5 Utilities

Mastering R programming is not only about understanding its programming concepts. Having a solid understanding of a wide range of R functions is also important. This chapter introduces you to many useful functions for data structure manipulation, regular expressions, and working with times and dates.

Useful Functions

By now, you surely have some idea of the huge amount of useful functions that are available in R. The functions of the apply family that you've learned about before are just an illustration of this diversity. Along the way you also used the sort function, the print function, the identical function, and many more. In this chapter, I'm going to talk about a number of easy, but very often used functions in the R language.

First, I'm going to talk about some math-related functions and next, I'll head over to functions that relate more closely to R's data structures. If you aced all the previous exercises, you should recognize some of them. Have a look at the following R code that uses several mathematical functions. The first two lines of code are nothing new; they simple create two vectors, v1 and v2. The last line of code is less straightforward so let's chop it up into pieces and see what each of the function components does along the way. The innermost function, abs() calculates the absolute value of an array of numerical values, in our case the vectors v1 and v2. I went ahead and already replaced the variable names with the actual vectors that they represent. The result is what we would expect. We get the positive value of all elements. Let's replace the vectors that result from calling the abs() functions in the expression, giving us the following line.

```
Mathematical utilities
v1 <- c(1.1, -7.1, 5.4, -2.7)
v2 <- c(-3.6, 4.1, 5.8, -8.0)
mean(c(sum(round(abs(v1))), sum(round(abs(v2)))))
```

```
abs(c(1.1, -7.1, 5.4, -2.7))
```

```
1.1 7.1 5.4 2.7
```

```
abs(c(-3.6, 4.1, 5.8, -8.0))
```

```
3.6 4.1 5.8 8.0
```

```
mean(c(sum(round(c(1.1, 7.1, 5.4, 2.7))),
       sum(round(c(3.6, 4.1, 5.8, 8.0)))))
```

The next inner function that we encounter is the round() function, which rounds the input. In the first round function, 1 (point) 1 gets rounded to 1, while 2 (point) 7 is rounded to 3. In the second round call 3 (point) 6 becomes 4 and 8 (point) 0 becomes 8.

```
round(c(1.1, 7.1, 5.4, 2.7))
```

```
1 7 5 3
```

```
round(c(3.6, 4.1, 5.8, 8.0))
```

```
4 4 6 8
```

If we replace the round() function with their results, we arrive at the sum() function. This function simply computes the sum of the input array. If you pass a matrix as an argument to the sum function, for example, the sum of all the matrix elements gets returned. In our example, a vector is passed to the sum function, so R simply calculates the sum of the vector elements. The first sum is 16, while the second sum equals 22. We've almost broken down our expression entirely if we fill in the results of both the sum calls.

```
sum(c(1, 7, 5, 3))
```

```
16
```

```
sum(c(4, 4, 4, 8))
```

```
22
```

The mean function calculates the arithmetic mean. Again, mean() is a generic function that is capable of handing different types of R objects, but most commonly you would use it on numerical arrays. In our case, the input to mean is a vector of length 2, containing the values 16 and 22. Recalling our primary school level math, it isn't a surprise that the result of this call is 19. Finally! We arrive at the end of our bulky arithmetic one-liner: the result is 19. This corresponds to the result that we get when we execute the code chunk we began with.

```
mean(c(16, 22))
```

```
19
```

This was still pretty easy, right? Let's head over to the next example to introduce some of the functions you'll often use when creating and manipulating data structures. Take a close look at this big fella'. No need to panic! There are some things we already know here. The list() function creates a list. Apparently, there are three list elements, named log, ch and int_vec. log is simply a logical, TRUE, ch is a character string, hello, but what about int_vec?

```
Functioms for data structure
li <- list(log = TRUE,
           ch = "hello",
           int_vec = sort(rep(seq(8, 2, by = -2), times = 2)))
```

Let's have a closer look. The innermost function here, is the seq() function seq generates a sequence of numbers. The first two arguments tell R the limits of the sequence. That is, where to start and end the sequence, respectively. The by argument specifies the increment value for the sequence on each step. For instance, this line of code would generate a sequence starting at 1, going to 10, with steps pf size 3. For our example, the seq function call would read as: generate a sequence from 8 to 2, while taking steps of -2. We get a vector of length 4. As before, let's go ahead and fill in the resulting vector to simplify the expression for int_vec.

```
sort(rep(seq(8, 2, by = -2), times = 2))
```

```
seq(1, 10, by = 3)
```

```
1 4 7 10
```

```
seq(8, 2, by = -2)
```

```
8 6 4 2
```

We now arrive at the rep() function. The rep() function has the ability to replicate its input, which typically is a vector or a list. Using the times argument, we can specify how the replication should happen. If the times argument is a vector of length 1, it tells rep() how often the entire structure should be repeated. This is case in our example, which results in a vector of length 8. You could also use the each argument inside rep. Instead or repeating the entire vector, every element gets repeated. Can you spot the difference? There are more advanced ways of using rep() which I won't detail here. Remember, help is just a single question mark (?) away in R! So our rep() call with the times argument results in a vector of length 8. Let's go ahead an replace the call with this result.

```
rep(c(8, 6, 4, 2), times = 2)
```

```
8 6 4 2 8 6 4 2
```

```
rep(c(8, 6, 4, 2), each = 2)
```

```
8 8 6 6 4 4 2 2
```

Finally, the sort() function is a generic function for sorting an input vector. You can use it on numerical values, but also on character and logical vectors. It works as you would expect, sorting the input vector in ascending order. By setting the decreasing argument, which is FALSE by default, to TRUE, we can reverse the order of arranging. So that's that. We've reverse engineered the bulky expression for the int_vec element inside our list. To check if we decomposed this expression correctly let's see what a direct evaluation looks like. Looks good! We thus end up with the following list definition. Remember the str() function to inspect the structure of this list? It's a great function to see the contents of your data structures in a concise way. Before you can roll up your sleeves for the some of the interactive exercises have a look at the following R expression.

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

```
2 2 4 4 6 6 8 8
```

```
sort(c(8, 6, 4, 2, 8, 6, 4, 2), decreasing = TRUE)
```

```
8 8 6 6 4 4 2 2
```

```
sort(rep(seq(8, 2, by = -2), times = 2))
```

```
2 2 4 4 6 6 8 8
```

```
str(li)
```

```
List of 3
 $ log     : logi TRUE
 $ ch      : chr "hello"
 $ int_vec : num [1:8] 2 2 4 4 6 6 8 8
```

This is (dot) functions are functions you can use to check the type of your data structure.

They return a logical. Because li is a list, is (dot) list of li returns TRUE. On the other hand, is (dot) list on a vector returns FALSE. Instead of is (dot) function, R also provides the as (dot) functions. These can be used to convert vectors to lists, for example. Now, calling is (dot) list on li2 will return TRUE as the vector was converted to a list using as (dot) list().

```
is.list(li)
```

```
TRUE
```

```
is.list(c(1, 2, 3))
```

```
FALSE
```

```
li2 <- as.list(c(1, 2, 3))
is.list(li2)
```

```
TRUE
```

Next, there might be some cases in which you want to convert your list to a vector. In these cases, you might want to use the unlist function. R flattens the entire list structure and returns a single vector. Notice the coercion here. Because vectors can only contain a single atomic type, the logical TRUE and the numerical vector are all coerced to character strings. Also notice how R tries to come up with meaningful names for all vector elements.

```
unlist(li)
```

```
   log      ch int_vec1 int_vec2 ... int_vec7 int_vec8
"TRUE" "hello"      "2"      "2" ...      "8"      "8"
```

Finally, you should definitely check out the append() and rev() functions. The append() function allows you to add elements to a vector or a list in a very readable way. In combination with the rev() function, which reverses elements in a data structure, we could create a new version of li that contains the same data in the different order. The rev() function first reverses the list, placing the int_vec vector first and the log value, TRUE last. Afterwards, append() concatenates the original vector li and its reversed version to create a list of 6 elements, double li's length. Notice how I used the str() function here to inspect the structure.

```
str(rev(li))
```

```
List of 3
 $ int_vec : num [1:8] 2 2 4 4 6 6 8 8
 $ ch      : chr "hello"
 $ log     : logi TRUE
```

```
str(append(li, rev(li)))
```

```
List of 6
 $ log     : logi TRUE
 $ ch      : chr "hello"
 $ int_vec : num [1:8] 2 2 4 4 6 6 8 8
 $ int_vec : num [1:8] 2 2 4 4 6 6 8 8
 $ ch      : chr "hello"
 $ log     : logi TRUE
```

Regular Expression

A possibly dreaded yet very important aspect of R are regular expressions. But what is a regular expression? Well, it's nothing more than a sequence of characters and metacharacters that form a search pattern which you can use to match strings. You can use a regular expression to check whether certain patterns exist in a text, to replace these patterns with other elements or to extract certain patterns out of a string. Regexes are particularly handy when you want to clean your data. You'll often turn to regular expressions to make your data ready for further analysis, especially when you're working with data from the web or from different sources. A comprehensive discussion of regular expressions could be a course by itself, so I won't go into too much detail here. First, I'll talk about the grepl() and grep() functions. Next, I'll go after the sub() and gsub() functions. Have a look at this vector of character strings that represent some animals. With the grepl()

```
animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
grepl(pattern = <regex>, x = <string>)
```

```
grepl(pattern = "a", x = animals)
```

```
TRUE FALSE  TRUE  TRUE FALSE
```

function, we can determine, for example, which of these animals has an “a” in their name. The first argument of grepl() is the pattern, while the second is the character vector where matches are sought. In our case, we’re looking for the pattern “a”, because we want to find the animals that have an “a” in their name. The x argument is equal to animals, the vector of animal names. The results makes sense. There is an a in “cat”, so a TRUE value signals that this pattern was found. In “moose”, on the other hand, there is no “a”, so the corresponding element is FALSE. Matching simply for “a” is great, but we can do much more with regular expressions. What if we want to match for strings that start with an “a”? We can use the caret metacharacter here. If we change our pattern from “a” to “caret a”, we see that only “ant” is matched, because it’s the only name from the animals vector that begins with an “a”. Just as the caret matches the empty at the beginning of a line, the dollar sign matches the empty string at the end of a line. So if we want to match for animals that end with an a, we can use an a followed by a dollar sign. This time, only “impala” is matched. There are many other metacharacters that I will not discuss here. If you want to learn more about them, you can check out the documentation on regular expressions in R by typing question mark regex in the console.

```
grepl(pattern = "^a", x = animals)
```

```
FALSE FALSE FALSE  TRUE FALSE
```

```
grepl(pattern = "a$", x = animals)
```

```
FALSE FALSE TRUE  FALSE FALSE
```

Apart from the grepl function, there is also the grep function. This function returns a vector of indices of the elements of x that yield a match. That’s quite different from grepl. Compare the grepl command, that gives a vector of logicals, with the grep command. However different, they are obviously related: grep simply gives the indices of the TRUE elements that the grepl function returns. One way in which you could compare grep and grepl would be using the which() function given a logical vector as input, this function returns the indices for which that vector is TRUE. If you try it out with our animal-matching attempts, you will get a familiar answer. This is precisely the output of the grep function. Of course, grep knows how to handle the different types of regular expression patterns just as grepl does. The pattern to match for strings which start with an “a” in combination with the grep function returns only 4, the index of “ant” inside animals.

```
grepl(pattern = "a", x = animals)
```

```
1 3 4
```

```
which(grepl(pattern = "a", x = animals))
```

```
1 3 4
```

```
grep(pattern = "^a", x = animals)
```

```
4
```

We now have covered some basics on how to check for the existence of patterns inside a vector of character strings. R, however, also provides some functions to directly replace these matches with other strings. I’m talking about the sub function. It basically takes three arguments: pattern, replacement, and x. Once again, the pattern argument corresponds to the regular expression you want to match strings. x is the character vector where these matches are sought. Finally, you assign a replacement value for the matches to the replacement

```
sub(pattern = <regex>, replacement = <str>, x = <str>)
```

```
sub(pattern = "a", replacement = "o", x = animals)
```

```
"cot" "moose" "impola" "ont" "kiwi"
```

argument. To see how this works, let's what happens if we set the pattern argument to "a", matching all characters "a", and replacement to "o". As before, x is simply equal to animals. As we'd expect, the "cat" string gets converted to "cot", so the "a" is replaced with an "o". In "moose" there were no "a"'s so nothing got replaced. In "impala" however, there are two "a"'s, but only the first "a" has been replaced with an "o". How come? Well, that's because the sub() function only looks for the first match in the string, and if it finds it, replaces it with the replacement argument, and immediately stops looking. If you want to replace every single match of a pattern in a string with the replacement argument, you should try the gsub function instead. Now, "impala" gets converted to "impolo", so the two "a"'s have been replaced.

```
gsub(pattern = "a", replacement = "o", x = animals)
```

```
"cot" "moose" "impolo" "ont" "kiwi"
```

There is one last metacharacter I want discuss, the vertical bar or the OR metacharacter. Its meaning is quite similar to the or operator you learned about to combine logicals. You can use it to match for different options. Say, for example, you want to replace every "a" or "i" with an underscore for the animals character vector. It is straightforward to use the pattern "a" vertical bar "i" inside gsub. Now, all a's and i's replaced by an underscore. Of course, you can extend this pattern even further.

```
gsub(pattern = "a|i", replacement = "_", x = animals)
```

```
"c_t" "moose" "_mp_l_" "_nt" "k_w_"
```

Times & Dates

One more thing I want to talk to you about are dates and times in R. Time information can come in pretty handy in different situations. For example, imagine you're writing a script that has to be run hourly on a remote server. Generating log files that contain timing information can help you structure log files and trace potential problems. For other very specific applications such as time-series analyses and seasonality studies. R's power to deal with times and dates will prove extremely useful.

The first step in our exploration will be simply ask R what the current date is. The output tells us that we're in the year 2019, the ninth month and seventh day. That is, the seventh of September, 2019. Time is going fat, as always. Is the variable today simply a character string or is there something else going on? Let's have a look to the class() function to see type of the today variable. today is a "Date" object, a special kind of R object that represents dates. To get the current time in R, we use Sys (dot) time(). We get both the time and the date in a very clear format. Notice the difference in capitalization here. Sys (dot) Date() is written with a capital D while Sys (dot) time() is written with a small t. As you'd expect, also the variable now is not a simple string. Let's find out with the class() function. The important class here is POSIXct. Apart from providing rich functionality for calculus and formatting, this class makes sure that the dates and times in R are compatible across different operating systems, according to the POSIX standard.

```
today <- Sys.Date()
today
```

```
"2019-09-07"
```

```
class(today)
```

```
"Date"
```

```
now <- Sys.time()
now
```

```
"2019-09-07" 10:34:52 CEST
```

```
class(now)
```

```
"POSIXct" "POSIXt"
```

Great, we know how to get the current date and time. What about creating Dates for other days? For example, let's say we want to get a Date object for May 14 in 1971. We can use the as (dot) Date() function to convert a character string to a Date. my_date is now an object of class Date. How did R know which elements of the string corresponds to the day of the month, which to the month and which to the year? Let's see what happens if we scramble the order of the date elements. R fails. That's because as (dot) Date() tries some default data formats. It first tries the ISO date format, which is the most common around the globe. This format represents the year with four digits, dash, month as a 2 digit number, dash, day as a two digit number. Because our first character string had this format, R was able to convert the string to a Date. However, for the second example, R failed because the date couldn't be inferred. We can fix this by setting the format argument of as (dot) Date() explicitly.

```
my_date <- as.Date("1971-05-14")
my_date
```

```
"1971-05-14"
```

```
class(my_date)
```

```
"Date"
```

```
my_date <- as.Date("1971-14-05")
```

```
Error in charToDate(x) :
  charcter string is not in a standard unambiguous format
```

If we specify that the first part of our date is the Year, the second part is the day of the month and the third part is the month, we can have R understand what we mean. If we now print my_date, we see the same date as before, great! There are many more ways to format a date, for example using the full name of a month or a two digit year. You can find a comprehensive list in the interactive exercises.

```
my_date <- as.Date("1971-05-14", format = "%Y-%d-
%m")
```

```
"1971-05-14"
```

To convert a string denoting an exact time, we can use the function as (dot) POSIXct(). Once again you could use the format argument inside as (dot) POSIXct to convert character strings with a different layout.

```
my_time <- as.POSIXct("1971-05-14 11:25:15"
my_time
```

```
"1971-05-14 11:25:15 CET"
```

The cool thing about R objects from Date and POSIXct classes is that you can do calculations with them. Consider my_date, which we created from a character string earlier. If we simply increment my_date with 1, R will increment the date by 1 day, giving us the 15th of May in 1971. Say we now want to know the difference between the 29th of September in 1998 and my_date. We first create a new data object, my_date2, from a character string. Next we can simply do a substraction, calculating the difference in days. There appears to be a time difference of exactly 10000 days, what a coincidence! Computations with POSIXct objects happen in the exact same fashion. The only difference is that the "time unit" of POSIXct is not a day as for Date objects, but is a second. Suppose we increment the my_time we created earlier by 1. We get the following time, so we're one second further in time now. To get the difference between times, let's create a new time, and calculate the difference with my_time.

```
my_time2 <- as.POSIXct(1974-07-14 21:11:55 CET")
my_time2 – my_time
```

```
"1971-05-15"
```

```
my_date2 <- as.Date(1998-09-29")
my_date2 – my_date
```

```
Time difference of 10000 days
```

```
my_time + 1
```

```
"1971-05-14 11:25:16 CET"
```

Because the time difference is so large, R simply displays the time difference in days. For any time difference, R will automatically display an easily interpretable time difference.

How is R able to do all these calculations so seamlessly? Well, under the hood, R represents dates and times as simple numerics. A Date object is simply

```
Time difference of 1157.407 days
```

a more advanced representation of the number of days since the first of January in 1970. If we unclass the my_date object, thus converting it from a Date to a numeric, we see the number 498, because the 14th of May in 1971 is exactly 498 days from the first of January in 1970. If you're calculating the difference between two days, in fact you're simply calculating the difference between numerical values, and that's something R is particularly good at. The same holds for the POSIXct objects, which are actually simple numerics that hold the number of second since the first of January in 1970 at midnight.

```
unclass(my_date)
```

```
498
```

```
unclass(my_time)
```

```
43064715
```

Now you know how R handles times and dates behind the scenes, you should have a clear idea of what you can and can't do with dates in R. Now R wouldn't be R if there weren't dedicated packages to deal with times in a more advanced fashion. If you want to learn more, You can check out the lubridate, zoo and xts packages, they're really neat.