ndexr: A Hanbook on R

Freddy Ray Drennan

2021-11-15

Contents

Chapter 1

Prerequisites

- 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.

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

Chapter 2

Functions

2.1 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.

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.

```
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')
```

```
## i Execute print_information
```

```
output <- print_information(mtcars)
```

```
## > 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
```

```
cli::cli_alert_success('Execute print_information complete')
```

```
## v Execute print_information complete
```

```
print(output)
```

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

We can take this information and put it in a `list`. A list holds whatever we want to put inside it. The left side is the `name` that the object will get within the list, the right side the object itself.

2.2 Indexes

Consider the values $1, 2, ...n$, where n is any positive integer. Indexing is just our way of specifying the location of something. An index in R always starts at `1` and extends through the integers until `n` equals the number of items we are counting.

```
seq(from = 1, to = 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

What is special about starting the sequence at `1` is that it creates a vector of integers that also match their location in the vector. When you think about vectors, remember that whether they be logical (`TRUE`/`FALSE`), numeric (`1.1`, `2`), integer (`1`, `4`), or character (`"a"`, `"cat"`), there is *always* an integer associated with it's position in the object.

```
message('Full letters vector')
```

```
## Full letters vector
```

```
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"
```

```
message('Selecting the first 10 values')
```

```
## Selecting the first 10 values
```

```
letters[seq(from = 1, to = 10)]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

2.3 What are Functions?

We can assign this **integer vector** to a **variable** called `data` . When we assign something to a variable, we can take it with us to use in other parts of our program. We will use a built in constant that R provides called `LETTERS` which is a **character vector** containing the 26 letters of the alphabet capitalized.

```
data = seq(1, 10)
data <- setNames(data, LETTERS[data])
print(data)
```

```
##  A  B  C  D  E  F  G  H  I  J
##  1  2  3  4  5  6  7  8  9 10
```

Maybe we are interested in how long the vector is, or what its names are.

```
length(data)
```

```
## [1] 10
```

```
class(data)
```

```
## [1] "integer"
```

```
names(data)
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

2.4 Sequences

Suppose we have the function $g(x) = x^2$, and want to use it. We can define the function below, and call it `square`. The function can be named whatever we like, `square` could be `platypus` for all I care. It doesn't really matter, except to whoever is reading your program - likely you at a later, more forgetful date. Generally, we want to name the things we create something useful, so that when we are reading our program, we understand what is happening. Naming functions is a

```
square = function(x) x^2
square(3) #
```

```
## [1] 9
```

Chapter 3

Literature

Here is a review of existing methods.

Chapter 4

Methods

We describe our methods in this chapter.

Chapter 5

Applications

Some *significant* applications are demonstrated in this chapter.

5.1 Example one

5.2 Example two

Chapter 6

Final Words

We have finished a nice book.