

Data Science for Public Policy Part II

Aaron R. Williams

2023-08-10

Table of contents

Preface	3
1 Advanced R Programming	4
1.1 Review	4
1.1.1 Relational Data	4
1.2 Programming	5
1.2.1 Selecting Data	5
1.2.2 Control Flow	10
1.3 Custom Functions	16
1.3.1 Motivation	16
1.3.2 Examples	16
1.3.3 Basics	18
1.3.4 Functions with Multiple Outputs	20
1.3.5 Referential Transparency	21
1.4 Debugging	22
1.5 Benchmarking	24
1.6 Organizing an Analysis	25
1.7 Packages	26
1.7.1 Use This	26
1.7.2 Package contents	26
1.7.3 Functions	27
1.8 Unit testing	27
1.9 Test coverage	28
1.9.1 Building the package	28
1.10 References	28
References	29

Preface

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

1 + 1

[1] 2

1 Advanced R Programming

We've mostly focused on the tidyverse and data analysis. This section will dive into advanced R programming topics like custom functions, debugging, and R packages.

1.1 Review

1.1.1 Relational Data

We've almost exclusively used data frames up to this point. We leveraged relations within our data to pick subsets of the data with functions like `filter()`.

```
library(tidyverse)

msleep |>
  filter(name == "Cow")

# A tibble: 1 x 11
   name genus vore order conservation sleep_total sleep_rem sleep_cycle awake
  <chr> <chr> <chr> <chr>    <chr>          <dbl>      <dbl>      <dbl> <dbl>
1 Cow   Bos   herbi Artiod~ domesticated      4        0.7        0.667    20
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Importantly, we almost never used indices or selected data by position, which can lead to errors if the underlying data change. For example, pick row number 5 and column number 4. This idea of using the relations in data reduces the chances of making mistakes and leads to clearer code.

1.2 Programming

1.2.1 Selecting Data

There are other ways to subset data, which are important when working with objects other than data frames. We will focus on `[]`, `[[]]`, and `$`.

1.2.1.1 Atomic Vectors

Much of our work focuses on four of the six types of [atomic vectors](#): logical, integer, double, and character. `[]` is useful for subsetting atomic vectors. Consider a vector with the first six letters of the alphabet:

```
letters_short <- letters[1:6]
```

We can use **positive integers** to subset to the first and fifth letters of the alphabet.

```
letters_short[c(1, 5)]
```

```
[1] "a" "e"
```

We can use **negative integers** to subset to the everything but the first and fifth letters of the alphabet.

```
letters_short[c(-1, -5)]
```

```
[1] "b" "c" "d" "f"
```

We can use **Booleans** (trues and falses) to subset to the first and fifth letters of the alphabet.

```
letters_short[c(TRUE, FALSE, FALSE, FALSE, TRUE, FALSE)]
```

```
[1] "a" "e"
```

This may seem silly, but we have many ways to create Booleans that we can then use to subset a vector.

```
booleans <- letters_short %in% c("a", "e")
```

```
booleans
```

```
[1] TRUE FALSE FALSE FALSE TRUE FALSE
```

```
letters_short[booleans]
```

```
[1] "a" "e"
```

We can use a **character vector** to subset a named vector.

```
named_vector <- c(a = 1, b = 2, c = 3)
```

```
named_vector
```

```
a b c  
1 2 3
```

```
named_vector[c("a", "c")]
```

```
a c  
1 3
```

We are able to select more than one element with `[]`, which will not be true of `[[]]` and `$`. One thing to look out for is vector recycling. Let's go back to `letters_short`, which is length six, but consider some indices of varying lengths.

```
letters_short[TRUE]
```

```
[1] "a" "b" "c" "d" "e" "f"
```

```
letters_short[c(TRUE, FALSE)]
```

```
[1] "a" "c" "e"
```

```
letters_short[c(TRUE, FALSE, TRUE)]
```

```
[1] "a" "c" "d" "f"
```

Caution

Wow, R recycles the Booleans. Six is divisible by 1, 2, and 3, so there are many ways to recycle the index to subset `letters_short`. This is dangerous and can quietly cause analytic errors.

Exercise 1

1. Create `letters_short`.
2. Try subsetting the vectors with indices with length four or five. What happens?

1.2.1.2 Lists

`[]` and `$` are useful for subsetting lists. Both can be used to subset data frames, but I recommend avoiding this.

Unlike `[]`, which returns multiple elements, `[]` and `$` can only return a single element and `[]` and `$` simplify objects by removing a layer of hierarchy.

`[]` can select an element by position or name, while `$` can only select an element by name. Consider a list with the first six letters of the alphabet.

```
alphabet <- list(  
  vowels = c("a", "e"),  
  consonants = c("b", "c", "d", "f")  
)
```

We can use `[]` to select the first or second element. In both cases, we get back a smaller list.

```
alphabet[1]
```

```
$vowels
```

```
[1] "a" "e"
```

```
class(alphabet[1])
```

```
[1] "list"
```

```
alphabet[2]
```

```
$consonants
```

```
[1] "b" "c" "d" "f"
```

```
class(alphabet[2])
```

```
[1] "list"
```

We can use `[[]]` to select the first or second element. Now, we get back a vector instead of a list. `[[]]` simplified the object by removing a level of hierarchy.

```
alphabet[[1]]
```

```
[1] "a" "e"
```

```
class(alphabet[[1]])
```

```
[1] "character"
```

We can also use `[[]]` to select an object by name.

```
alphabet[["vowels"]]
```

```
[1] "a" "e"
```

```
class(alphabet[["vowels"]])
```

```
[1] "character"
```

We can use `$` to select either vector by name.


```
alphabet$vowels
```

```
[1] "a" "e"
```

```
class(alphabet$vowels)
```

```
[1] "character"
```

Referring to objects by name should make for code that is more robust to changing data.

1.2.1.3 Before

```
alphabet1 <- list(  
  vowels = c("a", "e"),  
  consonants = c("b", "c", "d", "f")  
)
```

```
alphabet1[[2]]
```

```
[1] "b" "c" "d" "f"
```

```
alphabet1[["consonants"]]
```

```
[1] "b" "c" "d" "f"
```

1.2.1.4 After

```
alphabet2 <- list(  
  vowels = c("a", "e"),  
  confusing = "y",  
  consonants = c("b", "c", "d", "f")  
)
```


```
alphabet2[[2]]
```

```
[1] "y"
```

```
alphabet2[["consonants"]]
```

```
[1] "b" "c" "d" "f"
```

Subsetting lists can be difficult. Fortunately, RStudio has a tool than can help. Click on a list in your global environment. Navigate to the far right and click the list button with a green arrow. This will generate code and add it to the Console.

Name	Type	Value
 alphabet	list [2]	List of length 2
vowels	character [5]	'a' 'e' 'i' 'o' 'u'
consonants	character [21]	'b' 'c' 'd' 'f' 'g' 'h' ...

Interestingly, this tool avoids `$` and uses `[[]]` to pick the vector by name.

```
alphabet[["vowels"]]
```

```
[1] "a" "e"
```

1.2.2 Control Flow

1.2.2.1 For Loops

Loops are a fundamental programming tool for iteration; however, they are less common in R than in other programming languages. We previously focused on the Map-Reduce framework and `library(purrr)` instead of for loops for iteration.

For loops have two main pieces: 1. a header and 2. a body. Headers define the number of iterations and potential inputs to the iteration. Bodies are iterated once per iteration. Here is a very simple example:

```
for (i in 1:10) {  
  print(i)  
}
```

```
[1] 1  
[1] 2  
[1] 3  
[1] 4
```

```
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

We can use headers several different ways. Like above, we may just want to repeat the values in the index.

```
fruit <- c("apple", "banana", "canelope")
for (b in fruit) {

  print(b)

}
```

```
[1] "apple"
[1] "banana"
[1] "canelope"
```

We can use the header as an index.

```
fruit <- c("apple", "banana", "canelope")
for (i in 1:3) {

  print(fruit[i])

}
```

```
[1] "apple"
[1] "banana"
[1] "canelope"
```

We can leverage the index to use results from previous iterations.

```
result <- c(1, NA, NA, NA)
for (i in 2:4) {

  result[i] <- result[i - 1] * 2

}
```

```
}  
  
result
```

```
[1] 1 2 4 8
```

We've now seen three different ways to use the header.

1. Simply repeat the elements in the header (e.g. `print i`).
2. Use the elements in the header as an index (e.g. select the i^{th} element).
3. Use the header to reference past iterations (e.g. `i - 1`)

Don't grow vectors!

It is tempting to initialize a vector and then grow the vector with a for loop and `c()`. It is also tempting to initialize a data frame and then grow the data frame with `bind_rows()`. Because of R's design, this is computationally very inefficient. This is slow!:

```
vec <- c(1)  
  
for (i in 2:10) {  
  
  vec <- c(vec, i)  
  
}  
  
vec  
  
[1] 1 2 3 4 5 6 7 8 9 10
```

It is essential to pre-allocate vectors and then fill them in. It is also easy to make mistakes when creating indices (e.g. `1:length(x)` may end up as `c(1, 0)`). `seq_along()` is a helpful alternative to `:`. The following pre-allocates a vector and then uses the length of the vector to create an index.

```
numbers <- vector(mode = "numeric", length = 5)  
  
numbers
```

```
[1] 0 0 0 0 0
```

```
for (i in seq_along(numbers)) {  
  numbers[i] <- i  
}  
  
numbers
```

```
[1] 1 2 3 4 5
```

Let's consider a simple random walk with 100 steps. In this case, the person starts at location zero and random takes one step forward or one step back.

```
position <- vector(mode = "numeric", length = 100)  
  
set.seed(20230530)  
for (iter in 2:length(position)) {  
  position[iter] <- position[iter - 1] + sample(x = c(-1, 1), size = 1)  
}  
  
position
```

```
[1] 0 -1 0 1 2 1 0 1 0 1 2 1 0 1 2 1 2 1 2 1 2 1 2  
[26] 3 4 5 4 3 2 3 4 3 4 3 2 3 2 3 2 3 2 3 2 1 2 3  
[51] 2 3 4 3 4 5 6 5 6 5 6 7 6 5 6 7 8 9 10 11 12 13 12 13 12  
[76] 11 10 9 10 9 8 9 10 11 12 13 12 11 12 11 12 13 12 13 14 15 14 15 14 15
```

Exercise 2

1. Create the following list:

```
alphabet <- list(  
  vowels = c("a", "e"),  
  confusing = "y",  
  consonants = c("b", "c", "d", "f")  
)
```

2. Write a for loop and use `str_to_upper()` to transform all letters to upper case.

1.2.2.2 While Loops

While loops are similar to for loops; however, instead of predetermining the number of iterations in the header while loops determine a condition in the header and run until that condition is met. For loops can be rewritten as while loops. It can be useful to track the iteration number. Consider a simple example where we double `x` every iteration while `x < 1000`.

```
iteration <- 1  
x <- 2  
  
while (x < 1000) {  
  
  x <- x * 2  
  iteration <- iteration + 1  
  
}  
  
x
```

```
[1] 1024
```

```
iteration
```

```
[1] 10
```

Exercise 3

1. Write the random walk from the for loop above as a while loop. Stop the while loop when `position < -10` or `position > 10`. How many iterations did it take?

1.2.2.3 if, else, and else if

`if_else()` and `case_when()` apply conditional logic to a vector. We most frequently use those functions inside of `mutate()` to create a new variable or manipulate an existing variable.

R also has `if`, `else`, and `else if`, which are used to select sections of code to run. This is incredibly useful when programming outside of data manipulation. For example, we can use `if[\wedge]` to download a file only if it doesn't already exist.

```
if (!file.exists("data.csv")) {  
  
  download.file(url = "web-url.csv", destfile = "data.csv")  
  
}
```

Selection control flow has two important pieces. First, there is a conditional statement inside `()`. If the condition is `TRUE`, then evaluate. If it is `FALSE`, then don't evaluate. Second, there is a body contained in `{}`. Note the formatting in the above example.

The conditional statement must be a single `TRUE` or `FALSE`. If your statement involves more than one Boolean, then consider using `all()`, which evaluates to `TRUE` if everything is `TRUE` and `any()`, which evaluates to `TRUE` if any element is `TRUE`.

Let's consider a more sophisticated example.

```
if (distribution == "normal") {  
  
  x <- rnorm(n = 100)  
  
} else if (distribution == "poisson") {  
  
  x <- rpois(n = 100, lambda = 8)  
  
} else {  
  
  stop("distribution must be normal or poisson")  
  
}
```

This style of using `if`, `else if`, and `else` is fundamental for including options in custom functions.

1.3 Custom Functions

1.3.1 Motivation

Custom functions are an essential building block for good analyses. Custom functions are useful for abiding by the DRY (don't repeat yourself) principle. Under our conception of DRY, we should create a function any time we do something three times.

Copying-and-pasting is typically bad because it is easy to make mistakes and we typically want a single source source of truth in a script. Custom functions also promote modular code design and testing.

The bottom line: we want to write clear functions that do one and only one thing that are sufficiently tested so we are confident in their correctness.

1.3.2 Examples

Let's consider a couple of examples from (Barrientos et al. 2021). This paper is a large-scale simulation of formally private mechanisms, which relates to several future chapters of this book.

Division by zero, which returns `NaN`, can be a real pain when comparing confidential and noisy results when the confidential value is `zero`. This function simply returns 0 when the denominator is 0.

```
#' Safely divide number. When zero is in the denominator, return 0.
#'  
#' @param numerator A numeric value for the numerator  
#' @param denominator A numeric value for the denominator  
#'  
#' @return A numeric ratio  
#'  
safe_divide <- function(numerator, denominator) {  
  
  if (den == 0) {  
  
    return(0)  
  
  } else {  
  
    return(num / denom)  
  
  }  
}
```



```
}  
}
```

This function

1. Implements the laplace or double exponential distribution, which isn't included in base R.
2. Applies a technique called the laplace mechanism.

```
#' Apply the laplace mechanism  
#'  
#' @param eps Numeric epsilon privacy parameter  
#' @param gs Numeric global sensitivity for the statistics of interest  
#'  
#' @return  
#'  
lap_mech <- function(eps, gs) {  
  
  # Checking for proper values  
  if (any(eps <= 0)) {  
    stop("The eps must be positive.")  
  }  
  if (any(gs <= 0)) {  
    stop("The GS must be positive.")  
  }  
  
  # Calculating the scale  
  scale <- gs / eps  
  
  r <- runif(1)  
  
  if(r > 0.5) {  
    r2 <- 1 - r  
    x <- 0 - sign(r - 0.5) * scale * log(2 * r2)  
  } else {  
    x <- 0 - sign(r - 0.5) * scale * log(2 * r)  
  }  
  
  return(x)  
}
```

1.3.3 Basics

R has a robust system for creating custom functions. To create a custom function, use `function()`:

```
say_hello <- function() {  
  "hello"  
}  
  
say_hello()
```

```
[1] "hello"
```

Oftentimes, we want to pass parameters/arguments to our functions:

```
say_hello <- function(name) {  
  paste("hello,", name)  
}  
  
say_hello(name = "aaron")
```

```
[1] "hello, aaron"
```

We can also specify default values for parameters/arguments:

```
say_hello <- function(name = "aaron") {  
  paste("hello,", name)  
}  
  
say_hello()
```

```
[1] "hello, aaron"
```

```
say_hello(name = "alex")
```

```
[1] "hello, alex"
```

`say_hello()` just prints something to the console. More often, we want to perform a bunch of operations and then return some object like a vector or a data frame. By default, R will return the last unassigned object in a custom function. It isn't required, but it is good practice to wrap the object to return in `return()`.

It's also good practice to document functions. With your cursor inside of a function, go `Insert > Insert Roxygen Skeleton`:

```
#' Say hello
#'
#' @param name A character vector with names
#'
#' @return A character vector with greetings to name
#'
say_hello <- function(name = "aaron") {
  greeting <- paste("hello,", name)
  return(greeting)
}

say_hello()
```

```
[1] "hello, aaron"
```

As you can see from the [Roxygen Skeleton](#) template above, function documentation should contain the following:

- A description of what the function does
- A description of each function argument, including the class of the argument (e.g. string, integer, dataframe)
- A description of what the function returns, including the class of the object

Tips for writing functions:

- Function names should be short but effectively describe what the function does. They generally should be verbs while function arguments should be nouns. See the [Tidyverse style guide](#) for more details on function naming and style.
- As a general principle, functions should each do only one task. This makes it much easier to debug your code and reuse functions!
- Use `::` (e.g. `dplyr::filter()`) when writing custom functions. This will create stabler code and make it easier to develop R packages.

1.3.4 Functions with Multiple Outputs

When `return()` is reached in a function, `return()` is evaluated and evaluation ends and R leaves the function.

```
sow_return <- function() {  
  
  return("The function stops!")  
  
  return("This never happens!")  
  
}  
  
sow_return()
```

```
[1] "The function stops!"
```

If the end of a function is reached without calling `return()`, the value from the last evaluated expression is returned.

We prefer to include `return()` at the end of functions for clarity even though `return()` doesn't change the behavior of the function.

Sometimes we want to return more than one vector or data frame. `list()` is very helpful in these situations.

```
summarize_results <- function(x) {  
  
  mean_x <- mean(x)  
  
  median_x <- median(x)  
  
  results <- list(  
    mean = mean_x,
```

```

    median = median_x
  )

  return(results)
}

summarize_results(x = 1:10)

```

```

$mean
[1] 5.5

```

```

$median
[1] 5.5

```

1.3.5 Referential Transparency

R functions, like mathematical functions, should always return the exact same output for a given set of inputs¹. This is called referential transparency. R will not enforce this idea, so you must write good code.

1.3.5.1 Bad!

```

bad_function <- function(x) {

  x * y

}

y <- 2
bad_function(x = 2)

```

```

[1] 4

```

```

y <- 3
bad_function(x = 2)

```

¹This rule won't exactly hold if the function contains random or stochastic code. In those cases, the function should return the same output every time if the seed is set with `set.seed()`.

```
[1] 6
```

1.3.5.2 Good!

```
good_function <- function(x, y) {  
  x * y  
}  
  
y <- 2  
good_function(x = 2, y = 1)
```

```
[1] 2
```

```
y <- 3  
good_function(x = 2, y = 1)
```

```
[1] 2
```

Bruno Rodriguez has a [book](#) and a [blog](#) that explores this idea further.

1.4 Debugging

R code inside of custom functions can be tougher to troubleshoot than R code outside of custom functions. Fortunately, R has a powerful debugging tool.

The debugger requires putting custom functions in their own scripts. This is covered in Section [1.6](#).

To set up the debugger, simply select the red dot to the left of a line of code in a custom function and then source the custom function. After, there should be a red dot next to the defined function in the global environment.

Now, when the function is called it will stop at the red dot (the stop point). Importantly, the environment should reflect the environment inside of the function instead of the global environment.

Finally, RStudio gives several controls for the debugger. There is a button to Continue to the end of the function. There is a button to Stop execution.

```

5 my_function <- function(x) {
6
7   x <- x + 1
8
9   x <- x * 2
10
11  return(x)
12
13 }


```


(a)


Environment

History


Connections









Import Dataset



255 MiB



List



R

Global Environment

Values


y

3

Functions


bad_function


function (x)



good_function


function (x, y)





my_function

function (x)



(b)

Figure 1.1: Setting up the debugger

```

5 my_function <- function(x) {
6
7   x <- x + 1
8
9   x <- x * 2
10
11  return(x)
12
13 }

```

(a)

Environment

History

Connections

Import Dataset

261 MiB

List

R

my_function()

Values

x

2

Traceback

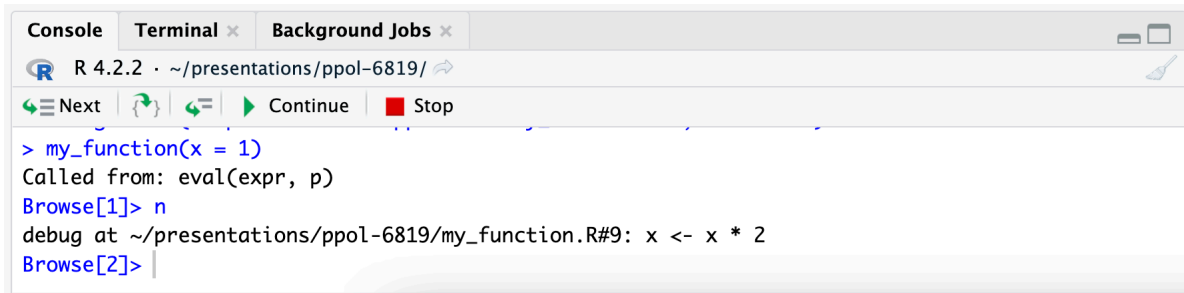
☐ Show internals

➔ my_function(x = 1) at my_function.R:9

(b)

Figure 1.2: Using the debugger

There is also a button with two brackets and a green arrow. This steps the debugger into another function. This is incredibly useful when functions are nested inside of functions.



The screenshot shows the R Studio interface with the 'Console' tab selected. The console displays the following text:

```
R 4.2.2 · ~/presentations/ppol-6819/
> my_function(x = 1)
Called from: eval(expr, p)
Browse[1]> n
debug at ~/presentations/ppol-6819/my_function.R#9: x <- x * 2
Browse[2]> |
```

Exercise 4

1. Create a custom function with at least three different pieces of R code.
2. Save the function in a .R script with the same name as the function.
3. Click the little red dot to the left of first line of code in the .R script.
4. Source the function with the source button at the top right.
5. Call the function. You should enter the debugger.

1.5 Benchmarking

Benchmarking is the process of estimating the run time of code. Oftentimes, benchmarking is used to compare multiple pieces of code to pick the more performant code. This raises a couple of issues:

1. Computing environments differ. My MacBook Pro with Apple M1 chips typically outperforms my work computer.
2. Other software can slow performance. When I open up Spotify my R processes typically slow down.

We can't solve problem 1 with an R package, but we can solve problem 2 by running tests multiple times. `library(microbenchmark)` makes this very easy.

Suppose we are interested in the median of a vector of 1 million numbers. We can easily calculate this with `median()` or `quantile()`. Suppose we are concerned about computation speed, so let's test the code performance:

```
library(microbenchmark)

x <- 1:1000000
```



```
microbenchmark::microbenchmark(
  median(x),
  quantile(x, probs = 0.5)
)
```

Unit: milliseconds

	expr	min	lq	mean	median	uq
	median(x)	8.515126	8.722792	9.507134	8.824188	9.710209
	quantile(x, probs = 0.5)	5.047042	5.180771	5.584579	5.237542	6.090333
	max neval cld					
44.226959	100	a				
7.635376	100	b				

Exercise 5

Let's compare `%>%` and `|>` to see if they have comparable computation times. Consider [this example](#) from Stack Overflow, which shows `|>` is clearly better.

1. Load `library(microbenchmark)` and add the `microbenchmark()` function.
2. Create `x1 <- 1:1000000`, `x2 <- 1:1000000`, and `x3 <- 1:1000000`
3. Test `median(x1)`, `x2 |> median()`, and `x3 %>% median()`.

1.6 Organizing an Analysis

We recommend writing functions for data analysis. We need a plan for how to add custom functions to our workflow built on RStudio projects and Quarto.

We typically recommend adding a directory called `R` or `src` in a project directory and then sourcing scripts in to Quarto documents. Keeping functions in separate scripts makes the functions easier to use in multiple documents and simplifies the debugging process outlined above.

We typically only add one function to an R script in the `R/` directory and name the script after the function (without parentheses). Next, we source function scripts at the top of Quarto documents after loading packages with the `source()`. [library\(here\)](#) is essential if when sourcing from a Quarto document that is in a subdirectory of the project.

1.7 Packages

At some point, the same scripts or data are used often enough or widely enough to justify moving from sourced R scripts to a full-blown R package. R packages make it easier to

1. Make it easier to share and version code.
2. Improve documentation of functions and data.
3. Make it easier to test code.
4. Often lead to fun hex stickers.

1.7.1 Use This

`library(usethis)` includes an R package template. The following will add all necessary files for an R package to a directory called `testpackage/` and open an RStudio package.

```
library(usethis)
create_package("/Users/adam/testpackage")
```

1.7.2 Package contents

The template includes a lot of different files and directories. We will focus on the minimum sufficient set of files for building a package.

`DESCRIPTION` contains the meta information about the package. Important lines include the package version and the license. Package versions are useful for tracking the version of the package used with an analysis. `library(usethis)` has a helper function for [picking a license](#).

```
Package: testpackage
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
  person("First", "Last", , "first.last@example.com", role = c("aut", "cre"),
    comment = c(ORCID = "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
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.2.3
```

1.7.3 Functions

Functions go in the R directory, just like when we sourced functions earlier. Be sure to reference packages directly with `::` (e.g. `stats::lm()`).

1.7.3.1 Roxygen

It is essential to use Roxygen skeletons with custom functions. RStudio makes this simple. Place the cursor in a function and select **Code > Insert Roxygen Skeleton**. The Roxygen skeleton creates help documentation for a package, which can be accessed with `?`.

```
#' Title
#'\n#' @param\n#'\n#' @return\n#' @export\n#'\n#' @examples
```

The title should be a brief description of the function. `param` describes each input to the function and `return` describes the output of the function.

1.7.3.2 Tests

1.8 Unit testing

Unit testing is the systematic testing of functions to ensure correctness.

Unit testing is essential to developing high-quality code—especially for large scale projects. Fortunately, `library(testthat)` and `library(usethis)` make unit testing easy in R packages.

`use_testthat()` will add a directory with the skeleton for a set of unit tests.

`use_test()` will add a new test. For example, `use_test("good_function")` will add a script to test the `good_function()` from earlier in the notes.

There are two pieces to a test. A **test** and an **expectation**. Tests group multiple expectations together and begins with `test_that()`. Expectations compare the output of the function against expected output. The script created by `use_test("good_function")` contains a shell that is easy to update. Consider this example:

```
test_that("good_function is correct for positive integers ", {  
  expect_equal(good_function(x = 2, y = 2), 4)  
  expect_equal(good_function(x = 3, y = 3), 9)  
})
```

1.9 Test coverage

Test coverage is the scope and quality of tests performed on a code base.

The goal is to develop tests with good test coverage that will loudly fail when bugs are introduced into code.

1.9.1 Building the package

The package needs to be documented and rebuilt each time the contents of the package are edited. The package should also be tested and checked on a regular basis.

1. Run `devtools::document()` to document all of the functions in the package.
2. Run `devtools::build()` to build the package.
3. Run `devtools::test()` to test the package. This runs all tests and returns diagnostics about passes, warnings, and failures.
4. Run `devtools::check()` to check the package. This runs tests and also checks for documentation and package setup.

This section barely scratches the surface of package development. Read [R Packages \(2e\)](#) to learn more.

1.10 References

References

Barrientos, Andrés F., Aaron R. Williams, Joshua Snoke, and Claire McKay Bowen. 2021. “A Feasibility Study of Differentially Private Summary Statistics and Regression Analyses with Evaluations on Administrative and Survey Data.” <https://doi.org/10.48550/ARXIV.2110.12055>.