4 Data frames

Data frames are the primary way in which R stores and manipulates data. Like matrices, they are two-dimensional tables. However, data frames are much more versatile than matrices. Importantly, and in contrast to a matrix, the different columns of a data frame can contain different data types. For example, we can store data tables with one row per datapoint and different types of information in each columns. There, we could have the names of subjects (character) in one column, various grouping variables (so-called 'factors' like gender, colour, genotype etc.) in other columns, and then numerical measurements in yet other columns. Storing data in this form enables the use of powerful methods—and quick short cuts—within R to perform various data analysis tasks.

4.1 Defining data frames

As an example, we shall construct a very simple data frame holding the data below on body mass (kg) and brain mass (g) for four animal species.

Animal	Body mass (kg)	Brain mass (g)
Gorilla	207	406
Human	62	1320
Triceratops	9400	70
Mole	0.122	3

Data taken from Rousseeuw, P.J. & Leroy, A.M. (1987) Robust Regression and Outlier Detection. Wiley, p. 57.

We do this by first creating three vectors, each containing one type of data, species name, mass, and weight. We then combine the three vectors into a data frame while specifying the column names we want to associate with our three types of information.

```
> v1 <- c("Gorilla", "Human", "Triceratops", "Mole")
> v2 <- c(207, 62, 9400, 0.122)
> v3 <- c(406, 1320, 70, 3)
> massFrame <- data.frame(Animal=v1, Body_mass=v2, Brain_mass=v3)
> massFrame
        Animal Body_mass Brain_mass
 1     Gorilla   207.000        406
 2       Human    62.000       1320
 3 Triceratops  9400.000         70
 4        Mole     0.122          3
```

Note that we have used underscores (_) to separate the words of "Body mass". R, and all other programming languages, don't like spaces and will not recognise that the word before and after the space are part of the same name. R also doesn't like variable names that start with numbers, so avoid those.

We can access the contents in the data frame in different ways. One method is to identify columns in the same way we did for matrices before. So we can get the species names with

```
> massFrame[1,]
   Animal Body_mass Brain_mass
 1 Gorilla       207        406
```

Or we can access the row of data for a particular animal, let's say the gorilla:

```
> massFrame[1,]
   Animal Body_mass Brain_mass
 1 Gorilla       207        406
```

Finally, we can pull out individual values by specifying their coordinates. Thus, we get the brain mass of the mole, in the fourth row and third columns with

```
> massFrame[4,3]
[1] 3
```

However, data frames are more sophisticated and allow us to access columns by their names. First, let us ask R for the names of the columns in our table.

```
> names(massFrame)
 [1] "Animal" "Body_mass"   "Brain_mass"
```

To access a column by its name, we append the column name to the name of the data frame, separated by a dollar sign, as in

```
> massFrame$Body_mass
 [1]  207.000   62.000 9400.000    0.122
```

Alternatively, we can specify the column name in the context of the coordinate system used above,

```
> massFrame[1,"Body_mass"]
 [1] 207
```

4.2 Reading data

For larger data sets, it is tedious to enter each value by hand as above. It is then much more efficient to enter the data in a spreadsheet (e.g., in Excel), save the table to a text file and use R functions to directly read in the data from this file. There are several functions to read data from files (have a look with "help(read.table)"). `read.table()` is the most general function that allows you to specify how your data table is organised, how the values in a row are separated (commas, tabs, white space, etc.) and so forth. Other functions are variations of `read.table()` that have preset values for certain arguments. For example, `read.delim()` assumes

that the data values are separated by tabs and that the first row of the file contains the columns names. This would be the case if you saved your Excel spreadsheet as a text file, making `read.delim()` a very handy shortcut. Another convenient function is `read.csv()`, which assumes that the values are separated by commas. CSV files are common in genomics, so this function is handy there.

There are many other ways to import (and export) data into R including Microsoft formats, files from other statistical programs such as SAS, Stata and interfaces to relational databases. These are beyond the scope of this course but more information can be found here R : Data Import/Export.

As an exercise, we will now go through the steps of loading data into R. Here is what you need to do:

- Open up a web browser and navigate to the Moodle page where this course is hosted.
- Download the file called `body_brain_mass.txt` containing the full data set of body and brain mass for 27 animals.
- Make sure the file is on your Desktop or alternatively in another folder on your computer that you know the location of. You may have to navigate to your `Downloads` folder or wherever your web browser saves files.
- Open the file in the R console text editor so that you can view its contents.
- Set R's working directory. This is the folder where R will look for files when you tell it to read in the data, or where it will deposit any files that you make it create (output data, figures etc.). You can do so with the little bit of graphical user interface that the R console offers. If working in Windows, select the menu item `File -> Change dir...` and then navigate to the folder where you put the data. On a Mac, the equivalent menu option is `Misc -> Change Working Directory`. Alternatively, you can use the command-line function `setwd` at the prompt in both systems. This is a bit trickier, as you have to specify the path towards your folder as the argument.

HINT: Use the Tab key for auto-completion of the file names. It will save time and help to avoid typos. You can try this be typing `massFrame <- read.table("body` and then pressing Tab. It can also be used for object or function names. Try this by typing `me` at the prompt and pressing Tab and you should see a list of functions appear staring with "me".

- Once we have set the working directory, we tell R to read in the file and store it in a data frame called `massFrame`,

```
> massFrame <- read.table("body_brain_mass.txt",header=T)
```

After you have read in the data, it is good practice that it has been correctly loaded. Here, the table is relatively small, so you could do it by just calling the name of the table, `massFrame`, for R to show you the entire data frame. But you would not like to do that after loading, for example, the millions of lines of a genomics dataset. There are a few things you can do to get a good idea of whether everything went

fine. First, you can double-check the dimensions of the data frame and see whether it corresponds to the number of rows and columns that you would expect. Use the `dim` function that returns the number of rows and columns as a vector.

```
dim(massFrame)
 [1] 27  3
```

You can also ask R to show you the column names with

```
names(massFrame)
 [1] "Animal"  "Body_mass"  "Brain_mass"
```

Finally, you can get a little preview of the data table and look at the first few rows with the function `head`,

```
> head(massFrame)
           Animal Body_mass Brain_mass
1 Mountain_beaver      1.35      465.0
2             Cow    465.00      423.0
3       Grey_wolf     36.33      119.5
4            Goat     27.66      115.0
5      Guinea_pig      1.04        5.5
6     Dipliodocus  11700.00       50.0
```

or look at the last few rows with `tail`.

4.3 Manipulating data

We shall continue working with our animal data stored in `massFrame`. First, we can examine the data by applying the `summary` function to each of the columns. The function takes the identity of the column (or indeed, many different kinds of variables containing data or other information, see `help("summary")` as an argument. It returns different summary statistics, depending on the type of data of its argument. Here, we have columns of numbers and `summary` produces some statistics to describe their distribution. Let's do this first for body mass, using its name to identify the column.

```
> summary(massFrame$Body_mass)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
    0.02     5.05    55.50  4437.00   493.00 87000.00
```

Now the same for brain mass:

```
> summary(massFrame$Brain_mass)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.4    53.0   157.0   612.7   431.5  5712.0
```

We have already seen that R provides many ways to subset and select data in various formats such as data frames, vector and matrices. We can try some of those with the brain mass data. We first access access

the first ten values of the animal names and body masses (columns 1 and 2) by specifying rows and columns in square brackets. To do so, we use the following syntax,

```
massFrame[1:10,c(1,2)]
            Animal Body_mass
1  Mountain_beaver      1.35
2              Cow    465.00
3        Grey_wolf     36.33
4             Goat     27.66
5       Guinea_pig      1.04
6      Dipliodocus  11700.00
7   Asian_elephant   2547.00
8           Donkey    187.10
9            Horse    521.00
10    Potar_monkey     10.00
```

Note that we use a sequence, `1:10`, to access the rows and use a vector, `c(1,2)`, to access the columns. The two options are equivalent, and `massFrame[1:10,1:2]`, `massFrame[c(1:10),1:2]` and `massFrame[c(1:10),c(1:2)]` would have done the trick, too. Things are different, of course, if we want to select animal names and brain mass, which are columns 1 and 3. There, column numbers are not consecutive and we need to specify a vector `c(1,3)` containing the two.

As seen previously, to access all the columns (or all rows) we simply leave out the respective column IDs. For example to access the first ten rows and all the columns we would use the following code.

```
> massFrame[1:10,]
            Animal Body_mass Brain_mass
1  Mountain_beaver  1.35     465.0
2              Cow  465.00   423.0
3        Grey_wolf  36.33    119.5
4             Goat  27.66    115.0
5       Guinea_pig  1.04     5.5
6      Dipliodocus  11700.00 50.0
7   Asian_elephant  2547.00  4603.0
8           Donkey  187.10   419.0
9            Horse  521.00   655.0
10    Potar_monkey  10.00    115.0
```

Very useful for subsetting a data frame is the `subset` function that we have already seen in the section on logical operators. It allows us to select rows that satisfy some condition, such as a numerical requirement. For example, if we wish to create a smaller data frame containing only those animals that have a body mass greater than 800 kg, we can do the following,

```
> subFrame <- subset(massFrame, Body_mass > 800)
> subFrame
             Animal Body_mass Brain_mass
6       Dipliodocus     11700       50.0
7    Asian_elephant      2547     4603.0
```

```
15 African_elephant      6654     5712.0
16      Triceratops      9400       70.0
25    Brachiosaurus     87000      154.5
```

What the `subset` function does is compare the Body_mass variable of each row to 800 and keep only those rows where the logical condition of value> 800 is satisfied.

Another alternative function for subsetting is `which`. This function outputs the row numbers of data points that satisfy the logical expression. To select all the animals with body mass greater than 800 kg we use the `which` function to give us the places in the vector for which the logical expression is true.

```
> which( massFrame$Body_mass > 800 )
 [1]  6  7 15 16 25
```

This means that the 6th, 7th, 15th, 16th and 25th entries in the body mass vector are all greater than 800 kg. We can use the positions returned to select the rows of the data frame as follows.

```
> rows <- which( massFrame$Body_mass > 800 )
> subFrame <- massFrame[ rows, ]
> subFrame
             Animal Body_mass Brain_mass
6       Dipliodocus     11700       50.0
7    Asian_elephant      2547     4603.0
15 African_elephant      6654     5712.0
16      Triceratops      9400       70.0
25    Brachiosaurus     87000      154.5
```

This gives us exactly the same subset of data as using the `subset` function, as expected. Either approach may be more or less useful depending on the situation.

We can also combine logical expressions in the `subset` (or `which`) function. For example to select the animals with body mass greater than 800kg and brain mass greater than 4000g we use the logical AND operator, `&`. For example

```
> subFrame <- subset(massFrame, Body_mass > 800 & Brain_mass > 4000)
> subFrame
             Animal Body_mass Brain_mass
7    Asian_elephant      2547       4603
15 African_elephant      6654       5712
```

We can immediately see that there are only two animals that satisfy these requirements; the Asian and African elephants.

The logical operator for OR is `|`. To select the animals with body mass greater than 800kg or brain mass greater than 4000g we can do the following,

```
subFrame <- subset(massFrame, Body_mass > 800 | Brain_mass > 4000)
```

```
> subFrame
              Animal Body_mass Brain_mass
6       Dipliodocus     11700       50.0
7    Asian_elephant      2547     4603.0
15 African_elephant      6654     5712.0
16      Triceratops      9400       70.0
25    Brachiosaurus     87000      154.5
```

Now we shall see how we can add columns to the data frame in particular, we shall add some columns that represent a elementary transformation of these data. Recall that above we used the `summary` function to examine the body and brain masses in the data frame. We can see that these data have huge ranges; the body mass ranges from 0.02-87000 kg and the brain mass ranges from 0.4 to 5712 g. Therefore, a log scale might be more appropriate for these data because it would bring the species with very large values closer to the others (a logarithm reduces large numbers more than small numbers). We can easily add the log-transformed data as new columns of the data frame as follows.

```
> massFrame$log_Body_mass <- log10( massFrame$Body_mass )
> massFrame$log_Brain_mass <- log10( massFrame$Brain_mass )
```

We added columns by simply assigning the output from the `log10` function to a new column name. We can see that the columns have been added by using the `names` function again.

```
> names(massFrame)
 [1] "Animal" "Body_mass" "Brain_mass" "log_Body_mass"
 [5] "log_Brain_mass"
```

Exercise 4.1

1. Calculate the total body mass and total brain mass of all 27 animals in the data frame. Hint: Use the `sum` function.
   Solution:
   total body mass = 119796 kg , total brain mass = 16541.6 g

2. Calculate the average body mass of all the animals with body mass larger than 100kg. Hint: Use the `mean` function.
   Solution:
   10854.74 kg

3. Calculate the average brain mass of all the animals with body mass larger than 100kg.
   Solution:
   1213.864 g

4.4 Writing data

In the previous section we log-transformed our data points and created new columns to hold them. We may wish to write the data frame to a file, so that we do not have to repeat this transformation every time we load the data. This is straightforward in R using the `write.table` function. Try writing the data to a file called "body_brain_mass2.txt" with the command

```
write.table(massFrame,"body_brain_mass2.txt")
```

The first argument is the data frame that we wish to write out and the second argument is the file name that we want to save it to. Your file will be created in the working directory that you specified before. If you want to write your data elsewhere, you have to specify the path in your argument to `write.table`.

Remember not to use spaces in file names as they can confuse R (and other programs). If you wish to have spaces in filenames, use the underscore character instead.

If you open up the file you created in the R text editor you will notice a few different features. Using the default options of `write.table`, R has written the row numbers for each data point and the column names. The fields (individual data points) are separated by a single white space and each field has been enclosed in quotes. Some of these features are not necessary, or even helpful. It is therefore recommended that you switch off the row names and the quoting and use a tab as the delimiter. This generally makes the table more readable in the text file. We can do this by specifying additional arguments,

```
write.table(massFrame,"body_brain_mass2.txt", row.names=F, quote=F,
    sep="\t")
```

If you now view the contents in the R text editor you should see it is a little easier to read.

How you format your data is a matter of taste and R has many options. Another commonly used option is to write the table to a comma-separated values (.csv) file. Similar to reading data, we can do this by specifying arguments to `write.table` or by using another version of that function with pre-set default arguments,

```
write.csv(massFrame,"body_brain_mass2.csv", row.names=F)
```

Look at the documentation, `help(write.table)` or `help(write.csv)` (same page), for more options.