Data Manipulation

> You might not think that programmers are artists, but programming is an extremely creative profession. It's logic based creativity.
> -- John Romero

We already introduced data structures, namely vectors, matrices, lists, and data frames and performed some basic tasks with them. R supports, of course, many more interesting and advanced actions to manipulate data in any desired way.

Vectors
We have already seen that we can access a subset of a vector using brackets. This option is far more powerful than shown before when combining the brackets with other operators.

```{r}
vec <- 1:10

vec[5]                # Show fifth element
vec[1:3]              # Show first to third element
vec[-1]               # Exclude first element
vec[-length(vec)]     # Exclude last element

```


R provides handy commands like `any()` and `all()` to manipulate data and perform fast logic tests.

```{r}
any(vec > 5)
all(vec < 10)
all(vec[-length(vec)] < 10)
```

You can also use brackets to filter data and advise R to only return specific values which match your filter criteria.

```{r}
vec[vec > 5]          # Brackets can hold expressions to filter data
vec[vec > 7] <- 0     # Setting all observations > 7 to zero
vec

subset(vec, vec < 5)  # subset() can also be used to filter data
```

Using `which()` it is possible to perform actions on indices which can be used to remove values from a vector for example.

```{r}
which(vec < 1)                          # Get indices of zeros in vec
vec[which(vec < 1)]

vec.positive <- vec[-which(vec < 1)]   # Remove zeros from vec
vec.positive
```

As in the real world, there are always multiple solutions to a problem. If you want to remove the zeros from the shown vector `vec` there are additional ways to receive the same result as for `vec.positive` in a more efficient, but not always more clear way. Make sure to understand the following solutions in addition to the one shown using `which()`.

```{r}
vec[vec > 0]                        # Nice, clean and efficient
subset(vec,vec > 0)                 # Same result using a function
vec[-vec < 0]                       # Also right, but not clear or fast
```

Let us have a detailed look at how the third and maybe confusing solution is derived. You should not use it as the others shown are more clear and way nicer, but it illustrates nicely how weird a solution can be achieved.

```{r}
-vec
-vec < 0
vec[-vec < 0]
```

Matrices and Arrays
Matrices support the same operations as vectors and of course, you can formulate ambitious filtering commands to extract or manipulate data.

```{r}
mat <- matrix(1:25, nrow = 5)
10 %% 3                             # Modulo operation
mat %% 2 == 1
mat[which(mat %% 2 == 1)]
```

Matrices as known from undergraduate mathematical courses usually have two dimensions. Sometimes there is the need for more dimensions and of course, R supports us with a construct that is similar to a more dimensional matrix - this structure is called an array. If you already played around with other programming languages you may insist, that an array does not need to have three or more dimensions, which is absolutely right. You can imagine *array* as a generic term for a (numeric) n-dimensional data-structure. A vector, therefore, can be called a one-dimensional array while matrices can also be called two-dimensional arrays. If you store data in a three-dimensional structure - the array is normally called a cube, but of course, we are not limited to three dimensions. Cubes of higher dimensions are usually called Hypercubes or just referred to as arrays. To printout, our n-dimensional structure on the two-dimensional command-line R gives back the array in slices. The command to create an array is `array()`.

```{r}
A <- array(c(1:9 , 11:19 , 21:29), dim=c(3 , 3 , 3))

A                                   # Display array

dim(A)
```

When entering higher dimensions the complexity level increases drastically and may get confusing. During our data science journey, we mostly use two-dimensional structures and therefore won't explore the world of n-dimensional arrays here.

Lists
Lists are somewhat similar to arrays because a list can contain another list and so on. This makes lists slightly more complicated than matrices. If we access an element within a list with the `[]`-Operator we get another list as output. To access the value of a list (within a list) we must use the double bracket `[[]]`-operator. The single bracket element tells you which subelement of a list is displayed and the double bracketed indices give you the specific element.

```{r}
L <- list(5:10,c("C++","R","Phyton"),c(TRUE , FALSE))

L[1]
typeof(L[1])
typeof(L[[1]])
L[[2]][2]                        # Accessing a specific element
L[2][2]                          # Works only with double brackets
```


Dataframes
Dataframes are only lists consisting of vectors of equal length, so all the list specifics also do apply for dataframes. Nevertheless, if we are operating in special matrix-like structures R supports us with a bunch of specialized functions that allow us to perform very fast manipulations. Some of the useful functions provided by R are `merge()`, `apply()`, `sweep()`, `stack()`, `aggregate()` and `transform()`.

Merging rows and columns {-}
We already discussed the functions `rbind()` and `cbind()` to merge data, especially matrices by row or by column. But if handling more complex datasets this won't get rid of duplicates, contained in each table, manually. Exactly for these cases the `merge()`-function was created.

```{r}
d <- data.frame(ID=1:4,list(
  Name=c("Homer","Marge","Bart","Lisa"),
  Age=c(38 , 34 , 10 , 8),
  Sex=c("m","f","m","f")
))

e <- data.frame(ID=c(1,4,3,2),list(
  Name=c("Homer","Lisa","Bart","Marge"),
  Height=c(182 , 120 , 122 , 223),
  Weight=c(108 , 33 , 35 , 58)
))

# Merge even works with unsorted dataframes and matches the datasets
# fully automatically using common columns.

merge(d,e)
```

To add rows in a similar and clever way there are packages available to take care of cases that can't be solved with `rbind()`. One of these functions is called `smartbind()` from the package `gtools`.

Apply functions to rows and columns {-}
A common, very fast and unbelievable useful helper is the function `apply()` which allows us to apply any function to every row or column of a dataframe. The function works in this way: `apply(data, MARGIN=\#, FUN=function() )`. `MARGIN` indicates if the function defined in `FUN` should be applied to rows `MARGIN=1` or columns `MARGIN=2`. `FUN` can be equal to any function, including self-written ones.

```{r}
mat <- matrix(1:10,byrow=T,ncol=5)
mat

apply(mat,MARGIN=1,FUN=mean)  # Equivalent to rowMeans()

apply(mat,MARGIN=2,FUN=sum)   # Equivalent to colSums()
```

This case is only to illustrate how `apply()` works. For the cases shown are specific functions like `rowMeans()` or `colSums()` available which are faster and should be used.

In addition to `apply()` there are two more functions which can be used to apply the desired function on a list: `lapply()` and `sapply()`. They differ only in their output. The function `lapply()` outputs a list, while `sapply()` outputs a vector if that's possible. They follow the exact same syntax as `apply()`.

Sweep out Statistics in a Matrix {-}
The function `sweep()` sweeps out a summary statistics in a way defined by the argument `FUN` with subtraction as default operation. In addition to `apply()` this allows for very fast and completely vectorized manipulations.

```{r}
sweep(mat , MARGIN = 1 , STATS = c(1 , 10))

sweep(mat , 1 , apply(mat , 1 , mean))
```

But `sweep()` is much more powerful than it seems on first sight. Dividing a column by its mean can also easily done with sweep by passing the argument `FUN` to it.

```{r}
mat <- matrix(rep(1:5 , 2) , byrow = T , ncol = 5)
mat

sweep(mat , 2 , colMeans(mat) , FUN = "/")
```

If you have problems using the sweep function you may want to convert your (sub-) dataframe to a numeric matrix, this can be done with `data.matrix()`.

Concatenate all values from a dataframe {-}
To concatenate all values from multiple columns of a dataframe one can use the function `stack()` which outputs a dataframe with the stacked values while ignoring character columns.

```{r}
d.stacked <- stack(d)        # Stacks all numeric values of dataframe d
d.stacked                    # Display the stacked data
```

You may have noticed the column `ind` in the resulting output. This shows the origin of the data and makes the stacking fully reversible with the command `unstack()`, except for eventually lost string containing columns of the dataframe.

Splitting dataframes while applying functions {-}
The function `aggregate()` allows us to split dataframes into subpopulations according to a provided measure and apply the desired function to each population.

```{r}
aggregate(d$Age,by=list(Sex=d$Sex),FUN=mean)
```

Transformations without recreating dataframes {-}
The function `transform()` can easily be used to manipulate columns in a dataframe without the need to recreate the entire dataframe.

```{r}
# Adding two new lines to the dataframe d
d <- data.frame(d,list(
  Height=c(182 , 223 , 122 , 120),
  Weight=c(108 , 58 , 35 , 33)
))

d    # Display the extended dataframe

transform(d,Height=Height/100,
            BMI=Weight/(Height/100)^2)
```

Strings
Strings are not only used to provide a description of your data. When dealing with more complex programs they become more and more useful. An often used case is creating variables while a program is running and using the created variables in the same instance to perform calculations without knowing the exact scheme of the variable names. This programming technique is called dynamic variable naming and we are going to explore this later in the course. But to be able to handle strings we need a couple of useful functions to deal with them.

When creating functions that output calculations on the command line like the `lm()`-function does when calculating linear models, it is useful to manipulate the appearance in the output to create a better readable experience for the user.

```{r}
```

```
string <- c("Statistics","and","calculus","are","wonderful!")
string

noquote(string)
cat(string)
```

Additional useful functions to manipulate the appearance of output are `print()`, `format()` and `sprintf()`. To concatenate strings into a single variable we commonly use the `c()`-function, if we really want to combine multiple strings into a single one we can use `paste()`.

```{r}

letters                 # Reserved word for all 26 small letters

LETTERS                 # Reserved word for all 26 capitalized letters

paste(letters[1:5],LETTERS[1:5],sep="+")

paste(letters[1:5],LETTERS[1:5],collapse="",sep="")

```

If you want to create a bunch of variable or column names consisting of the same string, but different numbers you can simply pass that string and a vector of numbers to `paste()` and it will return the desired combinations, which then can be used to name the columns of a dataframe for example.

```{r}
paste("name" , 1:3 , sep="")       # Easy name or variable generation
```

To extract substrings or to split a string to a certain scheme R provides the functions `substring()` and `strsplit()`. If you want to search through a string you can use `grep()` to find the respective index of the searched string in a bigger string or vector of strings. To replace parts of a string R provides the commands `gsub()` to replace all occurrences and `sub()` to replace the first occurrence in the target string.

If you are familiar with Linux or the world of Unix-based operating systems you may already be familiar with a lot of these string manipulating functions. In fact, most of these functions found in R can also be found in your favorite Linux shell as they originated from there.

```{r}
# To convert a String to upper or lower cases one can use the
# functions toupper() and tolower().

identical(toupper(letters),LETTERS)

grep("K", LETTERS)            # Returns index of searched string

string <- paste(rep(letters,2),collapse="",sep="")
string
```

```
gsub("o","X",string)

sub("o","X",string)
```

If you have a more complex task like finding generic patterns in strings or simply want to find more than just simple predefined letters or words you may want to make yourself familiar with regular expressions. Regular expressions (regex) allow you to come up with a generic description of what should be searched and returned in textual data and they often come in handy when you want to extract specific parts of a textual dataset for further processing. As the syntax of regex is quite confusing when seen for the first time and textual data isn't our main focus here we skip this part. Nevertheless, if you are interested in using them the built-in help system provides an excellent starting point when asked for `?regex`.

Dates and Times

Dates and times in computer science can easily fill a bunch of books. There are different time formats, different data types and there are countless routines to handle, measure and manipulate time objects. So let us focus on the basics to implement basic features in your programs when it comes to times. In R we have two simple methods to obtain the current time.

```{r}
date()

Sys.time()
```

Once we have a date we can encode and convert it to a POSIXlt-Object with the function `strptime()`. POSIX-Objects store the number of elapsed seconds since the 1. January 1970 - 00:00. Once we have a POSIXlt-Object it is easy to extract things like the month using the function `months()` or the day of the week using `weekdays()`.

```{r}

Sys.setlocale("LC_TIME","C")           # Set regional parameters once

strptime("15/mar/88",format="%d/%b/%y")

date <- strptime("15/mar/88",format="%d/%b/%y")

weekdays(date)

months(date)
```

As you can see in the code snippet above you need to pass a proper `format` argument to the function `strptime()` in order to decode or encode the given time accordingly. This may look like hieroglyphs at first sight but is indeed quite easy to learn. The following table provides an overview of the most useful codes. A more extensive reference can be found in the help system (`?strptime`) or in the Linux man pages, as the POSIX standard also originated from the Unix-universe.

|Code | Input / Output
|
|:---|:---|
| `%a` and `%A` | Abbreviated and full weekday name in the current locale. |
| `%b` and `%B` | Abbreviated and full month name in the current locale.
|
| `%d` | Day of the month as decimal number (01–31).
|
| `%e` | Day of the month with leading space for single decimal numbers (1–31). |
| `%H` | Hour as decimal number (00–23).
|
| `%m` | Month as decimal number (01–12).
|
| `%M` | Minute as decimal number (00–59).
|
| `%S` | Seconds as decimal number including 2 leap seconds (00–61). |
| `%T` | Equivalent to `%H:%M:%S`.
|
| `%U` | Week of the year as decimal number (00–53).
|
| `%V` | Week of the year as decimal number as defined in ISO 8601 (01–53). |
| `%W` | Week of the year as decimal number (00–53, UK convention). |
| `%y` | Year without century (00–99).
|
| `%Y` | Year with century (ISO 8601:2004).
|
| `%F` | Equivalent to `%Y–%m–%d` (ISO 8601 date format).
|
| `%D` | Date format such as `%m/%d/%y` (ISO C99 date format).
|

When operating with POSIX-Objects we can perform lots of useful tasks which make handling time objects really convenient like comparing them and calculating the elapsed time between two events.

```{r}
Event1 <- as.POSIXlt("1989-11-09") # Fall of Berlin wall
Event2 <- as.POSIXlt("1949-05-23") # Founding date of BRD

Event1 - Event2

difftime(Event1, Event2, units="hours")

Event1 > Event2
```

At this point, we are quite well equipped with knowledge to handle and manipulate all common appearances of data, but programming is much more. Data is important but we can leverage its ability only when we have more powerful tools to automatize dealing with it.

Exercises {-}
```{r, echo=FALSE, fig.cap=NULL, out.width="100%"}
knitr::include_graphics("gfx/CH00-ExercisesHeader.png")
```


EX 1 {-}

```{block2, type='rmdexercise'}
The function ´apply´ offers three main arguments ´X´, ´MARGIN´, ´FUN´.
Please explain their purpose and possible parameterization options.

```


EX 2 {-}

```{block2, type='rmdexercise'}
What is the idea behind the additional three dot argument ´...´
(dotdotdot) that can be set when using ´apply´?

```


EX 3 {-}

```{block2, type='rmdexercise'}
Please explain the following three lines of code (line by line):

```


```{r, eval=FALSE}
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
```


EX 4 {-}

```{block2, type='rmdexercise'}
Explain the difference between the two function calls without executing
them.

```


```{r, eval=FALSE}
paste("variable", 1:5, sep="")
paste("variable", 1:5, collapse="", sep="")
```


EX 5 {-}

```{block2, type='rmdexercise'}
What does the following code produce? Please derive the output and explain
what happens before executing the code.

```

```{r, eval=FALSE}
mat <- matrix(1:4, nrow=2)
mat2 <- mat %% 2 == 0
apply(mat2, 2, function(x){x[1]|x[2]})
```

EX 6 {-}

```{block2, type='rmdexercise'}
Calculate the average value of each column in the following matrix ´m´
after removing lines 10, 24 and 30-37.

```

```{r, eval=FALSE}
set.seed(5)
m <- matrix(sample(1:10^7), ncol=10, byrow=T)
```

EX 7 {-}

```{block2, type='rmdexercise'}
Try to come up with an explanation of what the following code does and why
it works.

```

```{r, eval=FALSE}
'['(mat, 2, 2)
```