R building blocks

What is a programming language?

R is a programming language. You probably have heard of the term - a programming language is a set of defined instructions or “rules” that a computer can execute.

How is R different from other programming languages?

R is a friendly language that can be run directly (once installed, of course) without any further steps and does not require you to be finished with the script in order to run chunks of it. Perfect for data analysis or going back and forward in your code.

R is also **very** readable and its abstraction level is high. We humans love and benefit from that. We can easily tell R to “sum”, “plot”, or “print”. This might look like obvious instructions but these simple operations would take a good chunk of code in other languages.

What is R famous for?

R was designed with the purpose of having a free and open-sourced (open for collaborations) programming language focused specifically in statistical analyses.

The reason is so widely used is precisely that - because it is free and people have made it “theirs” by implementing functions specific to quite niche fields - such as many in bioinformatics.

R shines in managing big data, analysing it, and very importantly - visualising it.

The purpose of this course

We believe that everyone should be able to explore their own data. At the end of this course, we hope you will be able to manage your data to get informative summaries, and find the *perfect* way to make that plot fit into your figure - all without having to go 10 different times to your local bioinformatician :-).

Most importantly, this course should give you a taste for programming and what R is able to offer.

Introduction to R

Variables

The most basic operation you can do in any programming language is to define a **variable**. A variable (same as in maths) is a "name" that we give to some piece of information and for which R will allocate a place in memory for. In R, there are two ways of assigning a variable: one is the same we usually do in maths `x = 100`; and the other one is `x <- 100` which is meant to provide some sense of direction, meaning "this piece of information (e.g. `100`) is going to be assigned into this variable name (e.g. `x`), and not vice versa". The most used and recommended form for it in R is the arrow operand `<-`.

Variable names are meant to be short but informative and comply with the language's limitations. It is not a trivial task sometimes but is *very* important to ensure readability in your code. In R, variable names have to start with a letter (no numbers, no other characters) and can be a combination of letters, digits, period (.) and underscores (_) - any other special characters are not allowed.

There are also forbidden words when it comes to variable names. These words are reserved for the syntax of R and will be confusing (or simply throw an error) when used. Examples of these include "if", "for", "while", "break", "TRUE", "NULL", "Inf", and many more. You can see a list of all reserved words by typing `?reserved` into the R command prompt. In addition to it, it is **highly** recommended to **never** use variable names that are also a function as this can cause a conflict when executing (and reading) your code.

Variable classes

R classifies information by type (or class) to know what sort of operations can be performed with your variables. The most basic classes of variables are numbers, characters (including those with multiple characters such as "hello world"), factors (discrete but with multiple options. e.g. `groupA`, `groupB` and `groupC`), and booleans (`TRUE` or `FALSE`).

You can easily interchange between some of these classes using `as.numeric(your variable here)`, `as.character(your variable here)`, and `as.factor(your variable here)` (although, sometimes not directly).

One thing we have to make peace with when programming, is that the computer is completely oblivious to what we are doing. It just executes what you write. For example, if R reads a "1" (surrounded by quotes), it will assume it is of a "character" class - although we might think it is obviously a number. This is a common issue specially when reading from files. We have to make sure that we are in agreement with R at all time. For this, the `class()` or `typeof()` functions come in handy.

```r
a <- "1"
paste("Here is a number: ", a, " of type: ", class(a), sep="")
```

```
## [1] "Here is a number: 1 of type: character"
```

```r
b <- as.numeric(a)
paste("Whoops, here it goes again: ", b, " as type: ", class(b), sep="")
```

```
## [1] "Whoops, here it goes again: 1 as type: numeric"
```

```r
c <- 2
paste("Here goes another number: ", c,
      ", which I proudly declared correctly as type: ", class(c), sep="")
```

```
## [1] "Here goes another number: 2, which I proudly declared correctly as type: numeric"
```

We say "function" to refer to a set of defined instructions which usually takes an input, performs an operation with it and (usually) returns an output. In this material, you can assume we are talking about a function when we write parentheses following a word. For example, `as.numeric()` is a function that takes a variable (input), in this case `a`, converts it into a number (performs an operation), and returns a number (output), which is now stored in `b`.

One dimensional data structures

Or "collections". One can define two types of one-dimensional collections in R: one is called a *vector* and the other one is a *list*. They differ quite a bit on how to access them and the sort of operations one can do with the items within them. But there are no rules on what you can put into them - they can store from simple numbers to complex objects.

You can define a vector using the function `c()`, which stands for "combine", and a list using the function `list()`.

Here are some basic operations one can do with a vector:

```r
# You can define a vector by doing
a_vector <- c(1,2,3)
# You can concatenate new items by doing
a_vector <- c(a_vector, "Hello world!", TRUE)

# Print it simply by
print(paste("This vectors has many items:",
            paste(a_vector, collapse = ", ")))
```

```
## [1] "This vectors has many items: 1, 2, 3, Hello world!, TRUE"
```

Access specific items using the brackets `[]`. R is a 1-indexed language contrary to other 0-indexed languages. This means that the first item in the vector is indexed as 1 and can be retrieved by placing 1 within the brackets.

```r
paste("The second item of the vector is: ", a_vector[2])
```

```
## [1] "The second item of the vector is:  2"
```

Access a slice of the vector by using the brackets. You will need the position where to start, a colon, and the position where to end.

```r
print(paste("The items from the position 2 to the 4th are: ",
            paste(a_vector[2:4], collapse = ", ")))
```

```
## [1] "The items from the position 2 to the 4th are:  2, 3, Hello world!"
```

Lists are created and managed similarly (but not equally) to vectors:

```r
# Define a list
a_list <- list(1,2,3)
# Define another list
another_list <- list("Hello world!", TRUE)
# Concatenate two lists to append items
final_list <- append(a_list, another_list)

# Print
print(paste("This list has many items:",
            paste(final_list, collapse = ", ")))
```

```
## [1] "This list has many items: 1, 2, 3, Hello world!, TRUE"
```

Access the second item or a slice

```r
paste("The second item of the list is: ", final_list[2])
```

```
## [1] "The second item of the list is:  2"
```

```r
print(paste("The items from the position 2 to the 4th are: ",
      paste(final_list[2:4], collapse = ", ")))
```

```
## [1] "The items from the position 2 to the 4th are:  2, 3, Hello world!"
```

Vectors and lists can sometimes can get long and confusing. Therefore, we can name the items in them to make it easier to access them later. This is done simply by passing the names you want to give to your items to `names()`:

```r
names(a_vector) <- c("my_fav_number",
                     "my_second_fav_number",
                     "my_third_fav_number",
                     "what_i_want_to_say",
                     "im_learning_so_much")
a_vector
```

```
##        my_fav_number my_second_fav_number  my_third_fav_number
##                  "1"                  "2"                  "3"
##   what_i_want_to_say  im_learning_so_much
##       "Hello world!"               "TRUE"
```

You can also define them named from the beginning by using the form [name] = [value]:

```r
languages <- c("Sweden" = "Swedish",
               "USA" = "English",
               "Mexico" = "Spanish",
               "Brazil" = "Portuguese",
               "Italy" = "Italian")
languages
```

```
##       Sweden          USA       Mexico       Brazil        Italy
##    "Swedish"    "English"    "Spanish" "Portuguese"    "Italian"
```

We can then access them quite easily!

```r
paste("In", names(languages["Mexico"]), "they speak", languages["Mexico"])
```

```
## [1] "In Mexico they speak Spanish"
```

Same with lists:

```r
names(final_list) <- c("my_fav_number",
                       "my_second_fav_number",
                       "my_third_fav_number",
                       "what_i_want_to_say",
                       "im_learning_so_much")
final_list
```

```
## $my_fav_number
## [1] 1
##
## $my_second_fav_number
## [1] 2
##
## $my_third_fav_number
## [1] 3
##
## $what_i_want_to_say
## [1] "Hello world!"
##
## $im_learning_so_much
## [1] TRUE
```

Lists can also be defined with names:

```r
favourite_foods <- list("Raquel" = "Pasta",
                       "Anita" = "Pizza",
                       "David" = "Burgers",
                       "Vivien" = "Ramen")
favourite_foods
```

```
## $Raquel
## [1] "Pasta"
##
## $Anita
## [1] "Pizza"
##
## $David
## [1] "Burgers"
##
## $Vivien
## [1] "Ramen"
```

Did you notice that in the output the names start with a $ in the output? That is because, to access any favorite food, you can do:

```r
paste("Vivien's favourite food is: ", favourite_foods$Vivien)
```

```
## [1] "Vivien's favourite food is:  Ramen"
```

Or using double brackets:

```r
paste("Raquel's favourite food is: ", favourite_foods[["Raquel"]])
```

```
## [1] "Raquel's favourite food is:  Pasta"
```

An important function to remember is `length()` which will tell you how long a vector or a list is.

```r
paste("We have registered", length(favourite_foods), "favourite foods")
```

```
## [1] "We have registered 4 favourite foods"
```

Another is `table()`, which will summarize how many times a certain value is in a vector (not for lists).

```r
true_or_false <- c(TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE,
                   TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE)
table(true_or_false)
```

```
## true_or_false
## FALSE  TRUE
##     8     9
```

Two-dimensional data structures

If you are wondering if you can make a collection of collections, the answer is yes! These are called two-dimensional data structures because they can be represented as a table.

Lists can contain vectors but vectors cannot contain lists.

```r
list_of_vectors <- list("some_numbers" = c("ten" = 10, "twenty_four" = 24, "thirty_six" = 36),
                        "some_names" = c("a" = "anita", "b" = "ben", "c" = "chris"))

print("My list of vectors:")
```

```
## [1] "My list of vectors:"
```

```r
list_of_vectors
```

```
## $some_numbers
##         ten twenty_four  thirty_six
##          10          24          36
##
## $some_names
##       a       b       c
## "anita"   "ben" "chris"
```

You can access a specific item by using a combination of the access operands, such as:

```
paste("First number of first vector:", list_of_vectors$some_numbers["ten"])
```

```
## [1] "First number of first vector: 10"
```

```
paste("Second name in my second vector:", list_of_vectors[["some_names"]]["a"])
```

```
## [1] "Second name in my second vector: anita"
```

Collections keep their numeric indexes so you can also do:

```
paste("Third name in my second vector:", list_of_vectors[[2]][3])
```

```
## [1] "Third name in my second vector: chris"
```

```
# Or
paste("Third name in my second vector:", list_of_vectors$some_names[3])
```

```
## [1] "Third name in my second vector: chris"
```

```
# It is all the same :-)
```

A cool couple of functions that will help you out if you forget what a variable contains are `str()` (from structure) and `summary()`.

```
print("Str says:")
```

```
## [1] "Str says:"
```

```
str(list_of_vectors)
```

```
## List of 2
##  $ some_numbers: Named num [1:3] 10 24 36
##   ..- attr(*, "names")= chr [1:3] "ten" "twenty_four" "thirty_six"
##  $ some_names  : Named chr [1:3] "anita" "ben" "chris"
##   ..- attr(*, "names")= chr [1:3] "a" "b" "c"
```

```
print("Summary says:")
```

```
## [1] "Summary says:"
```

```
summary(list_of_vectors)
```

```
##              Length Class  Mode
## some_numbers 3      -none- numeric
## some_names   3      -none- character
```

As you can see, it is a quick recap of what this variable is. In this case, `str()` will tell you that I passed a list of length 2 (2 vectors), named "some_numbers" - a named numeric vector - and "some_names" - a named char (character) vector. It also tells you their values and their "attr" (short for *attributes* - a defining characteristic of a variable, in this case their names!).

The output of `summary()` depends on what type of variable you have. For a list it is a bit of a shortened version of `str()`. It simply lists what is contained in it, the length of each of those things, and their class.

If you would ask for a summary of a numeric vector, it will give you some information about the distribution of the contained values.

```
numeric_vector <- c(14, 50, 23, 43, 100, 2)
summary(numeric_vector)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00   16.25   33.00   38.67   48.25  100.00
```

It is a neat function to explore different variable types!

Matrices and data frames

Lists and vectors are all ok, but it would become quite annoying if these would be super long. Right? This is where *matrices* and *data frames* come in handy.

Both of these data types can be visualized as a table or grid - like an excel file. Let's load one, but first we need to load the xlsx *library*. A library is a bunch of code written by other people that helps with a particular purpose - in this case reading and writing to excel files. We are going to read the file (specifying the sheet index number - in this case the first sheet), and print the "head" (first 6 entries without counting the header) of the file.

```
library(xlsx)
coldata <- read.xlsx("/Volumes/MyPassport/courses/basic_R/metadata_AA.xlsx", sheetIndex = 1)
head(coldata)
```

```
##   sample cell_line   organism type cnodition seqeuncgni_batch
## 1  AA112       pt6 chimpanzee ipsc        g1                1
## 2  AA113       pt6 chimpanzee ipsc        g1                1
## 3  AA114       pt6 chimpanzee ipsc        g1                1
## 4  AA115       pt6 chimpanzee ipsc        g1                2
## 5  AA116       pt6 chimpanzee ipsc        g1                2
## 6  AA117       pt6 chimpanzee ipsc        g2                1
```

A quick note about paths…

If this is the first time you have seen the name of a file written like this, you are probably wondering why we wrote it like this, and why do we need to specify such a long name. A path is simply the location in our system of a particular file. The path in the exercise is called an **absolute path**. This means that it is the most complete address you can give from the file in question. In this case we are looking for the file in (from right to left) a directory called `basic_R`, which is inside the `courses` directory, on `MyPassport` - which is an external hard drive that you can find in `Volumes`. Our file system starts at root (`/`), so all absolute paths start from there.

We can freely move in our file system to list and create files. To check your current location in your system you can do `getwd()` in R - which stands for "get working directory". Similarly, you can set your working directory to somewhere convenient like your desktop:

```
setwd("~/Desktop/")
```

You can then access files stored in your desktop without having to specify the absolute path. For example, if you stored your metadata sheet in your desktop, you could now do `read.xlsx("metadata_AA.xlsx", sheetIndex = 1)` instead.

RStudio is pretty handy and can auto-complete paths for you (to aviod typos). If you open quotes and then press tab, it will list the files or directories in your current working directory.

Back to the exercise…

`read.xlsx()` will automatically load the contents of the excel file in a data frame. Let's see what sort of variables it contains

```
str(coldata)
```

```
## 'data.frame':    75 obs. of  6 variables:
##  $ sample          : chr  "AA112" "AA113" "AA114" "AA115" ...
##  $ cell_line       : chr  "pt6" "pt6" "pt6" "pt6" ...
##  $ organism        : chr  "chimpanzee" "chimpanzee" "chimpanzee" "chimpanzee" ...
##  $ type            : chr  "ipsc" "ipsc" "ipsc" "ipsc" ...
##  $ cnodition       : chr  "g1" "g1" "g1" "g1" ...
##  $ seqeuncgni_batch: num  1 1 1 2 2 1 1 1 2 2 ...
```

This is a metadata excel file from a PhD student (we will call him Steve). It lists all samples in his experiment and some relevant information about them.

Steve had some trouble when typing the name of the last column, so we need to fix that for him… To do this, we will make use of the function `colnames()`, which will return the attribute of column names in the data frame.

```
colnames(coldata)
```

```
## [1] "sample"           "cell_line"        "organism"         "type"
## [5] "cnodition"        "seqeuncgni_batch"
```

We can edit the contents of this attribute, by assigning it a new value. You can either pass the entire vector with new column names, or specify for exactly which column we need to change the attribute for (in this case columns 5 and 6)

```
colnames(coldata) <- c("sample",
                       "cell_line",
                       "organism",
                       "type",
                       "condition",
                       "sequencing_batch")
head(coldata)
```

```
##   sample cell_line   organism type condition sequencing_batch
## 1  AA112       pt6 chimpanzee ipsc        g1                1
## 2  AA113       pt6 chimpanzee ipsc        g1                1
## 3  AA114       pt6 chimpanzee ipsc        g1                1
## 4  AA115       pt6 chimpanzee ipsc        g1                2
## 5  AA116       pt6 chimpanzee ipsc        g1                2
## 6  AA117       pt6 chimpanzee ipsc        g2                1
```

Much better.

Now we can extract some information about these, since I am Steve's bioinformatician but don't know much about his samples…

```
paste("Steve sequenced", nrow(coldata), "samples")
```

```
## [1] "Steve sequenced 75 samples"
```

```
# nrow as in "number of rows" (ncol for number of columns)
```

One can access each row or column by specifying it on brackets like: [row index number or name, column index number or name]. If you keep one of these brackets-slots empty, it will return all. As an example, `dataframe[1,]` will return first row, all columns.

```
print("Sample names:")
```

```
## [1] "Sample names:"
```

```
coldata[,1] # All sample names
```

```
##  [1] "AA112" "AA113" "AA114" "AA115" "AA116" "AA117" "AA118" "AA119" "AA120"
## [10] "AA121" "AA122" "AA123" "AA124" "AA125" "AA126" "AA127" "AA128" "AA129"
## [19] "AA130" "AA131" "AA132" "AA133" "AA134" "AA135" "AA136" "AA137" "AA138"
## [28] "AA139" "AA140" "AA141" "AA142" "AA143" "AA144" "AA145" "AA146" "AA147"
## [37] "AA148" "AA149" "AA150" "AA151" "AA152" "AA153" "AA154" "AA155" "AA156"
## [46] "AA157" "AA158" "AA159" "AA160" "AA161" "AA162" "AA163" "AA164" "AA165"
## [55] "AA166" "AA167" "AA168" "AA169" "AA170" "AA171" "AA172" "AA173" "AA174"
## [64] "AA175" "AA176" "AA177" "AA178" "AA179" "AA180" "AA181" "AA182" "AA183"
## [73] "AA184" "AA185" "AA186"
```

```
print("Sample names:")
```

```
## [1] "Sample names:"
```

```
coldata[,"sample"] # Same
```

```
##  [1] "AA112" "AA113" "AA114" "AA115" "AA116" "AA117" "AA118" "AA119" "AA120"
## [10] "AA121" "AA122" "AA123" "AA124" "AA125" "AA126" "AA127" "AA128" "AA129"
## [19] "AA130" "AA131" "AA132" "AA133" "AA134" "AA135" "AA136" "AA137" "AA138"
## [28] "AA139" "AA140" "AA141" "AA142" "AA143" "AA144" "AA145" "AA146" "AA147"
## [37] "AA148" "AA149" "AA150" "AA151" "AA152" "AA153" "AA154" "AA155" "AA156"
## [46] "AA157" "AA158" "AA159" "AA160" "AA161" "AA162" "AA163" "AA164" "AA165"
## [55] "AA166" "AA167" "AA168" "AA169" "AA170" "AA171" "AA172" "AA173" "AA174"
## [64] "AA175" "AA176" "AA177" "AA178" "AA179" "AA180" "AA181" "AA182" "AA183"
## [73] "AA184" "AA185" "AA186"
```

Alternatively, one can access columns (but not rows) as a named list:

```
print("Sample names:")
```

```
## [1] "Sample names:"
```

```
coldata$sample
```

```
##  [1] "AA112" "AA113" "AA114" "AA115" "AA116" "AA117" "AA118" "AA119" "AA120"
## [10] "AA121" "AA122" "AA123" "AA124" "AA125" "AA126" "AA127" "AA128" "AA129"
## [19] "AA130" "AA131" "AA132" "AA133" "AA134" "AA135" "AA136" "AA137" "AA138"
## [28] "AA139" "AA140" "AA141" "AA142" "AA143" "AA144" "AA145" "AA146" "AA147"
## [37] "AA148" "AA149" "AA150" "AA151" "AA152" "AA153" "AA154" "AA155" "AA156"
## [46] "AA157" "AA158" "AA159" "AA160" "AA161" "AA162" "AA163" "AA164" "AA165"
## [55] "AA166" "AA167" "AA168" "AA169" "AA170" "AA171" "AA172" "AA173" "AA174"
## [64] "AA175" "AA176" "AA177" "AA178" "AA179" "AA180" "AA181" "AA182" "AA183"
## [73] "AA184" "AA185" "AA186"
```

Let's name the rows per sample:

```
rownames(coldata) <- coldata$sample
```

Now we can access information per sample simply by doing:

```
coldata["AA121",]
```

```
##       sample cell_line   organism type condition sequencing_batch
## AA121  AA121       pt6 chimpanzee ipsc        g2                2
```

When one extracts a row or a column from a data frame, it is important to note that `$` or `[x,y]` will return a vector.

Can you find out how many samples we have per organism? (Remember `table()`? It can take more than one vector - in this case the values of a column!).

The output of `table()` behaves like a named vector - although it is its own class "table" (you can verify this by doing `class(table(coldata$cell_line))`). We can filter it as if it were a vector, for example, to keep only the `unique()` values:

```r
table(coldata$cell_line)
```

```
##
##    h48      h6     joc     pt6 sandra
##     15      15      15      15      15
```

```r
paste(unique(table(coldata$cell_line)), "samples per cell line!")
```

```
## [1] "15 samples per cell line!"
```

We can also extract a certain value out of a table the same way we would do to extract something of a named vector:

```r
paste("We have", table(coldata$organism)["human"], "human samples!")
```

```
## [1] "We have 30 human samples!"
```

Can you find out how many conditions we have in this experiment?

```r
unique_conditions <- unique(coldata$condition)

paste("There are", length(unique_conditions), "unique conditions")
```

```
## [1] "There are 3 unique conditions"
```

```r
table(coldata$cell_line, coldata$condition)
```

```
##
##          g1 g2 lacz
##   h48     5  5    5
##   h6      5  5    5
##   joc     5  5    5
##   pt6     5  5    5
##   sandra  5  5    5
```

```r
print("Each of them have five samples per cell line. Good job Steve.")
```

```
## [1] "Each of them have five samples per cell line. Good job Steve."
```

Basic operations

R has a bunch of "built-in" functions (operations you can make use of right after installation without the needing to install anything else). Here are a few basic examples of very handy functions:

Arithmetics

- Sum (`sum()`)
- Round (`round()`)
- Round down (`floor()`)
- Round up (`ceiling()`)
- Absolute values (`abs()`)
- Minimum (`min()`)
- Maximum (`max()`)
- Log (`log()`, `log2()`, or `log10()`)
- Exponential (`exp()`)

```
cat(paste("We will use our numeric_vector for these examples.\nHere are the values:",
            paste(numeric_vector, collapse = ", ")))
```

```
## We will use our numeric_vector for these examples.
## Here are the values: 14, 50, 23, 43, 100, 2
```

```
print(paste("The total sum of the numeric_vector is:", sum(numeric_vector)))
```

```
## [1] "The total sum of the numeric_vector is: 232"
```

```
print(paste("The largest number in the array is:", max(numeric_vector)))
```

```
## [1] "The largest number in the array is: 100"
```

```
print(paste("The smallest number in the array is:", min(numeric_vector)))
```

```
## [1] "The smallest number in the array is: 2"
```

```
print(paste("The log2 of the largest number is:", log2(max(numeric_vector))))
```

```
## [1] "The log2 of the largest number is: 6.64385618977472"
```

```
print(paste("Let's round that number to:", round(log2(max(numeric_vector)))))
```

```
## [1] "Let’s round that number to: 7"
```

```
print(paste("Or being modest:", floor(log2(max(numeric_vector)))))
```

```
## [1] "Or being modest: 6"
```

```r
print("We can also check the log2 of the entire array:")
```

```
## [1] "We can also check the log2 of the entire array:"
```

```r
log2(numeric_vector)
```

```
## [1] 3.807355 5.643856 4.523562 5.426265 6.643856 1.000000
```

```r
print("What about the exponential?")
```

```
## [1] "What about the exponential?"
```

```r
exp(numeric_vector)
```

```
## [1] 1.202604e+06 5.184706e+21 9.744803e+09 4.727839e+18 2.688117e+43
## [6] 7.389056e+00
```

Basic statistics

- Mean (`mean()`)
- Median (`median()`)
- Standard deviation (`sd()`)
- Quantiles (`quantile()`)

```r
print(paste("The mean of the array is:", mean(numeric_vector)))
```

```
## [1] "The mean of the array is: 38.6666666666667"
```

```r
print(paste("The median is:", median(numeric_vector)))
```

```
## [1] "The median is: 33"
```

```r
print(paste("The standard deviation is:", median(numeric_vector)))
```

```
## [1] "The standard deviation is: 33"
```

```r
print("The quantiles of the array are:")
```

```
## [1] "The quantiles of the array are:"
```

```r
quantile(numeric_vector)
```

```
##     0%    25%    50%    75%   100%
##   2.00  16.25  33.00  48.25 100.00
```

You can find many more in your cheatsheets!

Conditionals

Many times it is handy to ask R the question directly. So, if we want to know how many human samples there are, instead of making a table and extracting the answer from it, we can just ask directly:

```
coldata$organism == "human"
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE
```

This returned a vector of the same length as our input (in this case 75, as we have 75 samples) evaluating if each item in the vector (`coldata$organism`) is human (`TRUE`) or not (`FALSE`).

There are several conditional operations:

`a == b` means "is `a` equal to `b`?"

`a != b` means "is `a` not equal to `b`?"

`a > b` means "is `a` greater than `b`?" (numerics only)

`a < b` means "is `a` lower than `b`?" (numerics only)

`a => b` means "is `a` greater or equal to `b`?" (numerics only)

`a =< b` means "is `a` lower or equal to `b`?" (numerics only)

You can also summarize the output of these conditionals with `table()`:

```
table(coldata$organism == "human")
```

```
##
## FALSE  TRUE
##    45    30
```

You can also use conditionals to perform operations over those entries that satisfy the condition with `ifelse()`. Let's say we want to change the column "condition" to contain the same value for guide 1 and guide 2 (two crispr guides Steve is trying to knock down a particular gene), and a different one for the lacz controls.

`ifelse()` will take a condition (in this case `coldata$condition != "lacz"`) and two other values: the first one will be returned if the condition is satisfied (`TRUE`), otherwise the second one will be returned (`FALSE`).

Here are the results of the conditional:

```
coldata$condition != "lacz"
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [13] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [61]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [73] FALSE FALSE FALSE
```

Now we pass our values to `ifelse()` and assign it to a new column

```
print("Here is what ifelse returns")
```

```
## [1] "Here is what ifelse returns"
```

```
ifelse(coldata$condition != "lacz", "KD", "control")
```

```
##  [1] "KD"      "KD"      "KD"      "KD"      "KD"      "KD"      "KD"
##  [8] "KD"      "KD"      "KD"      "control" "control" "control" "control"
## [15] "control" "KD"      "KD"      "KD"      "KD"      "KD"      "KD"
## [22] "KD"      "KD"      "KD"      "KD"      "control" "control" "control"
## [29] "control" "control" "KD"      "KD"      "KD"      "KD"      "KD"
## [36] "KD"      "KD"      "KD"      "KD"      "KD"      "control" "control"
## [43] "control" "control" "control" "KD"      "KD"      "KD"      "KD"
## [50] "KD"      "KD"      "KD"      "KD"      "KD"      "KD"      "control"
## [57] "control" "control" "control" "control" "KD"      "KD"      "KD"
## [64] "KD"      "KD"      "KD"      "KD"      "KD"      "KD"      "KD"
## [71] "control" "control" "control" "control" "control"
```

```
# Now we assign it to the new column kd_ctrl
coldata$kd_ctrl <- ifelse(coldata$condition != "lacz", "kd", "ctrl")
head(coldata)
```

```
##       sample cell_line   organism type condition sequencing_batch kd_ctrl
## AA112  AA112       pt6 chimpanzee ipsc        g1                1      kd
## AA113  AA113       pt6 chimpanzee ipsc        g1                1      kd
## AA114  AA114       pt6 chimpanzee ipsc        g1                1      kd
## AA115  AA115       pt6 chimpanzee ipsc        g1                2      kd
## AA116  AA116       pt6 chimpanzee ipsc        g1                2      kd
## AA117  AA117       pt6 chimpanzee ipsc        g2                1      kd
```

Let's say you want to see only the knock-down samples. To do this, you would need to keep track of *which* rows satisfy the condition `coldata$kd_ctrl == "kd"`.

The function `which()` will take a conditional and return the index of those that satisfy it:

```
which(coldata$kd_ctrl == "kd")
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 16 17 18 19 20 21 22 23 24 25 31 32 33 34 35
## [26] 36 37 38 39 40 46 47 48 49 50 51 52 53 54 55 61 62 63 64 65 66 67 68 69 70
```

We can then use these values to subset `coldata`. We will need to pass these in the brackets (remember: [row number(s), column number(s)]). Here we just print the first few lines, but you can see the entire output for yourself! :-)

```
head(coldata[which(coldata$kd_ctrl == "kd"),])
```

```
##       sample cell_line   organism type condition sequencing_batch kd_ctrl
## AA112  AA112       pt6 chimpanzee ipsc        g1                1      kd
## AA113  AA113       pt6 chimpanzee ipsc        g1                1      kd
## AA114  AA114       pt6 chimpanzee ipsc        g1                 1      kd
## AA115  AA115       pt6 chimpanzee ipsc        g1                 2      kd
## AA116  AA116       pt6 chimpanzee ipsc        g1                 2      kd
## AA117  AA117       pt6 chimpanzee ipsc        g2                 1      kd
```

Multiple conditionals

You can combine different logic operations using the operands "&" (and), and "|" (or). You can also negate the results of a conditional (opposite) using "!" (negate).

For example, the human translation of the following example would be "which samples are of cell lines pt6 *or* sandra":

```
coldata$cell_line == "pt6" | coldata$cell_line == "sandra"
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE
```

This returned a vector of booleans answering the question for each entry we have. We can extract the indexes of the entries satisfying the condition using `which()`:

```
which(coldata$cell_line == "pt6" | coldata$cell_line == "sandra")
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

And finally get all the columns of coldata of those rows:

```
head(coldata[which(coldata$cell_line == "pt6" | coldata$cell_line == "sandra"),])
```

```
##       sample cell_line   organism type condition sequencing_batch kd_ctrl
## AA112  AA112       pt6 chimpanzee ipsc        g1                 1      kd
## AA113  AA113       pt6 chimpanzee ipsc        g1                 1      kd
## AA114  AA114       pt6 chimpanzee ipsc        g1                 1      kd
## AA115  AA115       pt6 chimpanzee ipsc        g1                 2      kd
## AA116  AA116       pt6 chimpanzee ipsc        g1                 2      kd
## AA117  AA117       pt6 chimpanzee ipsc        g2                 1      kd
```

But what if we want to get everything *except* samples from cell lines pt6 *or* sandra? We can simply use the negate operand right before the conditional (be sure to group the two conditionals first with parentheses, otherwise it will just negate the first conditional `coldata$cell_line == "pt6"`).

```r
head(coldata[which(!(coldata$cell_line == "pt6" | coldata$cell_line == "sandra")),])
```

```
##       sample cell_line organism type condition sequencing_batch kd_ctrl
## AA142  AA142        h6    human ipsc        g1                1      kd
## AA143  AA143        h6    human ipsc        g1                1      kd
## AA144  AA144        h6    human ipsc        g1                1      kd
## AA145  AA145        h6    human ipsc        g1                2      kd
## AA146  AA146        h6    human ipsc        g1                2      kd
## AA147  AA147        h6    human ipsc        g2                1      kd
```

The operand *and* works similarly. You can combine two operations and only allow it to be satisfied if *both* of them are satisfied.

For example, let's say I want the sample names of the g2 joc samples (that is, condition must be g2 *and* cell line must be joc):

```r
head(coldata[which(coldata$cell_line == "joc" & coldata$condition == "g2"), "sample"])
```

```
## [1] "AA177" "AA178" "AA179" "AA180" "AA181"
```

What about sample names of the joc samples which are *not* g2?

```r
# Either this
coldata[which(coldata$cell_line == "joc" & coldata$condition != "g2"), "sample"]
```

```
##  [1] "AA172" "AA173" "AA174" "AA175" "AA176" "AA182" "AA183" "AA184" "AA185"
## [10] "AA186"
```

```r
# Or this works
coldata[which(coldata$cell_line == "joc" & !coldata$condition == "g2"), "sample"]
```

```
##  [1] "AA172" "AA173" "AA174" "AA175" "AA176" "AA182" "AA183" "AA184" "AA185"
## [10] "AA186"
```

Other handy conditional functions / operands

- Find a pattern on a character object (`grepl()`)
- Check if a character object starts or ends with a motif (`startsWith()` or `endsWith()`)
- The %in% operand

```r
print("Which cell lines start with h? (case-sensitive)")
```

```
## [1] "Which cell lines start with h? (case-sensitive)"
```

```r
startsWith(coldata$cell_line, "h")
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE
```

```r
print("Which end with 6?")
```

```
## [1] "Which end with 6?"
```

```r
endsWith(coldata$cell_line, "6")
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE
```

```r
print("Which contain the letter a?")
```

```
## [1] "Which contain the letter a?"
```

```r
grepl("a", coldata$cell_line)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE
```

```r
most_interesting_cell_lines <- c("joc", "h6")
print("Which are samples from the most interesting cell lines?")
```

```
## [1] "Which are samples from the most interesting cell lines?"
```

```r
coldata$cell_line %in% most_interesting_cell_lines
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [73]  TRUE  TRUE  TRUE
```

Exercise

Please help Steve find out:

- What type of cell lines are these? How many are iPSC? How many are ESC?
- How many samples were sequenced in each sequencing batch?
- List the samples that were sequenced in sequencing batch #1
- List the joc and sandra chimpanzee lacz samples sequenced in batch #1

Three (or more) dimensional data structures

Ok, we are getting greedy. Are you wondering if can we make three-dimensional data structures (or with even more dimensions)? YES! Same logic, more headaches.

Let's say we want to split this data frame into two - to keep things organised. We will make a list of two data frames, one with the chimps, and the other one with the humans.

```
coldatas_organisms <- split(coldata, coldata$organism)
names(coldatas_organisms)
```

```
## [1] "chimpanzee" "human"
```

Now we can access them individually as:

```
head(coldatas_organisms$human)
```

```
##       sample cell_line organism type condition sequencing_batch kd_ctrl
## AA142  AA142        h6    human ipsc        g1                1      kd
## AA143  AA143        h6    human ipsc        g1                1      kd
## AA144  AA144        h6    human ipsc        g1                1      kd
## AA145  AA145        h6    human ipsc        g1                2      kd
## AA146  AA146        h6    human ipsc        g1                2      kd
## AA147  AA147        h6    human ipsc        g2                1      kd
```

And access their information as we normally would with a data frame. Here I am taking all h6 samples:

```
head(coldatas_organisms$human[which(coldatas_organisms$human$cell_line == "h6"),])
```

```
##       sample cell_line organism type condition sequencing_batch kd_ctrl
## AA142  AA142        h6    human ipsc        g1                1      kd
## AA143  AA143        h6    human ipsc        g1                1      kd
## AA144  AA144        h6    human ipsc        g1                1      kd
## AA145  AA145        h6    human ipsc        g1                2      kd
## AA146  AA146        h6    human ipsc        g1                2      kd
## AA147  AA147        h6    human ipsc        g2                1      kd
```

Debugging

So far we have seen how to define different variable types, work with them, we have read files and made summaries of their contents.

Super common mistakes on these sort of operations are:

- You used the wrong logic operand
- You are missing a parenthesis (either one or the two)
- You are trying to access a row or a column that is not named as you think (remember these things are case sensitive)
- You are trying to access a row or a column that does not exist (for example, if you say you want to access column 5 of the data frame - are you sure your data frame has 5 columns?)

Some strategies that you should remember when debugging are:

- Check what you are working with: Is the input to your command really what you think it is?
- Divide and conquer: If the command you are trying to run is long or has, for example, several conditions, try to test things in parts. Or even better, break it down into two or more smaller commands (it will probably be more readable anyway).
- Think like a robot: Think twice if what you are writing is **literally** what you want to execute. What other interpretations could the code you wrote have?
- When working with a script, narrow down where the problem could be: Can you pinpoint what line of code could be causing the problem?
- Check the output. ALWAYS. Problems usually happen when running several commands in chain without checking the intermediate outputs. Running things line by line and printing everything is sometimes (many times) needed.

If you tried all these and you still have a weird error, rest assured that chances are thousands of people had the exact same one. Googling skills are needed as a programmer.

You will find Stack Overflow to be your best friend from now on. It is a Q&A website for programmers. Most of the times, you will find the same or a similar question to the one you have, and most likely an answer to it. They can give you hints on to why a particular error might occur, but can also give you advice or ideas on to how to implement something in particular. Don't be afraid to try running copy-pasted code, that is how one learns :-).

Exercise

You have been performing single cell RNAseq of samples from Alzheimer's patients for about a year now. To keep track of them, you made `metadata_alz.tab`.

- How many samples have you sequenced so far?
- How many samples of each brain region (column `region`) you have per diagnosis (Alzheimer's or control) (column `dx`)?

One day, you wake up to a 10x Genomics email - some beads have been flawed during shipping. Your lab got some of these beads about a year ago. They don't know how many they have sent you, so they ask you to send all the quality control to them for a check up.

Today you finally receive their answer. Your hands start sweating - 40 of your samples have been compromised.

Your PI asks you for a summary of the samples that have been compromised:

- Which sequencing runs have been affected?
- Which sequencing run has the most samples affected?
- How many of these affected samples have been sent to our collaborators? (column `sent_to_collab`)
- Is this more frequent on a specific chemistry? (3' vs 5')
- Where does these samples come from? (samples starting with "RFT" belong to a collaboration with Fergusson Thomasson, "REW" belong to Elisa Watson)

You also need to provide an excel file with:

- Sample name
- Chemistry
- Sequencing run