A black and white logo

Description automatically generated

**StratoBayes Workshop**

**18th – 20th March 2024, Durham, UK**

**Worksheet for Day 1 Session 2:**

**Handling data: import and manipulation**

Adapted by Andrew Millard from materials prepared by Dr Sally Street,   
Department of Anthropology, Durham University.

This work is licensed under a   
[Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-nc-sa/4.0/).

[Creative Commons License](http://creativecommons.org/licenses/by-nc-sa/4.0/)

# Session 2: Handling data: import and manipulation

In this section we’ll tackle what can often be the most challenging step for new R-users: importing data and wrangling it into the right format for analysis. We’ll illustrate exercises using geochemical dataset from a section at Talat n’ Yissi in Morocco extracted from the published data of Maloof et al. (2005. Can. J. Earth Sci. 42: 2195-2216; 2010. Geol. Soc.Am. Bull.122:1731-1774). This is available at Talat\_n\_Yissi\_isotopes.csv

## Setting up a new R session

There are three things you will need to do routinely before starting any data analysis in a new R session: 1) create and save a new **script** file, 2) clear the **workspace** and 3) set a **working directory**.

1. Creating a script file

We’ve already seen how to create a **script** file – ordinarily it’s a good idea to organise your work by having one script file per project or set of analyses, but you can do this however makes most sense to you. So, you can either keep working on your script file from session 1 or create a new one for session 2.

2. Clearing the workspace

The **workspace** refers to all of the objects created and stored within a given session. When using RStudio, you can see all of the objects stored in the current session by looking at the top left window under the ‘Environment’ tab. You should see something like this:

A screenshot of a computer

Description automatically generated

Sometimes, stored objects can cause conflicts and unanticipated behaviour when moving between projects. It is very easy to forget what might be lurking in R’s memory and end up doing analyses on the wrong dataset! For this reason, it is always a good idea to clear R’s memory when starting a new script or set of analyses. You can do this by entering the command below:

rm(list=ls())

3. Setting a working directory

A **working directory** is a default folder that R will use to locate files within a given session. You’ll need to set one whenever you are going to be reading in external files, such as data tables. We’ll try this using data you’ve just downloaded. Move this file to a conveniently located folder, such as your desktop, or create a new folder specifically for this workshop. Then, you can choose a location using the function setwd[[1]](#footnote-2), with your chosen filepath inside the brackets (making sure to replace backslashes with forward-slashes if you are using Windows) e.g.

setwd("C:/Users/Andrew/Documents/SBWorkshop")

If you don’t know the filepath, navigate to the files, right-click and select ‘Properties’, and you’ll see it under ‘Location’ (on Windows 10). If you have trouble setting the working directory by entering commands, alternatively you can do it by clicking on a menu option in RStudio – select the ‘Session’ tab in RStudio, click ‘Set working directory’, then ‘Choose directory’. You should see a command like the one above pop up on the command line to confirm the working directory has changed successfully.

## Formatting data for R

R cannot easily import data directly from a spreadsheet program such as Excel. Instead, spreadsheet files should be converted to a plain text format such as .csv or .txt before you attempt to read them into R. Data should usually be formatted such that each row is a single observation and each column a single variable, reserving the top row for column headers if necessary.

You should format your data as clearly and simply as possible to avoid issues reading it into R. If you adhere to the suggestions below you should not encounter too many problems:

* Avoid special characters (use only letters, numbers, full-stops and underscores)
* Avoid spaces in the file or column names (use full-stops or underscores instead)
* Indicate missing data consistently with NA rather than leaving cells blank or using e.g. -999
* Including nothing else in the spreadsheet other than the data and column headers
* Input categorical variables as words, not numbers (e.g. lithology as ‘limestone’ or ‘sandstone’, not 1 and 2)
* Within a column, consistently input numbers as numbers and words as words
* Avoid importing data containing Excel formulae, macros or other complexities
* Do not use colour-coding to indicate category membership as this will not be read into R

Below is an example of some hypothetical data appropriately formatted in Excel for R:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sample\_ID** | **Depth** | **Lithology** | **Location** | **Subsampled** |
| 1 | 23 | limestone | Core 1 | Yes |
| 2 | 45 | limestone | Core 1 | Yes |
| 3 | 19 | sandstone | Core 1 | No |
| 4 | 42 | sandstone | Core 2 | No |
| 5 | 53 | NA | NA | NA |
| 6 | 63 | limestone | Core 2 | No |

And here is an example of the same data formatted badly in Excel for R:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sample\_ID** | **Depth** | **Lithology** | **Location** | **Subsampled** |
| 1 | 23 | limestone | Core 1 |  |
| 2 | 45 | limestone | Core 1 |  |
| 3 | Nineteen | sandstone | Core 1 |  |
| 4 | 42 | sandstone | Core 2 |  |
| 5 | 53 | - | Lost |  |
| 6 | 63 | limestone | Core 2 |  |

If it is important to you to use non-R friendly formatting in your raw data, you should keep two versions – one original spreadsheet file (with as much fancy formatting as you like) and one ‘clean’ copy for R. In Excel, you can convert .xlsx to plain text files just by clicking ‘Save As’ and selecting an appropriate format, such as comma-separated or tab-delimited text files.

## Importing data from spreadsheets

Assuming data are formatted appropriately for R, plain text files can be read into R fairly straightforwardly. The simplest way to do this is to use comma-separated (.csv) files, which can be imported into R using the function read.csv. Importing a dataset requires us to create a new object containing the data. Try for example reading in the Morocco data using the command below:

Talat\_isotopes <- read.csv("Talat\_n\_Yissi\_isotopes.csv", header = TRUE)

If the command worked, nothing will appear to happen, but you should see the new object ‘Talat\_isotopes’ appear in the Environment window. Helpfully, this will also report the numbers of rows (observations) and columns (variables).

**Note**: by default, read.csv assumes that there is just data, and when the first row of your data contains the column headers this has to be specified using header=TRUE.

It is absolutely essential at this stage that you check the data have been read in correctly, as mistakes with formatting and importing data are easy to make. Most fundamentally, you should check the version of the data that has been read into R against the original file, to make sure R has read in the correct number of rows and columns, in the correct format.

The functions str (‘structure’), head and tail are very useful for examining the data to make sure nothing has gone wrong. str can be used on a wide variety of different objects and tells you some basic information about what an object contains. For example, try using it on the imported data:

str(Talat\_isotopes)

And R should show you the following output:

A screenshot of a computer code

Description automatically generated

The top row tells you what kind of object R thinks it is (a data frame), and for data frames it will also report the number of rows and columns. Below, the column names are listed following the dollar signs. To the right of each variable name, str reports the type of variable R thinks each column contains, such as numeric (‘num’), integer (‘int’) or character (chr).

You should check carefully to make sure that variables are in the format you expect them to be in. Here, we can already see there might be some issues – R reports blank cells in the data (“”) as empty strings for some character variables (e.g. peaks), when these should in fact be read as missing data. Later on, we’ll cover how to handle missing data but for present purposes this is fine. Three columns did not have header values, so R has given them names of X, X.1 and X.2.

head and tail can be used to display the top and bottom few rows of your data, respectively. While not particularly useful by themselves, they can be helpful for spotting quickly if something has gone terribly wrong when importing the data. For example, try using head on the data:

head(Talat\_isotopes, n=10)

The n= argument indicates how many rows we wish to display, so you can adjust the value to whatever you wish. You can use tail in exactly the same way as head.

Next, we’ll cover several useful ways in which you can manipulate your data in R prior to statistical analysis: subsetting, transforming/creating variables, merging and handling missing data.

## Subsetting

The function subset lets you select subsets of your data based on a wide range of criteria. Subsetting in R involves creating a new object containing data that meet specified inclusion criteria. For example, let’s say we wanted to create a subset of the Talat data containing only certain lithologies. We can use the variable ‘formation’ for this purpose. From the output we saw when we used str on the data, we can see that formation is treated as a character string. We can ask R to report all the different values of a column using the unique function:

unique(Talat\_isotopes$formation)

R will display the names of all eight formations recorded in the dataset. Remember that the dollar sign in R is used to select a named column from within a dataframe.

Then, we could create a subset containing only e.g. Amouslek (K2c) using subset as follows:

K2c <- subset(Talat\_isotopes, formation == "Amouslek (K2c)")

If this works correctly, you’ll see the new data subset ‘K2c’ appear in the Environment window. You can then use str, head and tail on this object if you wish to examine it further – this is often a good idea as it will confirm if subsetting has worked as you expected it to.

You can select observations based on a range of different criteria using **relational operators**, which we’ve encountered before:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | exactly equal to |
| != | not equal to |
| !x | Not x |

Additionally, we can use **logical operators** to select criteria, including to combine multiple criteria:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| x | y | x OR y |
| x & y | x AND y |
| isTRUE(x) | test if x is TRUE |

For example, if you wanted to select only dolostone (dl lithology) with δ13C values above zero, you would use:

high\_d13C\_dl = subset(Talat\_isotopes, lithology == "dl" & d13C > 0)

**Note**: you do not have to name your objects the same as my suggestions. You can name them whatever you like and whatever makes most sense to you. You should, however, avoid spaces and special characters (but underscores are ok).

Alternatively, you can subset using indexing, but this can be a little trickier at first. When using indexing on a data frame, you can either index across the entire dataset using the following format:

*data[criteria]*

Or you can index based on criteria applied to rows and columns separately, using this format:

*data[row\_criteria, column\_criteria]*

As for vectors, indexing on dataframes can be done based on numeric positions. The command below, for example, would select rows 1 to 5 of the third column in the dataset:

Talat\_isotopes[1:5, 3]

Typically, however, it is more useful and convenient to select cases based on whether they meet specific criterion, using relational and/or logical operators. Here’s how we would extract a subset of the dataset from only the Issafene (K3a) formation, using indexing rather than the subset function:

K3a <- Talat\_isotopes[Talat\_isotopes$formation == "Issafene (K3a)", ]

Because we want to retain only those cases whose *rows* contain the formation name “Issafene (K3a)”, here the selection criterion inside the square brackets should go before the comma. We leave the column criterion (after the comma) blank in this instance, because here we want to retain *all* columns in the data subset. Keep an eye on this comma – it is very easy to forget it or put it in the wrong place.

### Thinning the data

Sometimes there is simply more data than you need. A large dataset may take a long time to complete a complex calculation like those used in StratoBayes, and for the purposes of creating and checking a workflow we may want to work with only a representative subset of the data. This can be done using indexing.

To select every 5th row from the isotopes dataset we can use:

Talat\_isotopes\_fifth <- Talat\_isotopes[seq(from = 1, to = length(Talat\_isotopes$sample), by = 5), ]

The seq function produces a sequence of integers from the first argument to the second argument in steps of the third argument. Use of the argument names is optional.

To make a random selection of 50 samples from the dataset we can use:

Talat\_isotopes\_50 <- Talat\_isotopes[sample(length(Talat\_isotopes$sample), 50), ]

The sample function randomly samples the number of integers given by the second argument between 1 and the first argument. If you examine this object you will see that the order of the samples has changed. To avoid this we can sort the set of integers before using it:

Talat\_isotopes\_50 <- Talat\_isotopes[sort(sample(length(Talat\_isotopes$sample), 50)), ]

## Transforming & creating new variables

### Continuous variables

You can transform your variables in any number of useful ways in R. You can do so by performing some function on a column of data, and either over-writing the existing data with the transformed data, or creating a new column to contain the transformed data. I would always recommend doing the latter to avoid any confusion down the line.

For example, let’s say that (for whatever reason) we wanted to round the δ18O values from each sample to one decimal place. We could do so as follows, using the round function:

Talat\_isotopes$d18O\_rounded <- round(Talat\_isotopes$d18O, digits = 1)

You can change the number of decimal places by changing the number specified by the digits argument. To confirm that this has worked as expected, you can use str or head/tail to examine the data.

You can transform your variables in numerous other ways using functions, including natural log-transformation (log) and square-root transformation (sqrt), for example.

You can also use arithmetic operators to transform your variables, including:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| ^ | Exponentiation |

For example, imagine that we wanted to report positions in centimetres, rather than metres. We could do so as follows:

Talat\_isotopes$depth\_cm <- Talat\_isotopes$m \* 100

### Categorical variables

When preparing categorical data for analysis, you will often wish to re-classify your variables by lumping categories together. One way to do this in R is by using if-else statements, with the function ifelse (although this can be a bit of a headache…)

For example, the data has stages A, B, and T, but we might wish to classify stages A and B as the lower part of the section. If so, we need to create an if-else statement that says the sample is the Lower part if it is in stage A or B and in the Upper part otherwise, as follows:

Talat\_isotopes$Section\_part <- ifelse(Talat\_isotopes$stage == "A" | Talat\_isotopes$ stage == "B", "Lower", "Upper")

So now we have the new variable ‘Section\_part, which is “Lower” for samples in stages A and B, and “Upper” for all other stages.

**Note:** when using ifelse statements, watch out for missing data as these will needed to be treated separately (i.e. R will need to know how to classify these in the new variable).

You can use ifelse statements to re-code categorical variables in numerous other ways, e.g. splitting categories into finer-grained groups or re-naming factor levels.

### Changing variable types

We can also transform variables from one type to another (i.e. between numeric, integer, and character), if appropriate. One common such transformation is splitting a continuous variable at some point, such as the mean or median, to create new binary categorical variable. Let’s say for example that we wanted to treat the variable ‘d18O’ as a categorical variable with two levels, “High” and “Low”. We could decide that all δ18O values greater than or equal to the median are to be categorised as “High” and those less than the median “Low”. We could do this using the following if-else statement:

Talat\_isotopes$d18O\_binary <- ifelse(Talat\_isotopes$d18O >= median(Talat\_isotopes$d18O), "High", "Low")

## Merging

Next, we’ll walk through how to merge two datasets together by a common identifier. In the study of Talat n’ Yissi, selected samples were analysed elemental composition and this data is in Talat\_n\_Yissi\_elements.csv For this exercise, you will need first to import the elemental data. We will be merging this dataset with the main isotope dataset using matching sample numbers.

Now read in the elemental dataset using read.csv. Go back to the commands we covered earlier on in this session, copy-paste them and change the file name appropriately to do this.

Once you’ve done this, we can merge the isotope and elemental datasets together using the merge function, based on their common sample numbers. The command below assumes you’ve read in the elemental data successfully and named it ‘Talat\_elements’:

Talat\_merged <- merge(Talat\_isotopes, Talat\_elements, by="sample")

Now use str on the merged dataset. From the output, you will see that the dataset contains a smaller number of rows (26) than the original isotope data (479) – this is because not all of the sample IDs match between the datasets. By default, merge assumes that you want to drop all non-matching cases, but if you want to keep them, you can do so by adding extra arguments. all=TRUE will keep all non-matching cases, while all.x=TRUE will keep all cases in the first dataset even if they cannot be matched with the second, and vice versa for all.y=TRUE.

## Handling missing data

### Re-coding missing data

Missing data can be represented in a variety of different ways in spreadsheets (e.g. using NA, -999, blank cells, etc.). By default, R understands ‘NA’ to mean missing, so you will run into problems if you use anything other than NA for missing values. For example, see what happens when you run the command below, which checks if any missing values in the ‘peaks’ column are missing using the any and is.na functions together:

any(is.na(Talat\_isotopes$peaks))

**Note:** the command above illustrates how functions can be nested within one another in R to perform multiple operations in a single line of code – this can be extremely useful. Nested functions are performed from inside out: here, the function “any” is applied to the result of “is.na”.[[2]](#footnote-3)

R returns ‘FALSE’, indicating there are no missing values. However, there are in fact only two entries in that column! We can confirm this by combining the length and which functions to report how many observations in the peak column are equal to "" (i.e. are blank).

length(which(Talat\_isotopes$peaks == ""))

First we’ll try directly manipulating the data using indexing, which will allow us to find and replace blank cells with ‘NA’ for the object containing the isotopes dataset. We can do this using the command below, which assigns NA to any cells meeting the criteria specified in the square brackets (equal to ‘blank’):

Talat\_isotopes[Talat\_isotopes == ""] <- NA

If you now examine the data using head or tail, you’ll see that previously blank cells are now indicated by NA, and using the any or length commands we used above gives different results.

**Important:** this strategy involves directly manipulating the data frame (i.e. over-writing) so you should be **very** careful to check nothing unexpected happens along the way when using this approach. However, if things do go wrong, you can always re-import the data and start over. The original files are completely unaffected.

Alternatively, when we import the data we could add an argument to instruct R to convert blank cells to NAs, as follows:

Talat\_isotopes <- read.csv("Talat\_n\_Yissi\_isotopes.csv, na.strings="")

### Removing missing data

Once we’re sure that R is interpreting missing data correctly, we might want to remove it prior to analysis. We can do so either by removing rows missing observations for specific variables or by removing rows missing data for *any* variables.

For example, to select a subset containing only those observations with peaks labelled, we could use:

Talat\_isotopes\_with\_peaks <- subset(Talat\_isotopes, !is.na(peaks))

Remember that the exclamation point in R stands for negation, so here we are creating a subset that contains only observations for which the peak is NOT missing (NA).

‘peaks’, however, is not the only column with missing data. If you wanted to drop rows missing data from multiple columns, you can do so using the AND operator, for example:

Talat\_isotopes\_one\_missing <- subset(Talat\_isotopes, !(is.na(peaks) & is.na(X)) )

Finally, if you want to exclude missing data altogether, you can use the complete.cases function to select only observations with data for all variables as follows:

Talat\_isotopes\_complete <- Talat\_isotopes[complete.cases(Talat\_isotopes), ]

**Note:** in this instance, the complete dataset will contain no rows, because no row of the data is complete!

## Session 2: questions and exercises

**1.** In R, what should you use to represent missing values in data?

**A.** na

**B.** NaN

**C.** NA

**D.** nan

**2.** What symbol should you use to select a named column from a data frame?

**A.** $

**B.** £

**C.** &

**D.** #

**3.** Which of the following means ‘is equal to’?

**A.** !=

**B.** <-

**C.** =

**D.** ==

**4.** Which of the following represents OR?

**A.** &

**B.** I

**C.** \

**D.** /

**5.** What is contained in the 25th row of the 7th column of the elements dataset? Use indexing to find the answer (0.2).

**6.** How many samples in the dataset have Low δ18O and are in the Higher part of the section? Use subsetting to find the answer.

**7.** How many samples in the elements dataset have Ca concentrations over 30% and Sr concentrations over 400 ppm? Use an ifelse statement to find the answer.

**8. I**n the isotope dataset, how many dolostone (dl) lithology samples are in the Igoudine formation?

**9.** The command below attempts to extract the sample numbers from the isotope dataset. Why does it return ‘NULL’?

Talat\_isotopes$Sample

**10.** The command below is an attempt to select only samples from Stage A from the isotopes dataset. Why does it return an error?

Stage\_A <- Talat\_isotopes[Talat\_isotopes$stage == "A"]

1. Note: if you wish to share your script with others in the interests of reproducibility, leaving the setwd command in the script is probably not a good idea as it will work only on your specific machine. [↑](#footnote-ref-2)
2. If you are more used to reading left-to-right, is.na(Talat\_isotopes$peaks) |> any() is an equivalent way to “pipe” the output of “is.na” to the function “any”. [↑](#footnote-ref-3)