PRACTISE YOUR
R PROGRAMMING SKILLS IN
44 EXERCISES

START HERE TO LEARN R

Solution

VOL. 1 VECTORS, ARITHMETIC, AND REGULAR SEQUENCES

HAN DE VRIES

Start Here To Learn R - vol. 1: Vectors, arithmetic, and regular sequences

Han de Vries

Contents

Chapter 1

Deliberate practice to learn R

If you are serious about learning R and being able to use R to solve real-world problems, this book is for you. It is the first volume in a series of compact eBooks, based on the simple premise that the best way to learn programming is by means of deliberate practice. This book offers a lot of practice, through 44 exercises. It also offers you a system to practice *deliberately*: each exercise is a step up from the previous one, and allows you to discover and practice a new concept, or extension to a concept you practised before.

We'll take it slowly but thoroughly! The first 50 pages of this book cover the very, very basics of R: Vectors, arithmetic operators and functions, and regular sequences. In many books you will find an explanation of these topics condensed to a few pages, in some online tutorials to a single blog post, and in popular cheat sheets these topics might have been summarized in three or four lines. I don't think compressing information this way is going to help you to really understand R, and develop the skills to actually use R to solve an actual problem.

To achieve this latter goal, your most important activity will be to sit down behind a computerscreen and keyboard, and interact with R. The exercises in this book are designed as a framework to guide that interaction, challenge you, and allow you to measure your progress. In fact, just reading this book without access to a computer, unfortunately, will be a waste of your time. I recommend to read the book on your computer screen (e.g. on the right side), and have an R session running on the left side of your screen.

The final 20 pages cover the application of what you practised in the first 50 pages, in new situations. Rather than learning a new concept, or R function, at the end of the book, I will challenge you to develop solutions to real problems, by writing short R scripts using R data structures and functions appropriately.

Throughout the book, the level of difficulty varies somewhat, deliberately. Like a running workout, some exercises act more like a warm-up, while others offer you just a lot of practice (mileage building), and still others aim to push you hard for a while (sprint). You should feel free to "break away from the pack", and skip an exercise if you are already thoroughly familiar with a particular function, or *peek at the solution* after you tried many times to solve the problem.

Yes, that wasn't a typo: All solutions to the exercises are included at the end of each chapter. Right below each exercise you will find a link that will bring you to the corresponding solution in the document right away. And once you get to the end of the solution, you can click a link back to the corresponding exercise. I tried to find a good balance here: When working on an exercise I don't want you to accidentally see the solution (which is why I didn't include a solution immediately below an exercise). On the other hand, the solutions include a lot of explanation and are important to read after you've done the exercises. So, I created the links between exercises and solutions to allow you to navigate back and forth conveniently.

Before we start with the first set of exercises, two more things: First, you might be interested in the website related to this book: www.r-exercises.com. There you will find thousands of exercises to practise your R skills on a broad set of beginner, as well as advanced, topics.

Second, I'd love to hear your feedback! Either on specific exercises, the book, or the approach of practising R in general. Please send me an email (han@r-exercises.com), or use hashtag #rexercises (@rexercises) on twitter.

Now, let's practise and start the first set. Ready, set, go!

Chapter 2

Creating vectors

New in this set
`c`, `<-`, `->`, `assign`

A vector is the most elementary way to store and structure data in R. For now, think of it as a list of numbers, which can be as short as a single number, or as long as about 2 billion(!) numbers. Perhaps you were used to working with lists of numbers already in a spreadsheet application (E.g., a row or column filled with numbers in Microsoft Excel), or statistics package (e.g. a numeric variable in SPSS or SAS). However, in R, vectors have so many applications that go beyond the use of data in the examples I just mentioned. R vectors are the basic building blocks underlying the most fancy dashboards, interactive apps, machine learning models, tables and figures.

Because vectors are such a key concept, we're going to practise their use and application slowly, step-by-step. For now we'll just practise *numeric* vectors (and save other types, such as *character* vectors, for later). In this set we're practising the basics of vectors, i.e. how to create vectors and assign them to a name. In subsequent sets we will practise how to use vectors, in order to e.g. calculate all kinds of statistics, carry out simulations, sort data, or calculate the distance between two cities.

Exercise 1

Let's start really easy (don't worry, we'll quickly move to more challenging problems) with a vector containing just a single number, which we also call a scalar. Enter a vector in R, by just typing a random number, e.g. `100`, at the prompt and hit the Enter key.

(Solution)

Exercise 2

Great! You just created your first vector! Now, let's first enter a vector with more than one number. E.g. a vector containing the numbers 1, 2, 3, 4, 5, in that order. If you enter these numbers just like this, R will respond with an error message. It throws an error, because it needs a little bit more information from our side that we actually want to store those numbers in a vector structure. We have to use the following notation for this:

`c(1, 2, 3, 4, 5).`

Now, enter a vector with the first 5 even numbers in R, and hit Enter.

(Solution)

Exercise 3

Let's now enter a much longer vector, containing the numbers 1 to 10, 10 times (use copy & paste). What do the numbers between square brackets in the R output mean?

(Solution)

Exercise 4

You should be pretty familiar with entering vectors now. You might actually feel a little *bored* by typing all these numbers. Life would be pretty miserable if we would have to enter data this way over and over again in R. But fortunately, there is a neat solution! We can *assign* a vector to a variable name such that we can retrieve the data we have entered, conveniently by just typing the name of the variable.

Try to assign a vector containing the numbers 1, 2, 3, 4, 5 to a variable named `a`, using the assignment operator (`<-`), and see which of the statements below work.

Enter each of the 9 statements one at a time at the prompt, hit Enter, and try to retrieve the contents of `a`, by typing `a` at the prompt after you entered each statement:

a) `a<-c(1, 2, 3, 4, 5)`
b) `a <- c(50, 60, 70, 80, 90)`
c) `a -> c(20, 31, 42, 53, 64)`
d) `c(5, 6, 7, 9, 10) <- a`
e) `c(101, 102, 103, 104, 105) -> a`
f) `a < - c(11, 12, 13, 14, 15)`
g) `a < -c(100, 99, 88, 77, 66)`
h) `assign(a, c(1000, 2000, 3000, 4000, 5000))`
i) `assign('a', c(83, 16, 35, 58, 3))`

(Solution)

Exercise 5

In an R script, you might have created dozens or even hundreds of vectors. In that case, naming them `a`, `b`, `c` etc. is not ideal, because it will be difficult to keep track of what all those letters actually mean. This problem is easily mitigated by using longer, and meaningful, variable names.

Assign the following vectors to a meaningful variable name:

a) `c(2, 4, 6, 8, 10, 12, 14, 16, 20)`
b) `0`
c) `3.141593`
d) `c(1, 10, 100, 1000, 10000, 100000)`

(Solution)

Exercise 6

Create vectors that correspond to the following variables names:

a) bmi
b) age
c) daysPerMonth
d) firstFivePrimeNumbers

(Solution)

Exercise 7

So far, we have created vectors from a bunch of numbers. Instead of numbers, however, you can also enter other vectors, e.g. `c(vector1, vector2, vector3)`, and string them together.

To practise this, let's first create three vectors that each contain just 1 element with variable names `p`, `q`, and `r`, and values 1, 2, and 3. Then, create a new vector that contains multiple elements, using the scalars we just created. I.e., create a vector `u` of length 3, with the subsequent elements of `p`, `q` and `r`.

(Solution)

Exercise 8

To play with this a little more, let's create a longer vector, using only the assignment operator (`<-`), the `c()` function, and the vector `u` we just created. I.e., create a new vector `u` with length 96 that contains the elements of `u` as follows: 1, 2, 3, 1, 2, 3, . . . , 1, 2, 3

Hint: Try to find a compact way to achieve this, using the `c()` function and the assignment operator multiple times. You should not have to type 32 times the letter `u`, but just 13!

(Solution)

Solutions

Solution 1

```
100
```

```
## [1] 100
```

Back to exercise

Solution 2

```
c(2, 4, 6, 8, 10)
```

```
## [1]  2  4  6  8 10
```

Back to exercise

Solution 3

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
##   [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8
##  [19]  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6
##  [37]  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4
##  [55]  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2
##  [73]  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10
##  [91]  1  2  3  4  5  6  7  8  9 10
```

The number in square brackets is a convenience counter that R adds to its output, which helps to locate the position in the vector of each element shown.

Back to exercise

Solution 4

Part a

This works, but is not preferred for stylistic reasons:

```
a<-c(1, 2, 3, 4, 5)
a
```

```
## [1] 1 2 3 4 5
```

Part b

Adding a space before and after the `<-` operator, as in (b), leads to cleaner code that is easier to read.

```
a <- c(50, 60, 70, 80, 90)
a
```

```
## [1] 50 60 70 80 90
```

Part c

Although it is possible to reverse the `<-` operator, it should always point from the vector to its name, not from the name to the vector. So this gives an error:

```
a -> c(20, 31, 42, 53, 64)
```

```
## Error in c(20, 31, 42, 53, 64) <- a : target of assignment
## expands to non-language object
```

Because the previous statement led to an error, `a` didn't change. So, if we look up the contents of `a`, we will see the same output as for part (b).

```
a
```

```
## [1] 50 60 70 80 90
```

Part d

R throws an error again, because the `<-` operator points from the name to the vector, instead of from the name to the vector:

```
c(5, 6, 7, 9, 10) <- a
```

```
## Error in c(5, 6, 7, 9, 10) <- a : target of assignment
## expands to non-language object
```

And, like before in part (c), `a` didn't change. So, if we look up the contents of `a`, we will still see the same output as for part (b).

```
a
```

```
## [1] 50 60 70 80 90
```

Part e

The following statement works, because the `<-` operator points from the vector to the name.

```
c(101, 102, 103, 104, 105) -> a
a
```

```
## [1] 101 102 103 104 105
```

Part f

The `<` and `-` part of the `<-` operator always have to be adjacent. So in the following statement, R reads the `<` operator, and then a `-` sign, instead of the `<-` operator. It executes the statement as a comparison (a topic we'll come to later), and doesn't throw an error. Note that `a` didn't change.

```
a < - c(11, 12, 13, 14, 15)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```
a
```

```
## [1] 101 102 103 104 105
```

Part g

Again, the `<` and `-` part of the `<-` operator always have to be adjacent, so the following is identical to part (f).

```
a < -c(100, 99, 88, 77, 66)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```
a
```

```
## [1] 101 102 103 104 105
```

Part h

Finally, instead of `<-`, we can also use the `assign()` function. But beware that the name of the vector has to appear between quotes. So, the following gives an error:

```
assign(a, c(1000, 2000, 3000, 4000, 5000))
```

```
## Error in assign(a, c(1000, 2000, 3000, 4000, 5000)) :
##  invalid first argument
```

```
a
```

```
## [1] 101 102 103 104 105
```

Part i

Below we use the `assign` function correctly.

```
assign('a', c(83, 16, 35, 58, 3))
a
```

```
## [1] 83 16 35 58  3
```

Back to exercise

Solution 5

```
evenNumbers <- c(2, 4, 6, 8, 10, 12, 14, 16, 20)
zero <- 0
pi <- 3.141593
powersOfTen <- c(1, 10, 100, 1000, 10000, 100000)
```

Back to exercise

Solution 6

Chapter 3

Working with vectors

New in this set
`+`, `-`, `*`, `/`, `^`, `%%`, *recycle rule*, `sqrt`, `pi`, `exp`, `log`, `sin`, `cos`, `acos`, `abs`

In the previous exercise set we practised vectors as a data structure. As I noted at the beginning of that set, perhaps you were already familiar with data in a vector-like structure in other applications such as Microsoft Excel or SPSS. If so, perhaps you also used those data to carry out calculations. In this set, we're going practise all sorts of calculations with vectors, from basic operations like addition and multiplication to somewhat more advanced arithmetics.

Exercise 1

Let's create the following vectors:

`u <- 4`

`v <- 8`

Use the elementary arithmetic operators `+`, `-`, `*`, `/`, and `^` to:

a) add `u` and `v`

b) subtract `v` from `u`

c) multiply `u` by `v`

d) divide `u` by `v`

e) raise `u` to the power of `v`

(Solution)

Exercise 2

Now, suppose u and v are not scalars, but vectors with multiple elements:

```
u <- c(4, 5, 6)
```

```
v <- c(1, 2, 3)
```

Without using R, write down what you expect as the result of the same operations as in the previous exercise:

a) add `u` and `v`

b) subtract `v` from `u`

c) multiply `u` by `v`

d) divide `u` by `v`

e) raise `u` to the power of `v`

(Solution)

Exercise 3

We just saw how arithmetic operators work on vectors with the same length. But how about vectors that differ in length? Let's find out... Consider the following vectors:

```
u <- c(5, 6, 7, 8)
```

```
v <- c(2, 3, 4)
```

Without using R, write down what you expect as the result of the same operations as in the previous exercise:

a) add `u` and `v`

b) subtract `v` from `u`

c) multiply `u` by `v`

d) divide `u` by `v`

e) raise `u` to the power of `v`

Then check your answer with R. Which rule does R use, when it has to deal with vectors of different lengths?

(Solution)

Chapter 4

Vectors and functions

New in this set
`min`, `max`, `length`, `sum`, `mean`, `median`, `var`, `sd`, `cor`, `cov`, `floor`, `trunc`, `ceiling`, `round`, `signif`, `cummax`, `cummin`, `cumsum`, `cumprod`, `sort`, `rank`, `order`, `pmax`, `pmin`

In the previous set we started with arithmetic operations on vectors. We'll take this a step further now, by practising a whole bunch of useful functions. Sofar, the functions we have practised (`log`, `sqrt`, `exp`, `sin`, `cos`, and `acos`) always return a vector with the same length as the input vector. In other words, the function is applied element by element to the elements of the input vector.

Not all functions behave this way though. For example, the function `min(x)` returns a single value (the minimum of all values in `x`), regardless of whether x has length 1, 100 or 100,000.

Exercise 1

Did you know R has actually lots of built-in datasets that we can use to practise? For example, the `rivers` data "gives the lengths (in miles) of 141 "major" rivers in North America, as compiled by the US Geological Survey" (you can find this description, and additonal information, if you enter `help(rivers)` in R. Also, for an overview of all built-in datasets, enter `data()`.

Have a look at the `rivers` data by simply entering `rivers` at the R prompt. Create a vector `v` with 7 elements, containing the number of elements (`length`) in `rivers`, their sum (`sum`), mean (`mean`), median (`median`), variance (`var`), standard deviation (`sd`), minimum (`min`) and maximum (`max`).

(Solution)

Exercise 2

For many functions, we can tweak their result through additional *arguments*. For example, the `mean` function accepts a `trim` argument, which trims a fraction of observations from both the low and high end of the vector the function is applied to.

a) What is the result of `mean(c(-100, 0, 1, 2, 3, 6, 50, 73), trim=0.25)`? Don't use R, but try to infer the result from the explanation of the `trim` argument I just gave. Then check your answer with R.

b) Calculate the mean of `rivers` after trimming the 10 highest and lowest observations. Hint: first calculate the trim fraction, using the `length` function.

(Solution)

Exercise 3

Some functions accept multiple vectors as inputs. For example, the `cor` function accepts two vectors and returns their correlation coefficient. The `women` data "gives the average heights and weights for American women aged 30-39". It contains two vectors `height` and `weight`, which we access after entering `attach(women)` (we'll discuss the details of `attach` in a later chapter).

a) Using the `cor` function, show that the average height and weight of these women are almost perfectly correlated.

b) Calculate their covariance, using the `cov` function.

c) The `cor` function accepts a third argument `method` which allows for three distinct methods ("pearson", "kendall", "spearman") to calculate the correlation. Repeat part (a) of this exercise for each of these methods. Which is the method chosen by the default (i.e. without specifying the method explicitly?)

(Solution)

Exercise 4

In the previous three exercises, we practised functions that accept one or more vectors of any length as input, but return a single value as output. We're now returning to functions that return a vector of the same length as their input vector. Specifically, we'll practise rounding functions. R has several functions for rounding. Let's start with `floor`, `ceiling`, and `trunc`:

- `floor(x)` rounds to the largest integer not greater than `x`

Chapter 6

Applications

Congratulations! You have now arrived at the fifth set, which completes our first lesson. You might remember we started off with numeric vectors, practised arithmetic operators and functions, and then concluded with generating our own sequences of numbers.

In this fifth set we're going to apply what we've learned so far to new situations. So, unlike the previous sets, I won't steer you into a direction to use a certain function, or tell you exactly which steps you have to take. However, all exercises can be handled with what you've practised in the previous sets.

However, this doesn't mean this will be an easy set. Knowing how to solve a real-world problem, by applying what you learned, choosing the right operators and functions, applying them to the data in correct order, is a different skill from what we practised in the previous sets.

If you get stuck, there are several strategies you could try: One strategy is to break down the problem in smaller chunks. You might want to write down what each chuck should accomplish in plain English (or whatever language you're most comfortable with). Then try to find an appropriate operator or function from the previous sets for each chunk.

Another strategy is to convert the problem as stated to a simplified version, e.g. by omitting certain parts or details. Try to solve this simplified problem first. Once you have achieved that, add one part or detail you previously omitted, and try to modify your solution. If it works, add the remaining parts one by one.

Exercise 1

Let's have a look at the `Seatbelts` data, which contain monthly data (from Jan 1969 to Dec 1984) on car accidents in Great Britain. To access the various variables, first enter:

```
attach(as.data.frame(Seatbelts))
```

(we'll get to the meaning of this statement at a later stage, don't worry about this now)

a) Calculate the following statistics for the number of car drivers killed:

   - sample mean
   - sample standard deviation
   - sample skewness
   - sample excess kurtosis

(find the formulas for these statistics online).

b) Calculate the average number of car drivers killed per month for the period jan 1969 - jan 1983 (no compulsory wearing of seatbelts), and the period feb 1983 - dec 1984 (compulsory wearing of seatbelts). Note: there is a variable which indicates if the law was in effect for each observation in the data.

(Solution)

Exercise 2

The `islands` data is a vector with the areas (in thousands of square miles) of the landmasses on earth which exceed 10,000 square miles. It is a *named* vector, so if you enter `islands` at the prompt, you will see the names of the landmass associated with each area. If you look at the data, you will notice that some areas are extremely large (e.g., Asia), while others are extremely small (e.g., Vancouver).

a) Convert the landmasses in `islands` from square miles to square kilometers. Find an appropriate formula for this online. What is the total landmass (in square kilometres) of all landmasses in `islands` combined?

b) Suppose all landmasses in `islands` would be one large continent with the exact shape of a circle. How long would it take to drive from a point on the edge of this circle in a straight line through the center of the circle to a point on the edge on the opposite side, assuming an average speed of 55 mph?

c) Find out if some sort of 80/20 rule applies. I.e., if we take 20% of the (number of) largest islands, do they cover 80% of the total landmass (in square miles) in the data, or is it less, or more?

(Solution)