ndexr: A Hanbook on R

Freddy Ray Drennan

2021-11-17

Contents

Chapter 1

Prerequisites

Book Outline

- Install R
- Install R Studio
- Windows Only: Install RTools
  - When installed, run in the RStudio Console: `write('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"', file = "~/.Renviron", append = TRUE)`
- Windows Only: Install WSL2
  - Computer should be completely updated before install.
- Install Git
- Create Github Account
- Fork r-handbook
- Install Docker and Docker Compose
- Create AWS Account
  - Billing will be discussed in the course, but don't expect to pay much - i.e., 10-20 dollars a month for high course activity.
  - Remember to `stop` EC2 servers when we begin using them. AWS is polite about your first few refund requests.
- Create Reddit Account
  - Follow Instructions here

Make sure you install the `tidyverse` packages. Update to renv later.

```
install.packages('tidyverse')
```

Chapter 2

Literature

Here is a review of existing methods.

Chapter 3

What is R

3.1 Types of Problems You Can Solve

3.2 Base R, Tidyverse, data.table

3.3 Arguments/ Developments within the language

3.4 What are Variables

3.4.1 Valid Variable Names

Chapter 4

Building Blocks

4.1 Vectors

Vectors are containers information of similar type. You can think of them as having $1 * n$ cells where n is *any* positive integer, and make up the rows and columns of tables. Vectors have a few components that make them special. First, they always contain the same type of value. R has many different data types, but the most common are **numeric**, **character**, and **logical** (**TRUE**/**FALSE**).

Rule 1: Vectors only contain one type of data.
Rule 2: Vectors are always $1xn$ **dimensional,** $1xn = n$ where n is the `length` of the vector.
Rule 3: Vectors don't always have names, but should if it makes sense.

4.2 Functions

Functions are containers where anything or nothing can happen, but whatever happens, it happens the same way every single time. They allow for generalization of complicated ideas and routines that we wish to repeat over and over again. A function may have an input, but no output. It may have an output, but no input, both or none. If it's something you need to do repeatedly, or containing code makes your program easier to read, then write a function for that process.

Rule 4: Functions have inputs, outputs, and a body. A function can have multiple outputs, but given a particular set of inputs, the solution should never change assuming you are not developing a function with randomness built in.

R has a built-in constant called `letters`. This means that no matter where you are writing R, `letters` will be available to you. We see that `letters` is a **character vector** in our program below, and use the composition of functions to create a program that describes `letters`.

```
print(letters)
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

Next, we can use some functions which take in pretty much any object that exists in R and spits back information regarding the `letters` data.

```
main <- function() {
  print_information <- function(x) {

    variable_name = deparse1(substitute(x))

    length_x = length(x)
    typeof_x <- typeof(x)
    is_vec_x <- is.vector(x)

    meta_list <- list(
      length = length_x,
      type = typeof_x,
      is_vector = is_vec_x
    )

    cli::cli_alert('Information about {variable_name}')

    cli::cli_alert_info("{variable_name} is a 1x{length_x} dimensional")
    cli::cli_alert_info("")

    purrr::iwalk(meta_list, function(x, index) {
      cli::cli_alert_info(glue::glue('{index} {x} is type {typeof(x)}'))
    })

    return(meta_list)
  }

  cli::cli_alert_info('Execute print_information')
  output <- print_information(mtcars)
  cli::cli_alert_success('Execute print_information complete')

  print(output)
```

```
}

main()
```

```
## i Execute print_information

## > Information about mtcars

## i mtcars is a 1x11 dimensional

## i

## i length 11 is type integer

## i type list is type character

## i is_vector FALSE is type logical

## v Execute print_information complete

## $length
## [1] 11
##
## $type
## [1] "list"
##
## $is_vector
## [1] FALSE
```

Chapter 5

Debugging

5.1 What is the debugger?

5.2 How to learn R without knowing any R

5.3 `browser()`

5.4 `debug` and `undebug`

5.5 `debugonce`

5.6 Understanding debugging output

Chapter 6

Vectors

6.1 `c`

6.2 `[` and `[[`

- Vectors
  - atomic
- Strings
  - Base R
  - `stringr`
    - Regular Expressions
  - Cheat Sheet
- Numbers
  - Integer
  - Double
- Factors
  - `as.factor` vs. `as_factor`
- Dates
  - Base R
  - `lubridate`

Chapter 7

Lists

7.1 `list`

7.2 `[` and `[[`

- Lists
  - `list()` and `c`
  - `[` and `[[`
  - Connection between lists and json
    - `jsonlite`

Chapter 8

Vectors

8.1 `c`

8.2 `[` and `[[`

- Tables
  - matrices
  - `data.frame` vs `tibble`
  - data.frames are lists with equal length, atomic vectors

Chapter 9

Functional Programming

2. Functions
    - Sequences
    - Mapping functions
    - pipes
    - void
    - `return`
        - Can a function return nothing?
        - What are side effects?
        - Multiple return statements

9.1 Base R

`apply`, `lapply`, `mapply`

9.2 Modern R

`purrr` * `map_*` * `map2_*` * `pmap_*` * Iterate over What? * Why are data.frames mapped over columnwise? * A: data.frames are lists, and mapping functions will iterate over each individual item in a list

Chapter 10

Tidy Data

- Concept of tidy data
  - Tidy Data Paper
- `tidyr`
  - `pivot_longer`
  - `pivot_wider`

Chapter 11

dplyr

- `dplyr` and data manipulation
  - main functions
    - `select`
    - `mutate`
    - `filter`
    - `transmute`
  - summarizing data
    - `group_by`
    - `summarize` - one row per group
    - `mutate` - one or many rows per group will have same value
    - `ungroup` - remove grouping
      - Not everything has to be a `group_by`
      - Solving group problems with vectors

- Joining Tables
  - `inner_join`
  - `full_join`
  - `left_join` / `right_join`