Basics

Ulrich Bergmann

6 February 2020

Before we can dive into the actual work with data you should be familiar with the basics of the R programming language. I will not spend too much time here as you have already seen many of the concepts in our Python sequence and they will only look slightly different in R.

1 Basic basics

1.1 Working directory

To know in which folder we are working, you can use the `getwd` function.

```r
getwd()
```

```
## [1] "/home/bergmul/documents/teaching/pp4rs/2020/dev/R/basics"
```

If you want to set a specific working directory, you can do this using the `setwd` function. We will not follow this approach as there is a better one by now.

```r
#setwd("/Users/jlanger/Dropbox/uzh_programming/r_programming")
```

Instead of setting the working directory explicitly, we create a new `R` project. In order to do so, we click on *File > New Project* and create a new project folder in a directory of our choice.

1.2 R projects

This new folder now includes an `.Rproj` file. If we click on this file a new RStudio session is started, our environment/workspace is cleaned, and our working directory is automatically set to the project folder.

Notice also that our command history and all open editor windows for the project are restored once we open it.

1.3 Removing and adding objects from workspace

Let's create two variables.

```r
y = 1
x = 2
```

To see which objects are available in our workspace, we can use the `ls` function.

```r
ls()
```

```
## [1] "x" "y"
```

If you want to remove an object, use the `rm` function.

```r
rm(y)
ls()
```

```
## [1] "x"
```

If you want to remove all object to start with a clean slate, type the following.

```r
rm(list = ls())
ls()
```

```
## character(0)
```

1.4 Installing and loading packages

Additional functionalities are available in R packages. To install these you have to use the `install.packages` function. The package name has to be passed as a string.

```r
# you do neet the repos argument!
# install.packages("purrr", repos = "http://cran.us.r-project.org")
```

To then load a package, we use the `library` function. Here, we will load a packages with the curious name `purrr` that we'll use later.

```r
library(purrr)
```

2 Vectors

Vectors provide the basic organization of data in R. There are basically two types of vectors:

- atomic vectors – logical, numeric, character – which are homogenous,
- lists, which can be heterogenous and can contain other lists.

Let's start with the atomic vectors.

2.1 Atomic vector types

2.1.1 Logical

Logical vectors (or booleans) can have three different values in R:

```r
TRUE
```

```
## [1] TRUE
```

```r
FALSE
```

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

```r
NA
```

```
## [1] NA
```

The first two you already know from Python. What about the third one? It is R's way of saying something is not available. For example, assume you have missing data in your dataframe and you conduct a logical comparison such as whether a value is bigger than 3. If a value is missing, then the resulting comparison for that row cannot be `TRUE` or `FALSE` but `NA`.

By the way, to see the type of a vector, you can use the `typeof` method.

```
typeof(TRUE)
```

```
## [1] "logical"
```

2.1.2 Numeric

Next in line are the numeric vectors. These come in two flavors: integers and doubles.

```
typeof(2.0)
```

```
## [1] "double"
```

```
typeof(2)
```

```
## [1] "double"
```

As you can see, R stores every number as a double by default. This takes up a lot of memory so if you are sure you only need integers you can append the number with a 'L' to force the coercion to integer values.

```
typeof(2L)
```

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

Finally, note that while integers only have one type of `NA` value, doubles also have the values `Inf` and `-Inf`. You can test for missing values or infinite values by using the following functions:

- `is.finite()`
- `is.infinite()`
- `is.na()`

2.1.3 Character

Apart from the logical and numeric vectors there exist character vectors which allow you to store strings. These are created by using single or double quotes.

```
hal = "I'm sorry, Dave. I'm afraid I can't do that."
hal
```

```
## [1] "I'm sorry, Dave. I'm afraid I can't do that."
```

```
hal2 = 'I\'m sorry, Dave. I\'m afraid I can\'t do that.'
hal2
```

```
## [1] "I'm sorry, Dave. I'm afraid I can't do that."
```

2.2 Vector coercion

Just like Python, R features implicit as well as explicit coercion. You can do explicit coercion by using the following functions:

- `as.logical()`
- `as.integer()`
- `as.double()`
- `as.character()`

I will now give you some examples of implicit coercion:

If you pass a logical vector to a function that expects numeric vectors, it converts `FALSE` to 0 and `TRUE` and 1.

```r
TRUE + FALSE
```

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

You can also go implicitly from numerical to logical vectors. People sometimes use numeric vectors for logical conditions:

```r
x = 0
y = 2

if (x) {
  print('Hello')
  }
if (y) {
  print('World')
}
```

```
## [1] "World"
```

As you can see here, 0 gets converted into `FALSE` while every other value gets converted to `TRUE`.

To check whether a vector is of a specific type, use one of the following functions from the `purrr` package:

- `is_logical()`
- `is_integer()`
- `is_double()`
- `is_numeric()`
- `is_character()`
- `is_atomic()`
- `is_list()`
- `is_vector()`

2.3 Vectors with multiple values

So far we have only spoken if vectors with length 1. However, vectors can store several elements of the same type. You can create such vectors using the `c()` function. Let's check out strings first.

```r
my_strings = c('This', 'is', 'a', 'vector.')
my_strings
```

```
## [1] "This"    "is"      "a"       "vector."
```

```r
length(my_strings)
```

```
## [1] 4
```

```r
# strings do not get automatically combined if you use the print fuction
print(my_strings)
```

```
## [1] "This"    "is"      "a"       "vector."
```

```r
# you need to combine the strings first with a function such as paste()
print(paste(my_strings, collapse = ' '))
```

```
## [1] "This is a vector."
```

The numeric vectors come next.

```
my_numbers_1 = c(1, 2, 3, 4)
my_numbers_1
```

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

```
# a way of creating a numeric sequence
my_numbers_2 = 1:4
my_numbers_2
```

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

```
# another way with seq()
my_numbers_3 = seq(1, 10, 2)
my_numbers_3
```

```
## [1] 1 3 5 7 9
```

```
# creating vectors with a certain value with rep
my_zeros = rep(0, times = 7)
my_zeros
```

```
## [1] 0 0 0 0 0 0 0
```

```
my_zeros2 = double(length = 5)
my_zeros2
```

```
## [1] 0 0 0 0 0
```

```
my_ones  = rep(1L, 3)
my_ones
```

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

If you create a vector with values of different type, R will automatically fall back to the most comprehensive type.

```
c(TRUE, 1)
```

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

```
c(TRUE, 1, 'Hello')
```

```
## [1] "TRUE"  "1"     "Hello"
```

To check the length of a vector, use the `length` function. Note that when you apply to a character vector it still gives you the number of elements, not the number of characters.

```
length(c('Hallo'))
```

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

```
length(c(1, 2, 3))
```

```
## [1] 3
```

2.4 Operating with vectors and recycling

Notice that you can do the usual arithmetic operations with vectors. Addition, multiplication etc. with scalars works as you would probaly expect.

```
c(1, 2, 3) * 2
```

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

```
c(1, 2, 3) + 2
```

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

If you operate with two vectors of the same length, the operator is applied element-wise.

```
c(1, 2, 3) + c(1, 2, 3)
```

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

```
c(1, 2, 3) * c(1, 2, 3)
```

```
## [1] 1 4 9
```

What happens though if you multiply two vectors of different length?

```
c(1, 2, 3, 4) * c(1, 2)
```

```
## [1] 1 4 3 8
```

Since the second vector is shorter than the first one, it gets recycled, i.e. it gets copied until it has the same length as the first one.

```
c(1, 2, 3, 4) * c(1, 2, 1, 2)
```

```
## [1] 1 4 3 8
```

Note that R does not warn you about this as long as the longer vector is a multiple of the shorter one.

```
c(1, 2, 3, 4) * c(1, 2, 3)
```

```
## Warning in c(1, 2, 3, 4) * c(1, 2, 3): longer object length is not a multiple of
## shorter object length
```

```
## [1] 1 4 9 4
```

2.5 Names

If you want to name the elements of a vector, you can do so using base R or `purrr`.

```
# name elements during creation, note that you do not need quotes
my_named_vector = c(a = 1, b = 2, c = 3, d = 4)
my_named_vector
```

```
## a b c d
## 1 2 3 4
```

```
# use purrr to set the names after creation
my_named_vector = set_names(1:4, c('a', 'b', 'c', 'd'))
my_named_vector
```

```
## a b c d
## 1 2 3 4
```

2.6 Subsetting

There are basically three ways of subsetting a vector with the subsetting function `[]`.

1. You can pass a **numeric vector** containing the indices of the elements you are interested in.

```r
a = c('A', 'B', 'C', 'D')
a[c(1, 4)]
```

```
## [1] "A" "D"
```

```r
# negative indices for dropping elements
a[c(-1)]
```

```
## [1] "B" "C" "D"
```

```r
# duplicate elements by using the same index again
a[c(1, 1, 1)]
```

```
## [1] "A" "A" "A"
```

2. You can pass a **boolean vector** to select the elements which meet a certain criterion.

```r
a = 1:10
a < 5
```

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

```r
a[a < 5]
```

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

3. You can pass **the names of elements as a character vector** if you work with a named vector.

```r
a = c(a = 1, b = 2, c = 3, d = 4)
a[c('b', 'd')]
```

```
## b d
## 2 4
```

2.7 Lists

Lists differ from atomic vectors in that they can be heterogenous, i.e. store vectors or elements of different types and are recursive in that they can contain other lists.

2.7.1 Defining lists

To define a list, use the `list` function. To analyze its structure, you can apply the `str` function to it.

```r
# list with 4 scalar elements
my_list = list(1, 2, 3, 4)
str(my_list)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

```
# list with 1 column-vector element
my_list = list(c(1, 2, 3, 4))
str(my_list)
```

```
## List of 1
##  $ : num [1:4] 1 2 3 4
```

```
# list with a boolean, a number, and a string element
my_list = list(1, TRUE, 'Hello, hello')
str(my_list)
```

```
## List of 3
##  $ : num 1
##  $ : logi TRUE
##  $ : chr "Hello, hello"
```

```
# list of two lists
my_list = list(list(1, 2), list('a', 'b'))
str(my_list)
```

```
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ :List of 2
##   ..$ : chr "a"
##   ..$ : chr "b"
```

2.7.2 Subsetting lists

Before we learn about subsetting, let's create a fancy list which features sublists, names and heterogeneity.

```
fancy_list = list(a = 1:12, b = 'Pancakes are lovely, dear!', c = TRUE, d = list(-99, 1))
```

There are two subsetting functions which you can apply to lists, `[]` and `[[]]`:

1. **To get a sublist**: If you use `[]` with lists, you get a sublist. For example, if we enter '4' for the last element in our list, we get a list in return which contains the list named 'd'.

```
str(fancy_list[4])
```

```
## List of 1
##  $ d:List of 2
##   ..$ : num -99
##   ..$ : num 1
```

2. **To get the element itself**: If you use `[[]]`, you get the element itself. For example, if we enter '4' for the last element in our list, we get the list itself.

```
str(fancy_list[[4]])
```

```
## List of 2
##  $ : num -99
##  $ : num 1
```

If elements are named, you can get also get an element by using a shorthand involving the dollar sign.

```
str(fancy_list$d)
```

```
## List of 2
##  $ : num -99
##  $ : num 1
```

You can do more complicated subsetting operations by combining `[]` and `[[]]`. Be very careful though.

```
# returns the last element in a list
str(fancy_list[4])
```

```
## List of 1
##  $ d:List of 2
##   ..$ : num -99
##   ..$ : num 1
```

```
# returns the last element as itself
str(fancy_list[[4]])
```

```
## List of 2
##  $ : num -99
##  $ : num 1
```

```
# returns the first element of the last element as a list
str(fancy_list[[4]][1])
```

```
## List of 1
##  $ : num -99
```

```
# returns the first element of the last element as itself
str(fancy_list[[4]][[1]])
```

```
##  num -99
```

3 Checking cases with conditionals

Well, we know this stuff from Python already, but all of it looks slightly different in Python, so let's dive right into it.

3.1 `if- else`

An if-else structure is useful if you want to execute different code blocks depending on whether a certain statement is evaluated as `TRUE` or `FALSE`:

```
recession = TRUE
if (recession){
  print("Booh!")
} else {
  print("Yay!")
}
```

```
## [1] "Booh!"
```

In our case, the statement is very simple, it is just the value of our variable `recession`, which is `TRUE`. In this case, the code block is executed and R prints out 'Booh!' to the display. If the statement would have been evaluated as `FALSE`, the code within the else-bock would have been executed instead. Notice the difference to the Python code we discussed:

1. Statements have to surrounded by round brackets.

2. We have to use curly brackets to delimit our code blocks. This problem is solved by colon + indentation in Python.

3.2 `if - else if - else`

If we want to check more than case, we can use the `if - else if - else` structure.

```
color = 'violet'
if (color == 'red') {
  print('It is a tomatoe!')
} else if (color == 'yellow') {
  print('It is a yellow pepper!')
} else if (color == 'violet') {
  print('It is an onion!')
} else {
  print('No idea what this is!')
}
```

```
## [1] "It is an onion!"
```

R checks each of the provided statements and executes the block of the first statement that is evaluated as true.

3.3 Checking multiple cases with `switch`

If you want to pass options or have a lot of conditions to check, you can use the `switch` function.

```
x = 1
y = 2
operation = 'plus'

switch(operation,
       plus = x + y,
       minus = x - y,
       times = x * y,
       divide = x / y,
       stop('You specified an unknown operation!')
       )
```

```
## [1] 3
```

3.4 Comparison and logical operators

We have already used the `==` operator which checks for equality between two values. You can also use the `identical` function. This makes sure that only one truth-value is returned. If more than one value is returned, say from a comparison of vectors with `==`, R will check only the first truth-value.

```
A = c(1, 2, 3, 4)
B = c(1, 3, 4, 5)

if (A == B) {
  print('They are equal!')
} else {
```

```r
  print('They are not equal!')
}
```

```
## Warning in if (A == B) {: the condition has length > 1 and only the first
## element will be used
```

```
## [1] "They are equal!"
```

```r
if (identical(A, B)) {
  print('They are equal!')
} else {
  print('They are not equal!')
}
```

```
## [1] "They are not equal!"
```

A disadvantage of `identical` is that you have to be very specific regarding types:

```r
identical(0L, 0)
```

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

You also have the following other operators for logical comparisons:

- `!=`: not identical,
- `<`: smaller than,
- `<=`: smaller than or equal,
- `>`: bigger than,
- `>=`: bigger than or equal,
- `!`: not,
- `&&`: logical 'and',
- `||`: logical 'or',
- `is.logical` etc.

Finally, note that the use of doubles in logical comparisons can be dangerous.

```r
1 - 1/3 - 1/3 - 1/3 == 0
```

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

R exhibits this strange behavior because there are always approximation errors when using floating point numbers.

```r
1 - 1/3 - 1/3 - 1/3
```

```
## [1] 1.110223e-16
```

You can use the `near` function from the `dplyr` package to account for these cases.

```r
dplyr::near(1 - 1/3 - 1/3 - 1/3, 0)
```

```
## [1] TRUE
```

4 Functions

You can of course also write functions in R. We will start by learning how to define them.

4.1 Function definitions

To see how to define a function, let's just write one.

```
calc_percent_missing = function(x){
  mean(is.na(x))
  }
calc_percent_missing(c(1, 2, 6, 3, 7, NA, 9, NA, NA, 1))
```

```
## [1] 0.3
```

You can see that we need three things to define a function:

1. a function name, in this case it's `calc_percent_missing`,
2. function arguments, in this case it's just one, the vector `x`,
3. the function body which is enclosed by curly parentheses.

Note that the last statement that is evaluated in the function body is automatically taken as the return value of the function.

4.2 Function arguments and default values

Function arguments in R can usually be broadly divided into two categories:

- **data**: either a dataframe or a vector,
- **details**: parameters which govern the computation.

For the latter you often want to define default values. You can do this just as in Python and we will look at our old friend, the Cobb-Douglas utility function.

```
cobb_douglas = function(x, a = 0.5, b = 0.5) {
  u = x[1]**a * x[2]**b
}
x = c(1, 2)
print(cobb_douglas(x))
```

```
## [1] 1.414214
```

```
print(cobb_douglas(x, b = 0.4, a = 0.6))
```

```
## [1] 1.319508
```

Note that I not only overwrote the default values but also changed the order of the arguments. Similarly to Python, you can change the order of arguments if you call the arguments by their keywords.

4.3 Arbitrary number of arguments

Sometimes you might want to write a function which takes an arbitrary number of arguments. You can do this with the dot-dot-dot argument. I demonstrate its usefulness with a nice little function by Hadley Wickham. Note that `str_c` is a function to combine strings into a single one.

```
commas <- function(...) {
  stringr::str_c(..., collapse = ", ")
}
commas(letters[1:10])
```

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

4.3 Function returns

I already told you that a function automatically returns the value of the last statement evaluated. You can however also be explicit about it by using the `return` function.

4.4 pipes

Piping is a practice that we will often use in this course to quickly chain a lot of functions together. I will tell you more about it later. For now, we only have to know under which conditions functions are pipeable.

1. A function is pipeable if you pass an object to it and the function returns a modified version of the object. An example of a pipeable function would be one to which you pass a numeric vector and it returns it in a sorted form.
2. A function is also pipeable if we pass an object to a function and the function returns it unmodified but creates what R programmers such as Hadley Wickham call a side-effect in the meantime. Such side-effects could be the drawing of a plot or the display of basic information about the object that is passed to the function. I will again illustrate this with a useful function by Wickham. We could program our function `calc_percent_missing` from above in such a way.

```r
calc_percent_missing = function(x) {
  n = mean(is.na(x))
  cat("Percent missing: ", n*100, '!', sep='')
  invisible(x)
}
test_vector = c(1, 2, 3, NA, 6, 9, NA, NA, NA, 10)
calc_percent_missing(test_vector)
```

```
## Percent missing: 40!
```

As you can see, this function displays the percentage of missing values on the screen. It still returns the unchanged vector `x`, although it does not print it out.

5 Iterations

Now, after we heard about functions and conditionals it is time to think about iterations again. We will first look at the kind of loops we also know from Python and then briefly look at the `map` function family.

Before we actually start, let's briefly create a dataframe or – more specifically – a tibble.

```r
# letters denote variable names
my_df = tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

You will notice that you have now a new dataframe in your workspace, `my_df`. We will work with it in the following.

5.1 `for` loops

5.1.1 Looping over numeric indices

Let's say you want to calculate the median for each of the columns in our dataframe. You can of course do this by hand.

```
median(my_df$a)
```

```
## [1] 0.2263692
```

```
median(my_df$b)
```

```
## [1] -0.2379822
```

```
median(my_df$c)
```

```
## [1] -0.142334
```

```
median(my_df$d)
```

```
## [1] -0.0006664581
```

This looks like a lot of repetition though, right? Maybe we can loop over the different columns.

```
output = vector("double", ncol(my_df))
for (i in seq_along(my_df)) {
  output[i] = median(my_df[[i]])
}
output
```

```
## [1]  0.2263692083 -0.2379822450 -0.1423339738 -0.0006664581
```

The first line initializes an output vector. Make sure to always pre-allocate space for your output. In this case, we have pre-specified the length of the double vector to be equal to the number of columns in our dataframe.

The second line contains the looping statement. The `seq_along` function acts as a safe alternative to `length` here. Why is it safer your ask?

```
test_vector = vector("double", 0)
seq_along(test_vector)
```

```
## integer(0)
```

```
1:length(test_vector)
```

```
## [1] 1 0
```

The body of the loop calculates the median for each of the dataframe columns. Note also the use of the subsetting operators here. Why did I use `[[]]` for subsetting the dataframe?

5.1.2 Looping over elements

You can also write loops to iterate over elements. This is particularly useful in those cases where you do not want to store output.

```
for (x in my_df) {
  print(median(x))
}
```

```
## [1] 0.2263692
## [1] -0.2379822
## [1] -0.142334
## [1] -0.0006664581
```

In this case, I did not store the median values anywhere but just displayed them on the screen. Note that it would have been difficult to use this kind of loop for storing the elements in an initialized output vector as there is no natural way to index an output vector.

5.1.3 Looping over names

Finally, note that you can also loop over the names of a vector.

```r
for (name in names(my_df)) {
  cat("Median for variable ", name, " is: ", median(my_df[[name]]), '\n', sep='')
}
```

```
## Median for variable a is: 0.2263692
## Median for variable b is: -0.2379822
## Median for variable c is: -0.142334
## Median for variable d is: -0.0006664581
```

5.2 Repeat until with `with`

Naturally, R also has a `while` structure which is particularly useful if you do not know how long a particular piece of code should be run. We had a longer discussion of this concept in the Python class, I will only provide one brief example here. The following loop prints out the square of a number that is provided by a user. If 'q' is entered, R leaves the loop.

```r
#while (TRUE) {
#  n <- readline(prompt="Enter an integer (quit with q): ")
#  if (n == 'q') {
#    break
#  } else {
#    cat(as.numeric(n), ' squared is ', as.numeric(n)**2, '!', sep = '')
#  }
#}
```

6 Split, apply, combine with `map` functions

There is one cool feature which I briefly want to tell you about. Very often we want to loop over a vector, do something to each element of the vector and then save the results. You could also summarize this as split - apply - combine. The `purrr` package provides the functionalities for such operations in a handy manner. Let's start with a basic example.

6.1 Basic example

To see how it works let's get back to a simple dataframe. To remind you (and me), I create a new one here. Note that the `b` column contains a missing value.

```r
my_df = tibble::tibble(
  a = rnorm(10),
  b = c(rnorm(9), NA),
  c = rnorm(10),
  d = rnorm(10)
)
```

Let's assume that we want to have the median of every column again. We already created a loop to do just that. There is a simpler way though. We can also use a `map` function to do this in a very compact manner.

```r
map_dbl(my_df, median)
```

```
##          a          b          c          d 
##  0.1572301         NA  0.5165605 -0.3613687
```

We can compute the mean and standard deviation in a similar way.

```r
map_dbl(my_df, mean)
```

```
##          a          b          c          d 
##  0.3734060         NA  0.4054269 -0.1132955
```

```r
map_dbl(my_df, sd)
```

```
##         a         b         c         d 
## 0.7992085        NA 0.9109602 0.7765902
```

Pretty convenient, right?

6.2 Syntax

While there are several `map` functions they all have the same argument structure:

1. You have to pass the **vector / list** on whose elements you want to operate on.
2. You have to pass the **function** which you want to apply to each element.

You can also pass additional arguments to the map functions. Say we want to compute the mean for our `b` column as well.

```r
map_dbl(my_df, mean, na.rm = TRUE)
```

```
##           a           b           c           d 
##  0.37340596  0.05296683  0.40542689 -0.11329553
```

Finally, I already told you that there are several map functions, but so far we have only used the `map_dbl` function. These are the others:

- `map()`: makes a list,
- `map_lgl()`: makes a logical vector,
- `map_int()`: makes an integer vector,
- `map_dbl()`: makes a double vector,
- `map_chr()`: makes a character vector.

The function choice depends on the output you expect. If we apply the `mean` to each element (i.e. column) of our dataframe, we usually expect to obtain a double vector that contains the means. But if we want a vector of strings, we can also do that.

```r
map_chr(my_df, mean, na.rm = TRUE)
```

```
##           a           b           c           d 
##  "0.373406"  "0.052967"  "0.405427" "-0.113296"
```

6.3 map2 and pmap

There are some extensions to the basic `map` functionality.

For example, what do we have to do if we want to iterate over two lists? We can use the `map2` function. Say we want to create a new dataframe but this time not all variables are suppose to have the same mean and standard deviation. Let's create two vectors with our desired means and standard deviations.

```
mu = list(0, 0, 0, 0)
sd = list(1, 5, 10, 20)

my_fancy_df = map2(mu, sd, rnorm, n = 20)
my_fancy_df = set_names(my_fancy_df, c('a', 'b', 'c', 'd'))
my_fancy_df = tibble::as_tibble(my_fancy_df)

# look at the head of the dataframe
head(my_fancy_df)
```

```
## # A tibble: 6 x 4
##         a      b       c       d
##     <dbl>  <dbl>   <dbl>   <dbl>
## 1 -1.59   -5.33    8.69  -42.9
## 2 -0.698  -5.81  -13.8    -0.391
## 3 -1.60    1.55    8.17  -12.8
## 4  0.0515 -0.274  -5.51   19.9
## 5  0.364  -5.74    0.882  42.2
## 6 -0.600   1.81    3.85  -17.3
```

OK. So what should we do if we want to iterate over more than two lists? Say we want to create vectors with different lengths, means and standard deviations. We could use `pmap`.

```
mu = list(1, 2, 3, 4)
sd = list(1, 10, 15, 20)
n  = list(1, 2, 3, 4)

# enter arguments in the right order
args = list(n, mu, sd)

str(pmap(args, rnorm))
```

```
## List of 4
##  $ : num 0.3
##  $ : num [1:2] 5.815 -0.536
##  $ : num [1:3] -0.889 12.302 -3.087
##  $ : num [1:4] 45.74 -19.37 3.86 7.22
```

And this is how our brief tour of R programming ends.

Sources

These notes were created by Julian Langer for the 2019 edition of PP4RS (https://pp4rs.github.io/2019-foundations-uzh/) and have only been minorly updated. His exposition is heavily inspired by the notes for a new book on R data science by Garrett Grolemund and Hadley Wickham. You can find detailed outlines here: http://r4ds.had.co.nz.