Data Structures

- To be able to identify the 5 main data types.
- To begin exploring data frames, and understand how they are related to vectors and lists.
- To be able to ask questions from R about the type, class, and structure of an object.
- To understand the information of the attributes “names”, “class”, and “dim”.

- How can I read data in R?
- What are the basic data types in R?
- How do I represent categorical information in R?

One of R’s most powerful features is its ability to deal with tabular data - such as you may already have in a spreadsheet or a CSV file. Let’s start by making a toy dataset in your `data/` directory, called `feline-data.csv`:

```
cats <- data.frame(coat = c("calico", "black", "tabby"),
                   weight = c(2.1, 5.0, 3.2),
                   likes_string = c(1, 0, 1))
```

We can now save `cats` as a CSV file. It is good practice to call the argument names explicitly so the function knows what default values you are changing. Here we are setting `row.names = FALSE`. Recall you can use `?write.csv` to pull up the help file to check out the argument names and their default values.

```
write.csv(x = cats, file = "data/feline-data.csv", row.names = FALSE)
```

The contents of the new file, `feline-data.csv`:

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

Tip: Editing Text files in R

Alternatively, you can create `data/feline-data.csv` using a text editor (Nano), or within RStudio with the **File -> New File -> Text File** menu item.

We can load this into R via the following:

```
cats <- read.csv(file = "data/feline-data.csv")
cats
```

```
##     coat weight likes_string
## 1 calico    2.1            1
## 2  black    5.0            0
## 3  tabby    3.2            1
```

The `read.table` function is used for reading in tabular data stored in a text file where the columns of data are separated by punctuation characters such as CSV files (csv = comma-separated values). Tabs and commas are the most common punctuation characters used to separate or delimit data points in csv files. For convenience R provides 2 other versions of `read.table`. These are: `read.csv` for files where the data are separated with commas and `read.delim` for files where the data are separated with tabs. Of these three functions `read.csv` is the most commonly used. If needed it is possible to override the default delimiting punctuation marks for both `read.csv` and `read.delim`.

Check your data for factors

In recent times, the default way how R handles textual data has changed. Text data was interpreted by R automatically into a format called “factors”. But there is an easier format that is called “character”. We will hear about factors later, and what to use them for. For now, remember that in most cases, they are not needed and only complicate your life, which is why newer R versions read in text as “character”. Check now if your version of R has automatically created factors and convert them to “character” format:

1. Check the data types of your input by typing `str(cats)`
2. In the output, look at the three-letter codes after the colons: If you see only “num” and “chr”, you can continue with the lesson and skip this box. If you find “fct”, continue to step 3.
3. Prevent R from automatically creating “factor” data. That can be done by the following code: `options(stringsAsFactors = FALSE)`. Then, re-read the cats table for the change to take effect.
4. You must set this option every time you restart R. To not forget this, include it in your analysis script before you read in any data, for example in one of the first lines.

5. For R versions greater than 4.0.0, text data is no longer converted to factors anymore. So you can install this or a newer version to avoid this problem. If you are working on an institute or company computer, ask your administrator to do it.

We can begin exploring our dataset right away, pulling out columns by specifying them using the `$` operator:

```
cats$weight
```

```
## [1] 2.1 5.0 3.2
```

```
cats$coat
```

```
## [1] "calico" "black"  "tabby"
```

We can do other operations on the columns:

```
## Say we discovered that the scale weighs two Kg light:
cats$weight + 2
```

```
## [1] 4.1 7.0 5.2
```

```
paste("My cat is", cats$coat)
```

```
## [1] "My cat is calico" "My cat is black"  "My cat is tabby"
```

But what about

```
#cats$weight + cats$coat
```

Understanding what happened here is key to successfully analyzing data in R.

Data Types

If you guessed that the last command will return an error because `2.1` plus `"black"` is nonsense, you're right - and you already have some intuition for an important concept in programming called *data types*. We can ask what type of data something is:

```
typeof(cats$weight)
```

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

There are 5 main types: `double`, `integer`, `complex`, `logical` and `character`. For historic reasons, `double` is also called `numeric`.

```
typeof(3.14)
```

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

```
typeof(1L) # The L suffix forces the number to be an integer, since by default R uses float numbers
```

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

```
typeof(1+1i)
```

```
## [1] "complex"
```

```
typeof(TRUE)
```

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

```
typeof('banana')
```

```
## [1] "character"
```

No matter how complicated our analyses become, all data in R is interpreted as one of these basic data types. This strictness has some really important consequences.

A user has added details of another cat. This information is in the file `data/feline-data_v2.csv`.

```
file.show("data/feline-data_v2.csv")
```

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
tabby,2.3 or 2.4,1
```

Load the new cats data like before, and check what type of data we find in the `weight` column:

```
cats <- read.csv(file="data/feline-data_v2.csv")
typeof(cats$weight)
```

```
## [1] "character"
```

Oh no, our weights aren't the double type anymore! If we try to do the same math we did on them before, we run into trouble:

```
#cats$weight + 2
```

What happened? The `cats` data we are working with is something called a *data frame*. Data frames are one of the most common and versatile types of *data structures* we will work with in R. A given column in a data frame cannot be composed of different data types. In this case, R does not read everything in the data frame column `weight` as a *double*, therefore the entire column data type changes to something that is suitable for everything in the column.

When R reads a csv file, it reads it in as a *data frame*. Thus, when we loaded the `cats` csv file, it is stored as a data frame. We can recognize data frames by the first row that is written by the `str()` function:

```
str(cats)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ coat        : chr  "calico" "black" "tabby" "tabby"
##  $ weight      : chr  "2.1" "5" "3.2" "2.3 or 2.4"
##  $ likes_string: int  1 0 1 1
```

Data frames are composed of rows and columns, where each column has the same number of rows. Different columns in a data frame can be made up of different data types (this is what makes them so versatile), but everything in a given column needs to be the same type (e.g., vector, factor, or list).

Let's explore more about different data structures and how they behave. For now, let's remove that extra line from our cats data and reload it, while we investigate this behavior further:

feline-data.csv:

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

And back in RStudio:

```
cats <- read.csv(file="data/feline-data.csv")
```

Vectors and Type Coercion

To better understand this behavior, let's meet another of the data structures: the *vector*.

```
my_vector <- vector(length = 3)
my_vector
```

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

A vector in R is essentially an ordered list of things, with the special condition that *everything in the vector must be the same basic data type*. If you don't choose the datatype, it'll default to `logical`; or, you can declare an empty vector of whatever type you like.

```
another_vector <- vector(mode='character', length=3)
another_vector
```

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

You can check if something is a vector:

```
str(another_vector)
```

```
##  chr [1:3] "" "" ""
```

The somewhat cryptic output from this command indicates the basic data type found in this vector - in this case `chr`, character; an indication of the number of things in the vector - actually, the indexes of the vector, in this case `[1:3]`; and a few examples of what's actually in the vector - in this case empty character strings. If we similarly do

```
str(cats$weight)
```

```
##  num [1:3] 2.1 5 3.2
```

we see that `cats$weight` is a vector, too - *the columns of data we load into R data.frames are all vectors*, and that's the root of why R forces everything in a column to be the same basic data type.

Discussion 1

Why is R so opinionated about what we put in our columns of data? How does this help us?

Discussion 1

By keeping everything in a column the same, we allow ourselves to make simple assumptions about our data; if you can interpret one entry in the column as a number, then you can interpret *all* of them as numbers, so we don't have to check every time. This consistency is what people mean when they talk about *clean data*; in the long run, strict consistency goes a long way to making our lives easier in R.

Coercion by combining vectors

You can also make vectors with explicit contents with the combine function:

```
combine_vector <- c(2,6,3)
combine_vector
```

```
## [1] 2 6 3
```

Given what we've learned so far, what do you think the following will produce?

```
quiz_vector <- c(2,6,'3')
```

This is something called *type coercion*, and it is the source of many surprises and the reason why we need to be aware of the basic data types and how R will interpret them. When R encounters a mix of types (here double and character) to be combined into a single vector, it will force them all to be the same type. Consider:

```
coercion_vector <- c('a', TRUE)
coercion_vector
```

```
## [1] "a"    "TRUE"
```

```
another_coercion_vector <- c(0, TRUE)
another_coercion_vector
```

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

The type hierarchy

The coercion rules go: `logical` -> `integer` -> `double` (“`numeric`”) -> `complex` -> `character`, where -> can be read as *are transformed into*. For example, combining `logical` and `character` transforms the result to `character`:

```
c('a', TRUE)
```

```
## [1] "a"    "TRUE"
```

A quick way to recognize `character` vectors is by the quotes that enclose them when they are printed.

You can try to force coercion against this flow using the `as.` functions:

```
character_vector_example <- c('0','2','4')
character_vector_example
```

```
## [1] "0" "2" "4"
```

```
character_coerced_to_double <- as.double(character_vector_example)
character_coerced_to_double
```

```
## [1] 0 2 4
```

```
double_coerced_to_logical <- as.logical(character_coerced_to_double)
double_coerced_to_logical
```

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

As you can see, some surprising things can happen when R forces one basic data type into another! Nitty-gritty of type coercion aside, the point is: if your data doesn’t look like what you thought it was going to look like, type coercion may well be to blame; make sure everything is the same type in your vectors and your columns of data.frames, or you will get nasty surprises!

But coercion can also be very useful! For example, in our `cats` data `likes_string` is numeric, but we know that the 1s and 0s actually represent `TRUE` and `FALSE` (a common way of representing them). We should use the `logical` datatype here, which has two states: `TRUE` or `FALSE`, which is exactly what our data represents. We can ‘coerce’ this column to be `logical` by using the `as.logical` function:

```
cats$likes_string
```

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

```
cats$likes_string <- as.logical(cats$likes_string)
cats$likes_string
```

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

Challenge 1

An important part of every data analysis is cleaning the input data. If you know that the input data is all of the same format, (e.g. numbers), your analysis is much easier! Clean the cat data set from the chapter about type coercion.

Copy the code template

Create a new script in RStudio and copy and paste the following code. Then move on to the tasks below, which help you to fill in the gaps (_____).

```
# Read data
cats <- read.csv("data/feline-data_v2.csv")

# 1. Print the data
_____

# 2. Show an overview of the table with all data types
_____(cats)

# 3. The "weight" column has the incorrect data type __________.
#    The correct data type is: ____________.

# 4. Correct the 4th weight data point with the mean of the two given values
cats$weight[4] <- 2.35
#    print the data again to see the effect
cats

# 5. Convert the weight to the right data type
cats$weight <- _______________(cats$weight)

#    Calculate the mean to test yourself
mean(cats$weight)

# If you see the correct mean value (and not NA), you did the exercise
# correctly!
```

Instructions for the tasks

1. Print the data

Execute the first statement (`read.csv(...)`). Then print the data to the console

Tip 1.1

Show the content of any variable by typing its name.

Solution to Challenge 1.1

Two correct solutions:

```
cats
print(cats)
```

2. Overview of the data types

The data type of your data is as important as the data itself. Use a function we saw earlier to print out the data types of all columns of the `cats` table.

Tip 1.2

In the chapter “Data types” we saw two functions that can show data types. One printed just a single word, the data type name. The other printed a short form of the data type, and the first few values. We need the second here.

> # Solution to Challenge 1.2
>
> ```
> str(cats)
> ```

3. Which data type do we need?

The shown data type is not the right one for this data (weight of a cat). Which data type do we need?

- Why did the `read.csv()` function not choose the correct data type?
- Fill in the gap in the comment with the correct data type for cat weight!

Tip 1.3

Scroll up to the section about the type hierarchy to review the available data types

Solution to Challenge 1.3

- Weight is expressed on a continuous scale (real numbers). The R data type for this is “double” (also known as “numeric”).
- The fourth row has the value “2.3 or 2.4”. That is not a number but two, and an english word. Therefore, the “character” data type is chosen. The whole column is now text, because all values in the same columns have to be the same data type.

4. Correct the problematic value

The code to assign a new weight value to the problematic fourth row is given. Think first and then execute it: What will be the data type after assigning a number like in this example? You can check the data type after executing to see if you were right.

Tip 1.4

Revisit the hierarchy of data types when two different data types are combined.

> ## Solution to challenge 1.4
>
> The data type of the column “weight” is “character”. The assigned data type is “double”. Combining two data types yields the data type that is higher in the following hierarchy:
>
> ```
> logical < integer < double < complex < character
> ```
>
> Therefore, the column is still of type character! We need to manually convert it to “double”. {: .solution}

5. Convert the column “weight” to the correct data type

Cat weight are numbers. But the column does not have this data type yet. Coerce the column to floating point numbers.

Tip 1.5

The functions to convert data types start with `as.`. You can look for the function further up in the manuscript or use the RStudio auto-complete function: Type “`as.`” and then press the TAB key.

> ## Solution to Challenge 1.5
>
> There are two functions that are synonymous for historic reasons:
>
> ```
> cats$weight <- as.double(cats$weight)
> cats$weight <- as.numeric(cats$weight)
> ```

Some basic vector functions

The combine function, `c()`, will also append things to an existing vector:

```
ab_vector <- c('a', 'b')
ab_vector
```

```
## [1] "a" "b"
```

```
combine_example <- c(ab_vector, 'SWC')
combine_example
```

```
## [1] "a"   "b"   "SWC"
```

You can also make series of numbers:

```
mySeries <- 1:10
mySeries
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1,10, by=0.1)
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4
## [16]  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9
## [31]  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3  5.4
## [46]  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
## [61]  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4
## [76]  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7  9.8  9.9
## [91] 10.0
```

We can ask a few questions about vectors:

```
sequence_example <- 20:25
head(sequence_example, n=2)
```

```
## [1] 20 21
```

```
tail(sequence_example, n=4)
```

```
## [1] 22 23 24 25
```

```
length(sequence_example)
```

```
## [1] 6
```

```
typeof(sequence_example)
```

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

We can get individual elements of a vector by using the bracket notation:

```
first_element <- sequence_example[1]
first_element
```

```
## [1] 20
```

To change a single element, use the bracket on the other side of the arrow:

```
sequence_example[1] <- 30
sequence_example
```

```
## [1] 30 21 22 23 24 25
```

Challenge 2

Start by making a vector with the numbers 1 through 26. Then, multiply the vector by 2.

Solution to Challenge 2

```
x <- 1:26
x <- x * 2
```

Lists

Another data structure you'll want in your bag of tricks is the `list`. A list is simpler in some ways than the other types, because you can put anything you want in it. Remember *everything in the vector must be of the same basic data type*, but a list can have different data

types:

```
list_example <- list(1, "a", TRUE, 1+4i)
list_example
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

When printing the object structure with `str()`, we see the data types of all elements:

```
str(list_example)
```

```
## List of 4
##  $ : num 1
##  $ : chr "a"
##  $ : logi TRUE
##  $ : cplx 1+4i
```

What is the use of lists? They can **organize data of different types**. For example, you can organize different tables that belong together, similar to spreadsheets in Excel. But there are many other uses, too.

We will see another example that will maybe surprise you in the next chapter.

To retrieve one of the elements of a list, use the **double bracket**:

```
list_example[[2]]
```

```
## [1] "a"
```

The elements of lists also can have **names**, they can be given by prepending them to the values, separated by an equals sign:

```
another_list <- list(title = "Numbers", numbers = 1:10, data = TRUE )
another_list
```

```
## $title
## [1] "Numbers"
##
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $data
## [1] TRUE
```

This results in a **named list**. Now we have a new function of our object! We can access single elements by an additional way!

```
another_list$title
```

```
## [1] "Numbers"
```

Names

With names, we can give meaning to elements. It is the first time that we do not only have the **data**, but also explaining information. It is *metadata* that can be stuck to the object like a label. In R, this is called an **attribute**. Some attributes enable us to do more with our object, for example, like here, accessing an element by a self-defined name.

Accessing vectors and lists by name

We have already seen how to generate a named list. The way to generate a named vector is very similar. You have seen this function before:

```
pizza_price <- c( pizzasubito = 5.64, pizzafresh = 6.60, callapizza = 4.50 )
```

The way to retrieve elements is different, though:

```
pizza_price["pizzasubito"]
```

```
## pizzasubito
##        5.64
```

The approach used for the list does not work:

```
#pizza_price$pizzafresh
```

It will pay off if you remember this error message, you will meet it in your own analyses. It means that you have just tried accessing an element like it was in a list, but it is actually in a vector.

Accessing and changing names

If you are only interested in the names, use the `names()` function:

```
names(pizza_price)
```

```
## [1] "pizzasubito" "pizzafresh"  "callapizza"
```

We have seen how to access and change single elements of a vector. The same is possible for names:

```
names(pizza_price)[3]
```

```
## [1] "callapizza"
```

```
names(pizza_price)[3] <- "call-a-pizza"
pizza_price
```

```
##  pizzasubito   pizzafresh call-a-pizza
##         5.64         6.60         4.50
```

Challenge 3

- What is the data type of the names of `pizza_price`? You can find out using the `str()` or `typeof()` functions.

Solution to Challenge 3

You get the names of an object by wrapping the object name inside `names(...)`. Similarly, you get the data type of the names by again wrapping the whole code in `typeof(...)`:

```
typeof(names(pizza))
```

alternatively, use a new variable if this is easier for you to read:

```
n <- names(pizza)
typeof(n)
```

Challenge 4

Instead of just changing some of the names a vector/list already has, you can also set all names of an object by writing code like (replace ALL CAPS text):

```
names( OBJECT ) <-  CHARACTER_VECTOR
```

Create a vector that gives the number for each letter in the alphabet!

1. Generate a vector called `letter_no` with the sequence of numbers from 1 to 26!
2. R has a built-in object called `LETTERS`. It is a 26-character vector, from A to Z. Set the names of the number sequence to this 26 letters
3. Test yourself by calling `letter_no["B"]`, which should give you the number 2!

Solution to Challenge 4

```
letter_no <- 1:26   # or seq(1,26)
names(letter_no) <- LETTERS
letter_no["B"]
```

Data frames

We have data frames at the very beginning of this lesson, they represent a table of data. We didn't go much further into detail with our example cat data frame:

```
cats
```

```
##     coat weight likes_string
## 1 calico    2.1         TRUE
## 2  black    5.0        FALSE
## 3  tabby    3.2         TRUE
```

We can now understand something a bit surprising in our data.frame; what happens if we run:

```
typeof(cats)
```

```
## [1] "list"
```

We see that data.frames look like lists 'under the hood'. Think again what we heard about what lists can be used for:

> Lists organize data of different types

Columns of a data frame are vectors of different types, that are organized by belonging to the same table.

A data.frame is really a list of vectors. It is a special list in which all the vectors must have the same length.

How is this "special"-ness written into the object, so that R does not treat it like any other list, but as a table?

```
class(cats)
```

```
## [1] "data.frame"
```

A **class**, just like names, is an attribute attached to the object. It tells us what this object means for humans.

You might wonder: Why do we need another what-type-of-object-is-this-function? We already have `typeof()`? That function tells us how the object is **constructed in the computer**. The `class` is the **meaning of the object for humans**. Consequently, what `typeof()` returns is *fixed* in R (mainly the five data types), whereas the output of `class()` is *diverse* and *extendable* by R packages.

In our `cats` example, we have an integer, a double and a logical variable. As we have seen already, each column of data.frame is a vector.

```
cats$coat
```

```
## [1] "calico" "black"  "tabby"
```

```
cats[,1]
```

```
## [1] "calico" "black"  "tabby"
```

```
typeof(cats[,1])
```

```
## [1] "character"
```

```
str(cats[,1])
```

```
##  chr [1:3] "calico" "black" "tabby"
```

Each row is an *observation* of different variables, itself a data.frame, and thus can be composed of elements of different types.

```
cats[1,]
```

```
##     coat weight likes_string
## 1 calico    2.1         TRUE
```

```
typeof(cats[1,])
```

```
## [1] "list"
```

```
str(cats[1,])
```

```
## 'data.frame':    1 obs. of  3 variables:
##  $ coat        : chr "calico"
##  $ weight      : num 2.1
##  $ likes_string: logi TRUE
```

Challenge 5

There are several subtly different ways to call variables, observations and elements from data.frames:

- `cats[1]`
- `cats[[1]]`
- `cats$coat`
- `cats["coat"]`
- `cats[1, 1]`
- `cats[, 1]`
- `cats[1, ]`

Try out these examples and explain what is returned by each one.

Hint: Use the function `typeof()` to examine what is returned in each case.

Solution to Challenge 5

```
cats[1]
```

```
##     coat
## 1 calico
## 2  black
## 3  tabby
```

We can think of a data frame as a list of vectors. The single brace `[1]` returns the first slice of the list, as another list. In this case it is the first column of the data frame.

```
cats[[1]]
```

```
## [1] "calico" "black"  "tabby"
```

The double brace `[[1]]` returns the contents of the list item. In this case it is the contents of the first column, a *vector* of type *character*.

```
cats$coat
```

```
## [1] "calico" "black"  "tabby"
```

This example uses the `$` character to address items by name. *coat* is the first column of the data frame, again a *vector* of type *character*.

```
cats["coat"]
```

```
##     coat
## 1 calico
## 2  black
## 3  tabby
```

Here we are using a single brace `["coat"]` replacing the index number with the column name. Like example 1, the returned object is a *list*.

```
cats[1, 1]
```

```
## [1] "calico"
```

This example uses a single brace, but this time we provide row and column coordinates. The returned object is the value in row 1, column 1. The object is a *vector* of type *character*.

```
cats[, 1]
```

```
## [1] "calico" "black"  "tabby"
```

Like the previous example we use single braces and provide row and column coordinates. The row coordinate is not specified, R interprets this missing value as all the elements in this *column* and returns them as a *vector*.

```
cats[1, ]
```

```
##     coat weight likes_string
## 1 calico    2.1         TRUE
```

Again we use the single brace with row and column coordinates. The column coordinate is not specified. The return value is a *list* containing all the values in the first row.

Tip: Renaming data frame columns

Data frames have column names, which can be accessed with the `names()` function.

```
names(cats)
```

```
## [1] "coat"         "weight"       "likes_string"
```

If you want to rename the second column of `cats`, you can assign a new name to the second element of `names(cats)`.

```
names(cats)[2] <- "weight_kg"
cats
```

```
##     coat weight_kg likes_string
## 1 calico       2.1         TRUE
## 2  black       5.0        FALSE
## 3  tabby       3.2         TRUE
```

Matrices

Last but not least is the matrix. We can declare a matrix full of zeros:

```
matrix_example <- matrix(0, ncol=6, nrow=3)
matrix_example
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    0    0    0    0
## [2,]    0    0    0    0    0    0
## [3,]    0    0    0    0    0    0
```

What makes it special is the `dim()` attribute:

```
dim(matrix_example)
```

```
## [1] 3 6
```

And similar to other data structures, we can ask things about our matrix:

```
typeof(matrix_example)
```

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

```
class(matrix_example)
```

```
## [1] "matrix" "array"
```

```
str(matrix_example)
```

```
##  num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
```

```
nrow(matrix_example)
```

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

```
ncol(matrix_example)
```

```
## [1] 6
```

Challenge 6

What do you think will be the result of `length(matrix_example)`? Try it. Were you right? Why / why not?

Solution to Challenge 6

What do you think will be the result of `length(matrix_example)`?

```
matrix_example <- matrix(0, ncol=6, nrow=3)
length(matrix_example)
```

```
## [1] 18
```

Because a matrix is a vector with added dimension attributes, `length` gives you the total number of elements in the matrix.

Challenge 7

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix`!)

Solution to Challenge 7

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix`!)

```
x <- matrix(1:50, ncol=5, nrow=10)
x <- matrix(1:50, ncol=5, nrow=10, byrow = TRUE) # to fill by row
```

Challenge 8

Create a list of length two containing a character vector for each of the sections in this part of the workshop:

- Data types

- Data structures

Populate each character vector with the names of the data types and data structures we've seen so far.

Solution to Challenge 8

```
dataTypes <- c('double', 'complex', 'integer', 'character', 'logical')
dataStructures <- c('data.frame', 'vector', 'list', 'matrix')
answer <- list(dataTypes, dataStructures)
```

Note: it's nice to make a list in big writing on the board or taped to the wall listing all of these types and structures - leave it up for the rest of the workshop to remind people of the importance of these basics.

Challenge 9

Consider the R output of the matrix below:

```
##      [,1] [,2]
## [1,]    4    1
## [2,]    9    5
## [3,]   10    7
```

What was the correct command used to write this matrix? Examine each command and try to figure out the correct one before typing them. Think about what matrices the other commands will produce.

1. `matrix(c(4, 1, 9, 5, 10, 7), nrow = 3)`
2. `matrix(c(4, 9, 10, 1, 5, 7), ncol = 2, byrow = TRUE)`
3. `matrix(c(4, 9, 10, 1, 5, 7), nrow = 2)`
4. `matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)`

Solution to Challenge 9

Consider the R output of the matrix below:

```
##      [,1] [,2]
## [1,]    4    1
## [2,]    9    5
## [3,]   10    7
```

What was the correct command used to write this matrix? Examine each command and try to figure out the correct one before typing them. Think about what matrices the other commands will produce.

```
matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)
```

- Use `read.csv` to read tabular data in R.
- The basic data types in R are double, integer, complex, logical, and character.
- Data structures such as data frames or matrices are built on top of lists and vectors, with some added attributes.