called `river_means.Rd`.

```
..
river_means.Rd                658 B          Sep 10, 2021, 11:45 AM
```

The contents of the file here are not too important, as this file is automatically created using `document()`. The important point is that after running `document()`, you can view the documentation of your function!

```r
?river_means
```



river_means {RPRW} R Documentation

Means of river lengths and discharges

Description

Means of river lengths and discharges

Usage

```
river_means(data, pattern)
```

Arguments

`data`	A data frame with two rows. The first row indicates river length and second row indicates river discharge. The columns of data indicate river names.
`pattern`	A pattern by which to include only particular rivers

Value

The mean river length and mean river discharge for the desired rivers

Examples

```
data("rivers")
river_means(rivers, pattern = "Mi")
```

There are several other tags available to customize your documentation – for more information about these tags and documenting data (which is different than documentation function; the rivers data is documented in the RPRW package hosted on GitHub), see here. Though we skip it here, the RPRW package source does document the rivers data and can be viewed using `?rivers` after loading `RPRW`.

1.2.7 The Second Function

In `river_means()`, we used a few functions: `<-`, `grep()`, `names()`, `[`, and `rowMeans()`. These functions are all from the base package, which is a special package in the way that it operates with R packages. If you use functions from the base package in your package, you don't have to give R any warning. If you use a function from another package, however, you do need to let R know where that function is coming from. Next we explore this.

Suppose we want to create a function in our package that computes the mean instead of a mean. First we run

```
use_r("river_medians")
```

Then we write the function. Note that there is no `rowMedians()` function, so we use `apply()` to summarize across rows (see `?apply` for more detail).

```
river_medians <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  apply(new_data, 1, median)
}
```

But wait! You'll notice that the function `median()` is in the stats package, not the base package (run `?median` to check). So we need to let R know that we will be using a function from a package that is not the base package. To accomplish this, there are two steps to take. First, you need to tell R that you are using another package

```
use_package("stats")
```

```
√ Adding 'stats' to Imports field in DESCRIPTION
* Refer to functions with `stats::fun()`
```

You will notice that the package stats was added to the Imports field in the DESCRIPTION file – we will get to DESCRIPTION later, but for now just remember that it contains metadata about your package. You only need to run `use_package("stats")` once per package. That is, if you are using a different function from stats, you don't need to run `use_package("stats")` again. Second, you need to tell the `river_median()` function to use the `median()` function from the stats package. The best practice is to preface any function from outside the base package with `packagename::`. This adjustment to `river_median()` yields

```
river_medians <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  apply(new_data, 1, stats::median)
}
```

Though this is the best practice, it does add some extra typing and can be cumbersome if you are using outside functions often. The `packagename::` prefix can be avoided if you import `median()` to `river_median()` using the roxygen tag `@importFrom`

```
#' Medians of river lengths and discharges
#'
#' @param data A data frame with two rows. The first row indicates river length and
#'   second row indicates river discharge. The columns of data indicate river names.
#' @param pattern A pattern by which to include only particular rivers
#'
```

```r
#' @return The median river length and mean river discharge for the desired rivers
#' @importFrom stats median
#' @export
#'
#' @examples
#' data("rivers")
#' river_medians(rivers, pattern = "Mi")
river_medians <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  apply(new_data, 1, median)
}
```

If you are using several functions from stats, it may be easier to use the `@import` tag, which imports all functions from a package at once.

```r
#' Medians of river lengths and discharges
#'
#' @param data A data frame with two rows. The first row indicates river length and
#'   second row indicates river discharge. The columns of data indicate river names.
#' @param pattern A pattern by which to include only particular rivers
#'
#' @return The median river length and mean river discharge for the desired rivers
#' @import stats
#' @export
#'
#' @examples
#' data("rivers")
#' river_medians(rivers, pattern = "Mi")
river_medians <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  apply(new_data, 1, median)
}
```

Be careful when using `@importFrom` or `@import` to ensure that two functions from different packages don't have the same name – this is when using `packagename::` to prefix the function is crucial.

1.2.7.1 Exercises

1. Play around with using these functions for some new patterns – do you notice anything strange?

These exercises are more challenging, so if they don't make sense now, that is okay! Make sure to re-review the solutions after the workshop.

2. Write a new function, `river_stat()`, that takes a data frame, pattern,

and a general function by which to summarize river length and discharge. This general function should not be an actual function but rather a placeholder so that a user may insert their own function as an argument.

3. Rewrite `river_stat()` so that it also takes additional arguments to the summarizing function (hint: use `...` as an argument)

1.2.8 DESCRIPTION

The DESCRIPTION file is automatically added while creating an R package and contains the R package's metadata.

```
Package: RPRW
Title: R Packages for Reproducible Workflows
Version: 0.0.0.9000
Authors@R:
    person(given = "Michael",
           family = "Dumelle",
           role = c("aut", "cre"),
           email = "first.last@example.com")
Description: A companion R package for "Using R Packages for Reproducible Workflows"
    at the 2021 EPA R Workshop.
License: `use_mit_license()`, `use_gpl3_license()` or friends to pick a
    license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
Depends:
    R (>= 2.10)
Imports:
    stats
```

The DESCRIPTION file is where you track version numbers, authorship, and additional R packages that your R package uses. Two fields in DESCRIPTION do most of the communication regarding how your R package uses additional R packages:

1. Imports: Packages here must be installed in order for your package to work. As a result, any package listed in `Imports` will be installed alongside your package. Packages in `Imports` help build the foundation of your package.

2. Suggests: Packages here enhance your package but are not required for your package to work. You might use suggested packages for enhanced plotting, additional data sets, or more. Packages in `Suggests` can add finishing touches to your package, but they are not part of your package's foundation.

Other fields used to communicate how your R package uses additional R packages are `Depends`, `LinkingTo`, and `Enhances`. The difference between `Depends` and `Imports` is subtle – the general advice is to use `Imports` instead of `Depends`.

1.2.9 NAMESPACE

While the DESCRIPTION file communicates what packages your package *uses*, the NAMESPACE file communicates *how* your package uses these packages. More specifically, the NAMESPACE file controls which functions your package exports (makes available to others) and what functions from what packages must be available for your exported functions to work. This file is automatically generated by devtools and should not be edited by hand.

In `river_median()`, if you called `median` using `stats::median`, your NAMESPACE file will look like

```
# Generated by roxygen2: do not edit by hand

export(river_means)
export(river_medians)
```

If you used the `@importFrom stats median` approach, your NAMESPACE will look like

```
# Generated by roxygen2: do not edit by hand

export(river_means)
export(river_medians)
importFrom(stats,median)
```

If you used the `@import stats` approach, your NAMESPACE will look like

```
# Generated by roxygen2: do not edit by hand

export(river_means)
export(river_medians)
import(stats)
```

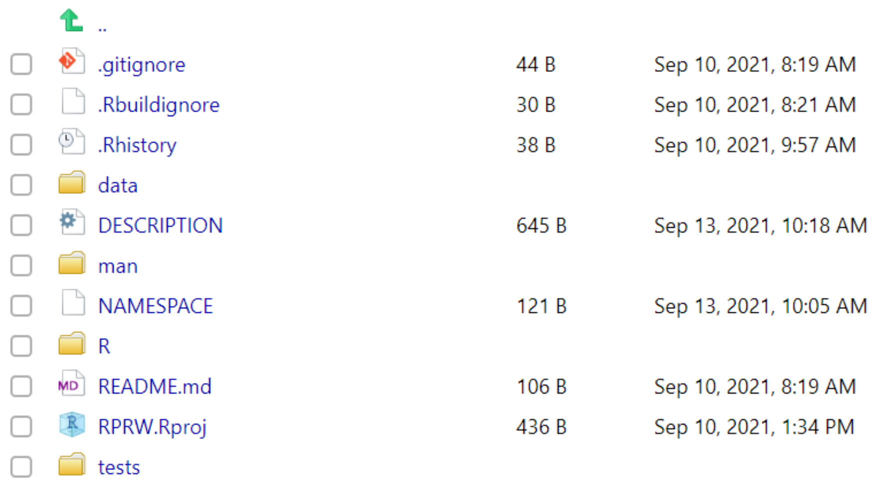
1.2.10 testthat

Testing your code to make sure that it performs as intended is an important step in the package building process. Though upfront, it may seem like extra work, implementing a rigorous testing procedure for your package will provide several benefits: fewer bugs, better code structure, easier restarts, and robust code. In R, testing is incorporated into your package through the testthat package. To begin using testthat, run

```
use_testthat()
```

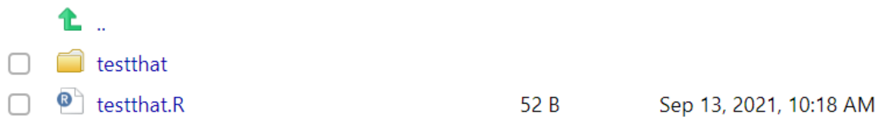
```
✓ Adding 'testthat' to Suggests field in DESCRIPTION
✓ Setting Config/testthat/edition field in DESCRIPTION to '3'
✓ Creating 'tests/testthat/'
✓ Writing 'tests/testthat.R'
```

The root of your package directory shoudl look like



..			
☐	.gitignore	44 B	Sep 10, 2021, 8:19 AM
☐	.Rbuildignore	30 B	Sep 10, 2021, 8:21 AM
☐	.Rhistory	38 B	Sep 10, 2021, 9:57 AM
☐	data		
☐	DESCRIPTION	645 B	Sep 13, 2021, 10:18 AM
☐	man		
☐	NAMESPACE	121 B	Sep 13, 2021, 10:05 AM
☐	R		
☐	README.md	106 B	Sep 10, 2021, 8:19 AM
☐	RPRW.Rproj	436 B	Sep 10, 2021, 1:34 PM
☐	tests		

The tests folder should look like



..			
☐	testthat		
☐	testthat.R	52 B	Sep 13, 2021, 10:18 AM

Tests are generally written on a function-by-function basis. All tests for a function are contained in an R script titled `test-function_name`. For example, to start testing `river_means()`, run

```r
use_test("river_means")
```

```
✓ Writing 'tests/testthat/test-river_means.R'
* Modify 'tests/testthat/test-river_means.R'
```

Your `testthat` folder should look like

..			
☐	test-river_means.R	838 B	Sep 13, 2021, 10:43 AM

Tests can be fairly detailed and cover many components of a function (such as input types, output types, function output, etc.). Here we write a simple test that calculates whether our function, `river_means()`, yields output that we would expect if we calculated the means “by hand” for an example scenario where our pattern includes Missouri and Mississippi.

```r
test_that("the mean length is calculated correctly in a test case", {

  # calculate values required for the test for length

  ## calculate the means from the function
  river_means_val <- river_means(rivers, "Missouri|Mississippi")
  river_means_length <- river_means_val[[1]]

  ## calculate the means "by hand"
  raw_vec_length <- unlist(rivers["length", c("Missouri", "Mississippi")])
  raw_means_length <- mean(raw_vec_length)

  # perform the actual test for length

  ## check that the function and "by hand" output matches
  expect_equal(river_means_length, raw_means_length)
})
```

```
#> Error in test_that("the mean length is calculated correctly in a test case", : coul
```

The tests in testthat are prefixed with `expect_`. If you have many tests, the `test()` function runs all of them in the `testthat` folder:

```r
test()
```

```
i Loading RPRW
i Testing RPRW
√ |  OK F W S | Context
√ |   1       | river_means

== Results =====================================================================
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
```

We can write a similar test for discharge and then repeat the process for `river_meadians()`. Then `test()` returns

```r
test()
```

```
i Loading RPRW
i Testing RPRW
√ |  OK F W S | Context
√ |   2       | river_means
√ |   2       | river_medians

== Results =====================================================================
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 4 ]
```

All tests pass – hooray! I want to again emphasize how important testing is. I know that it seems like an extra chore, but in my experience, writing careful tests has always paid off...with interest.

1.2.10.1 Exercises

1. Write similar tests for `river_means()` (discharge), `river_medians()` (length), and `river_medians()` (length)

1.2.11 License

At some point, your package needs a license. The license places restrictions on how your package can be shared with others. Licensing can be complicated quickly, so I refer you here for more information. For illustration purposes, we will use a GPL-3 license for this package.

```r
use_gpl_license()
```

```
√ Setting active project to 'path_to_RPRW_package/RPRW'
√ Setting License field in DESCRIPTION to 'GPL (>= 3)'
√ Writing 'LICENSE.md'
√ Adding '^LICENSE\\.md$' to '.Rbuildignore'
```

The licensing update will be reflected in the DESCRIPTION file.

1.2.12 Vignettes

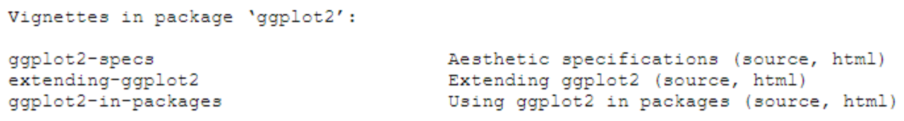
Vignettes act as high-level user guides for your package. A vignette acts as the glue that binds together all the documentation from the individual functions to solve a particular problem. Typically, they guide the user through a typical workflow one would experience while using your package. The ggplot2 package is a popular package for visualizing data. After installing ggplot2

```r
install.packages("ggplot2")
```

you can view its available vignettes by running

```r
vignette(package = "ggplot2")
```

A file will pop up alongside your R scripts with the contents



```
Vignettes in package 'ggplot2':

ggplot2-specs                          Aesthetic specifications (source, html)
extending-ggplot2                      Extending ggplot2 (source, html)
ggplot2-in-packages                    Using ggplot2 in packages (source, html)
```

Then to view a specific vignette, run `vignette(topic, package)`. For example, to view the vignette regarding **aes**thetic specifications, run

```
vignette("ggplot2-specs", "ggplot2")
```

You will then see the vignette in the bottom-right hand pane of RStudio. Vignettes are also available on a package's CRAN page – for the ggplot2 aesthetic specifications vignette, see here.

To include vignettes in your R package, first run

```
use_vignette("river-statistics", "River Statistics")
```

```
√ Adding 'knitr' to Suggests field in DESCRIPTION
√ Setting VignetteBuilder field in DESCRIPTION to 'knitr'
√ Adding 'inst/doc' to '.gitignore'
√ Creating 'vignettes/'
√ Adding '*.html', '*.R' to 'vignettes/.gitignore'
√ Adding 'rmarkdown' to Suggests field in DESCRIPTION
√ Writing 'vignettes/river-statistics.Rmd'
* Modify 'vignettes/river-statistics.Rmd'
```

A few things happen after running `use_vignette()`: your DESCRIPTION file is edited, a vignettes folder is inserted into the root of your directory, and a `.Rmd` file is added to the vignettes folder. The `.Rmd` file extension indicates that the vignette is an RMarkdown document. RMarkdown documents provide a convenient way to create *dynamic* documents. Dynamic documents combine code and text and form the foundation for reproducible report writing in R. More on RMarkdown is available here, here and here. I highly recommend you get some experience with it, as its tools are quite powerful.

While we won't create a vignette for our package during the workshop, I have added a vignette to the companion R package. After installation, it can be viewed by running

```
vignette(river-statistics, "RPRW")
```

1.2.13 `check()`

So now we have built our R package and are ready to share it with the world! But we should probably check to make sure we did not make any small mistakes? This is what `check()` does – it runs through a series of checks on your package to make sure it can be properly installed and shared. `check()` takes a few minutes to run, but it will return errors, warnings, and notes associated with your package. Though the warnings and notes are important, it is most crucial to address the errors immediately. Hopefully your output after running `check()` looks like

```
-- R CMD check results ---------- RPRW 0.0.0.9000 ----
Duration: 34.4s
```

```
0 errors ✓ | 0 warnings ✓ | 0 notes ✓
```

1.2.14 `install()`

After `check()` returns zero errors (and hopefully zero warnings and notes), you can install your package by running

```r
install()
```

After installation, you can use `library()` to load your package (like you do any other R package).

1.2.15 Congratulations

Congratulations on building an R package!

1.3 Debugging

Even the best of programmers write code that may fail in unintended ways. This is referred to as a “bug,” and the process of fixing the “bug” is known as “debugging”. Don’t expect to always write perfect code – do expect to have the tools necessary to track down bugs and remedy them. Though we went through a rigorous documentation and testing procedure when creating `river_means()` and `river_medians()`, there are still bugs present in these functions.

1.3.1 A Mysterious Error Message

We have used `river_means()` and `river_medians()` to successfully find means and medians for all sorts of river combinations. But running

```r
river_means(rivers, "R")
```

```
#> Error in rowMeans(new_data): 'x' must be an array of at least two dimensions
```

yields a mystifying error. Something is wrong – and we need to figure out what. A good first step is to copy and paste the error into a Google search engine and see if anyone has solved the problem yet. If you are lucky, this will help you find the bug. If not, you need to use another approach. Fortunately, R has tools to help isolate bugs.

1.3.2 `traceback()`

The `traceback()` function is run after code generating an error and it identifies where the error occurred.

```r
traceback()
```

```
#> No traceback available
```

Locating the source of bugs goes a long way towards removing them. Here, we see that the function failed at the `rowMeans()` step of `river_means()`. While useful, we still don't exactly know why the error is occurring.

1.3.3 `browser()`

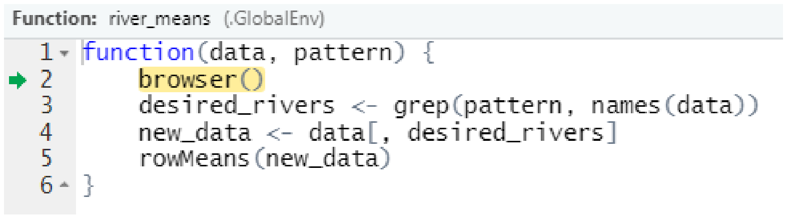
Before I learned about `browser()`, I would try to debug by storing my arguments locally (so that they were in the global environment) and then sequentially running each line of the function one-at-a-time. Perhaps some of you have done this too. Unfortunately, for many reasons (which we don't discuss in detail today), its an inferior method of debugging compared to `browser()`. The `browser()` method involves inserting `browser()` into the body of your function. Running your function with `browser()` inside of it then let's you interactively step into the function at `browser()`. No more storing arguments and running code line-by-line! Let's try this out with `river_means()`

```r
river_means <- function(data, pattern) {
  browser()
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  rowMeans(new_data)
}
```

Now when running `river_means()`, you will step into the function. For example,

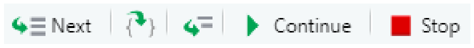
```r
river_means(rivers, "R")
```

will open a new file that looks like



```
Function: river_means (.GlobalEnv)
function(data, pattern) {
    browser()
    desired_rivers <- grep(pattern, names(data))
    new_data <- data[, desired_rivers]
    rowMeans(new_data)
}
```

In your new interactive context, you will see a few buttons in the R console (lower left-hand window)



These are five buttons to type into the console and evaluate (`Enter/Return`) that let you navigate the interactive context. In order (left to right), they are

- Next (`n`) executes the next line of the code
- Step (`s`) steps into the function called by the current line of code
- Finish (`f`) finishes execution of the current function
- Continue (`c`) leaves the interactive context and continues execution of the function
- Stop (`Q`) leaves the interactive context and terminates execution of the function

In the interactive context of `river_means()`, we see `data` and `pattern` are defined:

```r
print(data)
```

```
#>           Missouri Mississippi Yukon Colorado Arkansas Columbia  Red Canadian
#> length        3768        3544  3190     2330     2322     2000 1811     1458
#> discharge     1956       18400  6340       40     1004     7730  852      174
```

```r
print(pattern)
```

```
#> [1] "R"
```

Pressing `n` executes `browser()`. Pressing `n` again executes

```r
desired_rivers <- grep(pattern, names(data))
```

Inspecting `desired_rivers`, we see

```r
print(desired_rivers)
```

```
#> [1] 7
```

This seems correct, so the error does not appear to be here. Let's press `n` to evaluate the next line

```r
new_data <- data[, desired_rivers]
```

Inspecting `new_data`, we see

```
new_data
```

```
#> [1] 1811  852
```

Well this seems weird – it does not look like a data frame. Let's inspect the structure

```
str(new_data)
```

```
#>  num [1:2] 1811 852
```

It is not a data frame, but rather a numeric vector. Because `rowMeans()` requires an array of two or more dimensions (e.g. matrix or data frame), the next line of code fails. Pressing `n` again returns the error and removes you from the interactive context

```
rowMeans(new_data)
```

```
#> Error in rowMeans(new_data): 'x' must be an array of at least two dimensions
```

A similar error occurs within `river_medians()` when running `apply()`

So what is happening here? We know the code works when the pattern yields at least two matches in `rivers`, and we know the code behaves oddly when the pattern only has one match. Looking at the documentation of `[` we see a `drop` argument, which coerces to the lowest possible dimension when equal to `TRUE`. When subsetting data frames, `drop` is `TRUE` by default. So when subsetting a data frame using a single column, the data frame structure is only preserved when `drop = FALSE`. Accommodating this change in `river_means()` and `river_medians()` yields functions whose bodies looks like

```
river_means <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers, drop = FALSE]
  rowMeans(new_data)
}

river_medians <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers, drop = FALSE]
  apply(new_data, 1, median)
}
```

Running these functions with `pattern = "R"` behaves as intended

```
river_means(rivers, "R")
```

```
#>    length discharge
#>      1811       852
```

```
river_medians(rivers, "R")
```

```
#>    length discharge
#>      1811       852
```

For more information about debugging in R, watch or read.

1.3.4 Another Error – No Mysterious Message

We received and fixed an error message that occurred when the pattern only matched one river. But what happens when it matches zero rivers?

```
river_means(rivers, "ZZZ")
river_medians(rivers, "ZZZ")
```

These types of bugs are especially pernicious because there is no error message associated with them. Whenever the output is unexpected, use `browser()` to diagnose the problem. In this context, `desired_rivers` is a length-zero vector, which causes problems in the remaining parts of the function. To guard against these types of bugs, program defensively and force the function to return an error message early.

```
river_means <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  if (length(desired_rivers) == 0) {
    stop("This is an error message that stops the function")
  }
  new_data <- data[, desired_rivers, drop = FALSE]
  rowMeans(new_data)
}
river_means(rivers, "ZZZ")
```

```
#> Error in river_means(rivers, "ZZZ"): This is an error message that stops the function
```

1.3.4.1 Exercises

1. Rewrite `river_medians()` so that it stops when no patterns are matched and returns an informative error message.

1.4 Additional Resources

- R Packages by Hadley Wickham and Jenny Bryan
- Writing an R package from scratch by Hilary Parker
- Writing R Extensions by CRAN (this resource is very technical)

Chapter 2

A Research Compendia

Placeholder

Chapter 3

Turning An R Package into a Reproducible Research Compendia

Chapter 4

Extensions

4.1 RMarkdown

4.2 GitHub

Chapter 5

Exercise Solutions

Q 1. What are some of your favorite R packages?

A 1. This is for you to answer! Though a few of my favorites include devtools, styler, rticles, rlang, and purrr.

Q 1. Of those ew have talked about so far, what benefits of R packages are most appealing to you?

A 1. This is also for you to answer. My answer is “all of them!”, though if I had to pick one, it is saving time by enforcing a structure – future me appreciates when past me does this.

Q 1. How many functions does `river_means()` call within the body of the function? What package are these functions in?

A 1. There are five functions: `<-`, `grep()`, `names()`, `[`, and `rowMeans()`. They are all in the “base” package, which can been seen in each function’s documentation

```
?`<-`
?grep
?names
?`[`
?rowMeans
```

Q 1. Play around with using these functions for some new patterns – do you notice anything strange?

A 1. If zero matches or one match occur for a pattern, there is some unexpected behavior. If there are zero matches, you get `NaN` for an answer (which stands for not a number).

```
river_means(rivers, "ZZZ")
```

2. If there is one match, you get a mysterious error – more on this later.

```r
river_means(rivers, "Red")
```

Q 1. Write a new function, `river_stat()`, that takes a data frame, pattern, and a general function by which to summarize river length and discharge. This general function should not be an actual function but rather a placeholder so that a user may insert their own function as an argument.

A 1.

```r
#' Summary statistics of river lengths and discharges
#'
#' @param data A data frame with two rows. The first row indicates river length and
#'   second row indicates river discharge. The columns of data indicate river names.
#' @param pattern A pattern by which to include only particular rivers
#' @param FUN A function to summarize the rivers
#'
#' @return The summarized river length and mean river discharge for the desired rivers
#' @export
#'
#' @examples
#' data("rivers")
#' river_stat(rivers, "Mi", min)
river_stat <- function(data, pattern, FUN) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  apply(new_data, 1, FUN)
}
river_stat(rivers, "Mi", min)
```

```
#>    length discharge
#>      3544      1956
```

```r
river_stat(rivers, "Mi", max)
```

```
#>    length discharge
#>      3768     18400
```

```r
river_stat(rivers, "Mi", mean)
```

```
#>    length discharge
#>      3656     10178
```

```r
river_stat(rivers, "Mi", stats::median)
```

```
#>    length discharge
#>      3656     10178
```

Q 1. Rewrite `river_stat()` so that it also takes additional arguments to the summarizing function (hint: use `...` as an argument)

A 1.

```r
#' Summary statistics of river lengths and discharges
#'
#' @param data A data frame with two rows. The first row indicates river length and
#'   second row indicates river discharge. The columns of data indicate river names.
#' @param pattern A pattern by which to include only particular rivers
#' @param FUN A function to summarize the rivers
#' @param ... Additional arguments to pass to \code{FUN}
#'
#' @return The summarized river length and mean river discharge for the desired rivers
#' @export
#'
#' @examples
#' data("rivers")
#' river_stat(rivers, "Mi|C", mean, trim = 0.5)
river_stat <- function(data, pattern, FUN, ...) {
  desired_rivers <- grep(pattern, names(data))
  new_data <- data[, desired_rivers]
  apply(new_data, 1, FUN, ...)
}
river_stat(rivers, "Mi|C", mean, trim = 0)
```

```
#>    length discharge
#>      2620      5660
```

```r
river_stat(rivers, "Mi|C", mean, trim = 0.5)
```

```
#>    length discharge
#>      2330      1956
```

Q 1. Write similar tests for `river_means()` (discharge), `river_medians()` (length), and `river_medians()` (length)

A 1.

```r
test_that("the mean discharge is calculated correctly in a test case", {

  # calculate values required for the test for discharge

  ## calculate the means from the function
  river_means_val <- river_means(rivers, "Missouri|Mississippi")
  river_means_discharge <- river_means_val[[2]]

  ## calculate the means "by hand"
  raw_vec_discharge <- unlist(rivers["discharge", c("Missouri", "Mississippi")])
  raw_means_discharge <- mean(raw_vec_discharge)

  # perform the actual test for discharge
  ## check that the function and "by hand" output matches
  expect_equal(river_means_discharge, raw_means_discharge)
})
```

```
#> Error in test_that("the mean discharge is calculated correctly in a test case", : co
```

```
test_that("the median length is calculated correctly in a test case", {

  # calculate values required for the test for length

  ## calculate the medians from the function
  river_medians_val <- river_medians(rivers, "Missouri|Mississippi")
  river_medians_length <- river_medians_val[[1]]

  ## calculate the medians "by hand"
  raw_vec_length <- unlist(rivers["length", c("Missouri", "Mississippi")])
  raw_medians_length <- median(raw_vec_length)

  # perform the actual test for length

  ## check that the function and "by hand" output matches
  expect_equal(river_medians_length, raw_medians_length)
})
```

```
#> Error in test_that("the median length is calculated correctly in a test case", : cou
```

```
test_that("the mean discharge is calculated correctly in a test case", {

  # calculate values required for the test for discharge

  ## calculate the medians from the function
  river_medians_val <- river_medians(rivers, "Missouri|Mississippi")
  river_medians_discharge <- river_medians_val[[2]]

  ## calculate the medians "by hand"
  raw_vec_discharge <- unlist(rivers["discharge", c("Missouri", "Mississippi")])
  raw_medians_discharge <- median(raw_vec_discharge)

  # perform the actual test for discharge

  ## check that the function and "by hand" output matches
  expect_equal(river_medians_discharge, raw_medians_discharge)
})
```

```
#> Error in test_that("the mean discharge is calculated correctly in a test case", : co
```

Q 1. Rewrite `river_medians()` so that it stops when no patterns are matched and returns an informative error message.

A 1.

```r
river_medians <- function(data, pattern) {
  desired_rivers <- grep(pattern, names(data))
  if (length(desired_rivers) == 0) {
    stop("The pattern provided does not match any rivers in the data provided")
  }
  new_data <- data[, desired_rivers, drop = FALSE]
  rowMeans(new_data)
}
river_medians(rivers, "ZZZ")
```

```
#> Error in river_medians(rivers, "ZZZ"): The pattern provided does not match any rivers in the d
```

The error messages in the `RPRW` package are more informative but more difficult to code.

Bibliography

Ben Marwick, Carl Boettiger, and Lincoln Mullen. Packaging data analytical work reproducibly using r (and friends). *The American Statistician*, 72(1): 80–88, 2018.