An Introduction to Base R & the Tidyverse

Programming with Base R

Objects in R

Everything you do in R will involve some kind of **object** that you have created. Think of an **object** like a box that you can place data in, so that R can later access and manipulate the data. An important of the code below is the assignment operator `<-` which is how R knows to assign `value` to `object_name`.

```
1 object_name <- value
```

Atomic Vectors

- An atomic vector is just a simple vector of data.
- R recognizes six types of atomic vectors:
  - Integers
  - Doubles (Numeric)
  - Characters
  - Logicals
  - Complex
  - Raw

Integer & Numeric Vectors

Integer vectors contain only integers. Add L after each number so R recognizes it as an integer. **Numeric (doubles) vectors** contain real numbers. These are the default vectors for numbers.

```
integer_vec <- c(1L, 2L, 50L)
numeric_vec <- c(1, 2, 50, 45.23)
```

Character Vector

Character vectors contain only text data also referred to as string data. Basically anything surrounded by `""` or `''` is considered string data.

```
character_vec <- c("1", "abc", "$#2")
```

Logical Vector

Logical vectors are vectors that can only contain TRUE or FALSE values also referred to as boolean values.

```r
logical_vec <- c(TRUE, FALSE)
```

Adding Attributes

You can think of attributes as metadata for R objects. As a user you will not need to worry too much about attributes directly, but attributes tell R how to interact with the specific object and allow the user to store information that is secondary to the analyses they are conducting.

names Attribute

```r
days_of_week <- 1:7
names(days_of_week) <- c("mon", "tues", "wed", "thurs", "fri", "sat", "sun")
names(days_of_week)
```

```
[1] "mon"   "tues"  "wed"   "thurs" "fri"   "sat"   "sun"
```

```r
attributes(days_of_week)
```

```
$names
[1] "mon"   "tues"  "wed"   "thurs" "fri"   "sat"   "sun"
```

dim Attribute

```
days_of_week <- 1:14
dim(days_of_week) <- c(2, 7) # 2 Rows, 7 Columns
attributes(days_of_week)
```

```
$dim
[1] 2 7
```

```
class(days_of_week)
```

```
[1] "matrix" "array"
```

Creating Factors

R stores categorical data using factors, which are integer vectors with two attributes: `class` and `levels`.

```
1 days_of_week <- factor(c("mon", "tues", "wed", "thurs", "fri", "sat", "sun"))
2 typeof(days_of_week)
```

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

```
1 attributes(days_of_week)
```

```
$levels
[1] "fri"   "mon"   "sat"   "sun"   "thurs" "tues"  "wed"

$class
[1] "factor"
```

Data Frames: Best way to Represent Data

Data frames are the best way to structure and store data in R. Data frames are sort of the R equivalent of an excel spreadsheet.

Each column in a data frame is a vector, so a data frame can combine a numeric vector as one column with a character vector as another column.

```
1 data_frame_1 <- data.frame(NUMERIC = c(1, 3), CHARACTER = c("a", "b"),
2                            LOGICAL = c(TRUE, FALSE))
3 data_frame_1
```

```
  NUMERIC CHARACTER LOGICAL
1       1         a    TRUE
2       3         b   FALSE
```

Viewing Your Data

You can use `View()` to open up a spreadsheet-like view of your data.

```
View(data_frame_1)
```

Selecting Data from Data Frames

You will mainly select data from data frames using one of the two following methods:

```
data_frame_1[1, 1] # Index the row and/or column
```

```
[1] 1
```

```
data_frame_1[, 1] # Leaving the column or row index blank selects the whole vector
```

```
[1] 1 3
```

```
data_frame_1$NUMERIC # Use a $ operator to reference the column name
```

```
[1] 1 3
```

Functions in R

Functions are objects in R that take user inputs, apply some predefined set of operations, and return an expected output.

```
1  sum(c(1, 3))
```

```
[1] 4
```

The Elements of a Function

R comes with a variety of predefined functions and they all follow the same structure:

- A **name** for the function.
- The **arguments** that change across different function calls.
- The **body** which contains the code that is repeated across different calls.

The Elements of a Function

```
name <- function(argument) {
  body
}
```

Example Base Function

```r
x <- c(1, 4, 6)
sum(x)
```

```
[1] 11
```

```r
mean(x)
```

```
[1] 3.666667
```

```r
min(x)
```

```
[1] 1
```

Linking Functions Together

R lets you link any number of functions together by nesting them. R will start with the innermost function and then work its way outward.

```
1  sum(abs(c(-1, -1, 1, 1)))
```

```
[1] 4
```

Using the pipe |>

The |> operator allows you to take the output of one function and feed it directly into the first argument of the next function. Using the |> makes it easier to read your code, which is a good thing.

```
c(-1, -1, 1, 1) |>
  abs() |>
  sum()
```

```
[1] 4
```

Packages: The Lifeblood of R

A lot of what makes R such an effective programming language (especially for statistics) is the sheer number of available R packages. An R package is a collection of functions that complement one another for a given task. New packages are always being developed and anyone can author one!

Installing & Loading Packages

You can use `install.packages` to install a package once and then `library` to load that package and gain access to all of its functions.

```
install.packages("package_name")
library(package_name)
```

Reading and Writing Data

There are a number of different methods to read and write data into R. The two most common functions are:

```
data <- read.csv("filepath/file-name.csv")

write.csv(data, "filepath/file-name.csv")
```

Importing Data from an R Package

Oftentimes, R packages will come with their own datasets that we can load into R. The `peopleanalytics` package has many such datasets that we will use today:

```
1 data_employees <- peopleanalytics::employees
```

Getting Help with R

There are two ways to get help in R:

- Add `?` in front of your function, which will result in RStudio displaying the help page for that function.
- Google what you are trying to do. More often than not, someone else has run into your problem, found a solution, and posted it. Stand on their shoulders!

```
1  ?sum()
```

Introduction to the Tidyverse

What is the Tidyverse?

The tidyverse is a collection of R packages that “share a common philosophy of data and R programming and are designed to work together.”

Installing Packages from the Tidyverse

```
1  install.packages("tidyverse")
```

`tibble`: Data frame of Tidyverse

Tibbles are the tidyverse's version of a `data.frame`. They can be loaded from the tidyverse package: `tibble`.

```
1 data_employees_tbl <- tibble::as_tibble(data_employees)
2 data_employees_tbl
```

```
# A tibble: 1,470 × 36
   employee_id active stock_opt_lvl trainings   age commute_dist ed_lvl ed_field
         <int> <chr>          <int>     <int> <int>        <int>  <int> <chr>
 1        1001 No                 0         0    41            1      2 Life Sc…
 2        1002 Yes                1         3    49            8      1 Life Sc…
 3        1003 No                 0         3    37            2      2 Other
 4        1004 Yes                0         3    33            3      4 Life Sc…
 5        1005 Yes                1         3    27            2      1 Medical
 6        1006 Yes                0         2    32            2      2 Life Sc…
 7        1007 Yes                3         3    59            3      3 Medical
 8        1008 Yes                1         2    30           24      1 Life Sc…
 9        1009 Yes                0         2    38           23      3 Life Sc…
10        1010 Yes                2         3    36           27      3 Medical
# ℹ 1,460 more rows
# ℹ 28 more variables: gender <chr>, marital_sts <chr>, dept <chr>,
#   engagement <int>, job_lvl <int>, job_title <chr>, overtime <chr>,
#   business_travel <chr>, hourly_rate <int>, daily_comp <int>,
#   monthly_comp <int>, annual_comp <int>, ytd_leads <int>, ytd_sales <int>
```

`dplyr`: Your Data Multitool

The package `dplyr` should become your go-to data manipulation and structuring tool! It contains many useful functions that make it surprisingly easy to manipulate and structure your data.

The Philosophy of `dplyr` Functions

Every function in `dplyr` follows this philosophy:

- First argument is always a data frame.
- Remaining arguments are usually names of columns on which to operate.
- The output is always a new data frame (tibble).

`dplyr` functions are also further grouped by whether they operate on **rows**, **columns**, **groups**, or **tables**.

Using `dplyr` to Operate on Rows

The following `dplyr` functions can **filter**, **reduce**, or **reorder** the rows of a data frame:

```
dplyr::filter(data_employees_tbl, job_level %in% c(4, 5))

dplyr::distinct(data_employees_tbl, ed_lvl, ed_field)

dplyr::arrange(data_employees_tbl, work_exp)
```

Using `dplyr` to Operate on Columns

The following `dplyr` functions can **select**, **rename**, **add/change**, or **relocate** the columns of a data frame:

```
dplyr::select(data_employees_tbl, dept)

dplyr::rename(data_employees_tbl, job_level = job_lvl)

dplyr::mutate(data_employees_tbl, salary = monthly_comp * 12)

dplyr::relocate(data_employees_tbl, job_lvl, .before = employee_id)
```

Using `dplyr` to Operate on Groups

The following `dplyr` functions can **group** and **summarize** your data by a predefined group indicator:

```
data_employees_tbl |>
  dplyr::group_by(
    job_lvl
  ) |>
  dplyr::summarize(
    annual_comp_mean = mean(annual_comp),
    annual_comp_median = median(annual_comp)
  )
```

In this code chunk, we have grouped by an employee's job level and summarized their annual salary by job level.

Using `dplyr` to Operate on Tables

The following`dplyr` functions can be used to **join different tables (data frames)** together by a unique identifier:

```
data_job <- peopleanalytics::job |> tibble::as_tibble()

data_payroll <- peopleanalytics::payroll |> tibble::as_tibble()

data_job_payroll <-
  data_job |>
  dplyr::left_join(
    data_payroll,
    by = "employee_id"
  )
```

R Resources

https://r4ds.hadley.nz/