- Functional programming and unit testing for data munging
-
- ****1** Why this book?
  - ****1.1** Motivation
  - ****1.2** Who am I?
  - ****1.3** Thanks
  - ****1.4** License
- ****2** Introduction
  - ****2.1** Getting R
  - ****2.2** A short overview of functional programming
  - ****2.3** A short overview of unit testing
- ****3** Functional Programming
  - ****3.1** Introduction
    - ****3.1.1** Function definitions
    - ****3.1.2** Properties of functions
  - ****3.2** Mapping and Reducing: the *base* way
    - ****3.2.1** Mapping with `Map()` and the `*apply()` family of functions
    - ****3.2.2** `Reduce()`
  - ****3.3** Mapping and Reducing: the `purrr` way
    - ****3.3.1** The `map*()` family of functions
    - ****3.3.2** Reducing with `purrr`
    - ****3.3.3** Other useful functions from `purrr`
  - ****3.4** Anonymous functions
  - ****3.5** Wrap-up
  - ****3.6** Exercises
- ****4** Unit testing
  - ****4.1** Introduction
  - ****4.2** Unit testing with the `testthat` package
  - ****4.3** Actually running your tests
  - ****4.4** Wrap-up
  - ****4.5** Exercises
- ****5** Packages
  - ****5.1** Why you need your own packages in your life
  - ****5.2** R packages: the basics
  - ****5.3** Writing documentation for your functions
  - ****5.4** Unit test your package
  - ****5.5** Checking the coverage of your unit tests with `covr`
  - ****5.6** Wrap-up
- ****6** Putting it all together: writing a package to work on data

**Functional programming and unit testing for data munging with R

Chapter 2 Introduction

2.1 Getting R

Since I'm assuming you have an intermediate level in R, you already should have R and Rstudio installed on your machine. However, you may lack some of the following packages that are needed to follow the examples in this book:

- covr: to check the coverage of your unit tests
- dplyr: to clean, transform, prepare data
- lazyeval: for lazy evaluation
- lubridate: makes working with dates easier
- memoise: makes your function remember intermediate results
- purrr: extends R's functional programming capabilities
- readr: provides alternative functions to `read.csv()` and such
- roxygen2: creates documentation files from comments
- stringr: makes working with characters easier
- testthat: the library we are going to use for unit testing
- tibble: provides a nice, cleaner alternative to `data.frame`
- tidyr: works hand in hand with `dplyr`

If you're missing some or all of these packages, install them. You'll notice that most, if not all, of these packages were authored or co-authored by Hadley Wickham, currently chief scientist at Rstudio, so you can install most of these packages by installing a single package called `tidyverse`:

`sourceCode r install.packages("tidyverse")`

The `tidyverse` package installs some other useful packages that we will not use, but you should check them out anyways!

2.2 A short overview of functional programming

What is functional programming? Wikipedia tells us the following:

> In computer science, functional programming is a programming paradigm —a style of building the structure and elements of computer programs— that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

That's the first paragraph of the Wikipedia page and it's quite heavy already!

So let's try to decrypt what is said in this paragraph. Functional programming is a programming paradigm. You may have heard of object oriented programming, or imperative programming before. You actually probably program in an imperative way without knowing it. Imperative programming is usually how programming is taught at universities, and most people then keep on programming in this way, especially in applied sciences like applied econometrics. Usually, people that write code in an imperative way tend to write very long scripts that change the state of the program gradually. In the case of a statistician (I will use the word 'statistician' to mean any person that works with datasets. Be it an economist, biologist, data scientist, etc.) this usually means loading a dataset, doing whatever has to be done by writing each instruction in a file, then running everything. Sometimes this statistician has to save temporary datasets, and then write other scripts that do a series of computations on these temporary datasets and then not forget to delete said temporary datasets. Functional programming is different, in that you write functions that do one single task and then call these functions successively on your data set. These functions can be used for any other project, can be easily documented and tested (more on this below). Because each function performs a single task and is well documented, it is also easier to understand what the program is supposed to do. Comments in a thousand-lines file are actually not that much useful. The file is so long, that even when commented you simply cannot make any sense of what is going on. It is also easier to automate tasks and navigate through the code. Since one function does one single task, if you're looking for the line of code that creates variable `X`, just look in the function called `create_var_X()`, instead of `CTRL-F`ing around. 1000 lines long script. You can also be sure that your functions do not do anything else (basically, this is what is meant by "eliminating side effects") than the single task you gave them. You can *trust your functions*.

2.3 A short overview of unit testing

At the end of the last section I wrote that you can *trust your functions*. Is that true though? Functional programming can make your life easier, but it does not prevent you from introducing bugs in your code. However, what functional programming makes easily possible, is to very easily and effectively test your code thanks to unit testing. You probably already test your code, by hand. You write some loop that is supposed to sum the first 10 integers and then you try it out and check if, indeed, your loop returns 55. Because this is the correct result, you save your work and continue programming something else, and so on. Unit testing is this, but in an automated way. Instead of just trying things out in the interpreter, you write unit tests. You write code that actually checks your functions. You save this unit tests somewhere, and then re-run them whenever you make changes to your code. Even if you don't change some parts of your code, you re-run every unit test. Because you actually never know what may happen. Maybe changing a single line in one of your functions introduced some unforeseen consequences that breaks functionality some place else. When you change code, and *all* your unit tests still pass, then you can be confident that your code is correct (actually, don't be too confident, because maybe you didn't write enough unit tests to cover every case. But we will see how we can be sure there is enough *coverage*).

** **