A Handbook on R

Freddy Ray Drennan

2021-11-22

Introduction

A Dose of R

Let's solve a problem using R. Suppose we have a friend that is interested in the current trend regarding COVID-19 cases. The first thing we will probably do is try to figure out an efficient and reliable way for importing Covid-19 data into our R session. Conveniently, the `COVID19` package allows us to pull the latest data without any hard work and consists of one function - `covid19`.

A function does stuff given a set of inputs. Remember seeing equations like $y = 2x + 3$ in algebra? We said given $x = 2$ then $y = 7$. because $2(2) + 3 = 7$. In calculus, equations became functions with a slightly different syntax. $f(x) = 2x + 3$. What happened to the y? Well, nothing really. We can say $y = f(x) = x^2$ or $result = f(x)$ or $z = g(x, y) = x + min(x, y)$. Once we have these definitions, we can go further and compose them together. For example, $f(g(x, y)) = (x + min(x, y))^2$. Now, the outputs of functions become the inputs for other functions.

```
library(cli)

# Same as f(x) = 2x + 3
f <-  function(x) {
  x <- x * x
  x
}

g <- function(x=NULL, y=NULL) {
  result <- x + min(x, y)
  result
}

print(f(g(3, 4)))
```

```
## [1] 36
```

We now have a way of describing inputs and output a little more clearly. Instead of writing, $(3 + min(3, 4)) * (3 + min(3, 4))$ we can write $f(g(3, 4))$ or try new creations like $z(x, y) = f(g(f(x), f(y)))$ so $z(1, 2) = f(g(f(1), f(2))) = f(g(1, 4)) = f(2) = 4$.

Now just take this idea about functions and expand your definition of inputs and outputs to be any number, none or many, and of any type that R supports - character, numeric, date/time, data.frame or list - all of which we'll cover.

In your RStudio console, you can write the following to install the `COVID19` package using the `install.packages` function. If you are interested in learning more about this function, you can write `?install.packages` in your console and the documentation for the function will appear.

```
# For help menu, uncomment next line
# ?install.packages

# If the package is not yet installed, you can install it by passing
# a string with the package name to the `install.packages` function
install.packages(pkgs = c('COVID19'))
```

For a full list of what packages are available through the `install.packages` function, please check out the Contributed Packages page at CRAN or scrape it yourself.

```
library(rvest)
cran_packages <- 'https://cran.r-project.org/web/packages/available_packages_by_date.ht
html_table(html_element(read_html(cran_packages), 'table'))
```

```
## # A tibble: 18,468 x 3
##    Date       Package      Title
##    <chr>      <chr>        <chr>
##  1 2021-11-23 dataCompareR Compare Two Data Frames and Summarise the Difference
##  2 2021-11-23 DescTools    Tools for Descriptive Statistics
##  3 2021-11-23 makepipe     Pipeline Tools Inspired by 'GNU Make'
##  4 2021-11-22 abtest       Bayesian A/B Testing
##  5 2021-11-22 Bestie       Bayesian Estimation of Intervention Effects
##  6 2021-11-22 brms         Bayesian Regression Models using 'Stan'
##  7 2021-11-22 cgrcusum     Continuous Time Generalized Rapid Response CUSUM
##  8 2021-11-22 clustra      Clustering Trajectories Anchored at Intervention Time
##  9 2021-11-22 cpsR         Load CPS Microdata into R Using the 'Census Bureau D~
## 10 2021-11-22 crul         HTTP Client
## # ... with 18,458 more rows
```

- *WRITE ABOUT HELP MENU?*

I'm a man of few words. So let's get to it. The two lines of code below consist of three functions. The first two are `library` and `covid19`, but the third is hidden, and I will disclose where shortly. *Functions* are spaces for stuff to happen. Functions help us make common procedures repeatable. By creating a function with a particular name and inputs, we can get some sort of useful (or not useful, the world's your oyster) output.

In this case, `library` loads *packages* from a folder in the R environment called library. You can see which ones your R environment knows about by running the function `.libPaths()`. `covid19` is a function from the `COVID19` package, and would only be available after executing `library(COVID19)` or if `library(COVID19)` is omitted, by pulling it from the package namespace directly by preceding the function with the package name and two colons like so: `COVID19::covid19`. Generally speaking, you simply use `library` because it reduces the amount of text on the page.

```
library(COVID19)
```

```
## Warning: package 'COVID19' was built under R version 4.1.2
```

```
covid_data <- covid19(
    country = 'United States',
    start = '2021-01-01',
    end = "2021-11-21",
    verbose = FALSE
)
```

How do I know if a function is vectorized

Initial Setup

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')
```

What is R

Types of Problems You Can Solve

Base R, Tidyverse, data.table

Arguments/ Developments within the language

What are Variables

Valid Variable Names

Building Blocks

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 always contain the same type of value. R has many different types of vectors, but the most common are **numeric**, **character**, and **logical** (**TRUE**/**FALSE**).

Vectors are cool. I like to think of them as boxes that can only be stacked on top of one another.

```
typeof(c(TRUE))
```

```
## [1] "logical"
```

```
typeof(c(TRUE, 1))
```

```
## [1] "double"
```

```
typeof(c(TRUE, 1, 'a'))
```

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

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
```

Debugging

What is the debugger?

How to learn R without knowing any R

`browser()`

`next`, `continue`

`debug` and `undebug`

`debugonce`

Understanding debugging output

LOTS OF DEBUGGING EXERCISES CANNOT STRESS ENOUGH

Vectors

`c`

`[` and `[[`

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

Lists

`list`

`[` and `[[`

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

Tables

`c`

`[` and `[[`

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

Functional Programming

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

Base R

`apply`, `lapply`, `mapply`

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

Tidy Data

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

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`

Project Outline

To be expanded over many chapters

1. Windows vs Mac vs Linux
2. Docker Installation
    - Windows needs to set up VM in bios
3. RStudio IDE
    - Cheat Sheet
4. reddit api creds
5. reticulate
    - Enough R to know Python
        - Type Conversions
    - miniconda installation
    - virtual environments
6. Package Structure
    - Defaults for RStudio
        - Rebuild and Restart with Roxygen2
    - `.env`
    - `.gitignore`
    - `.Rprofile`
    - `.Renviron`
    - Packages necessary for efficient development
        - usethis
        - roxygen
        - devtools
            * Cheat Sheet
    - Make and Makefiles
        - Automating Package Build
    - Unit Testing (probably bad location for ut, no code written)

    - testthat

7. Git
    - Github
    - git circle, workflow
8. Retrieving Data from API
    - `praw`
    - `dotenv` and `.env`
    - Old Reddit Code to start with
9. Docker and Docker Compose Introduction
    - `.dockerignore`
10. Create Postgres Database
    - What are Ports?
    - Postgres Credentials
11. Create functions for Storing Reddit Data
12. *Need preferred method for streaming data, i.e., Airflow not a good scheduler for scripts that are always running* and need a kickstart on failure, timeout, etc. Docker with `restart: always` may be sufficient
13. Plumber API
    - Add to docker-compose
    - Functions for ETL, Shiny Application
14. ETL with Airflow and HTTP operator connected to Plumber API
15. Shiny
    - Reactive Graph
        - Order does not matter, the graph does
    - Why Modules?
        - `map` over modules
16. Automating Infrastructure
    - `awscli`
    - `boto3`
        - `biggr`
    - Create EC2 Server from R
        - User Data