University of Glasgow

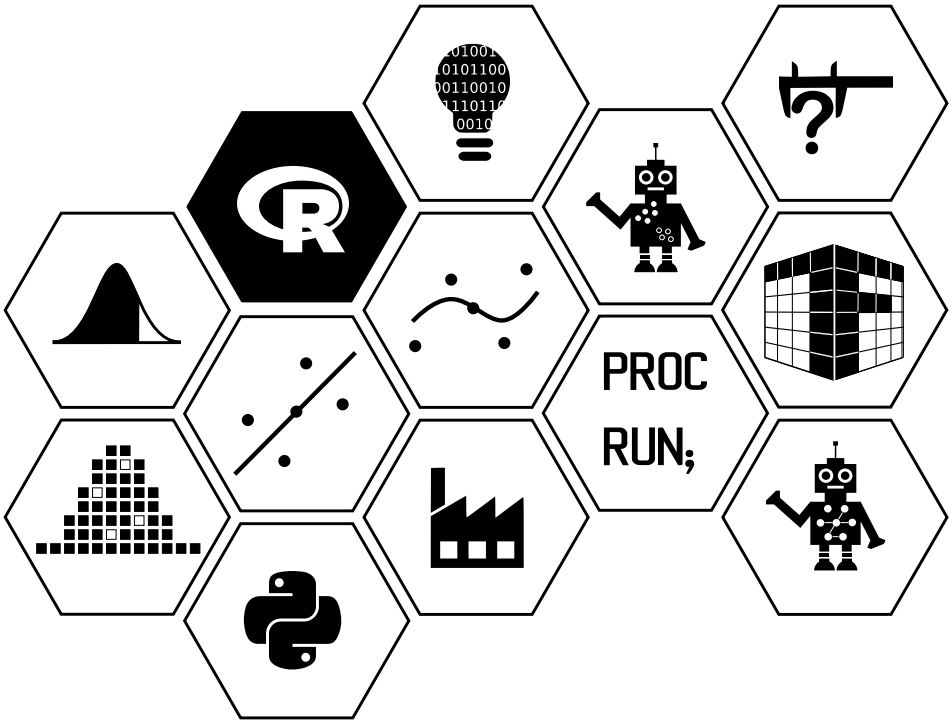
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

Week 3:

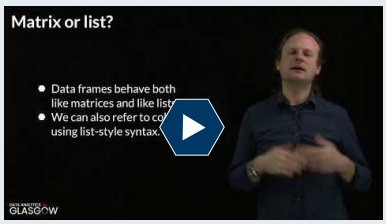
Basic Data Management in R



DATA ANALYTICS GLASGOW

Data frames

Data frame basics



Data frames

https://youtu.be/SSei0Sf_nu0

Duration: 10m58s

Why data frames? Last week we learned how to create matrices. In principle, we could use matrices to store data sets. However, matrices (and vectors) have one important constraint: all entries of a vector or a matrix have to be of the same data type. Many data sets we work with have both numerical and factor variables, so we would need to store our data using different data types for the different columns. We could in theory use lists to manage such data. However, lists do not enforce the constraint that each variable has the same number of observations, so if we used lists our data would soon get messy.

Let's look at an example illustrating this limitation of matrices.

Consider a matrix `kids` containing age, weight and height of two children.

```
kids <- rbind(c( 4, 15, 101),
              c(11, 28, 132))
colnames(kids) <- c("age", "weight", "height")
rownames(kids) <- c("Mary", "John")
kids
```

```
##      age weight height
## Mary   4     15    101
## John  11     28    132
```

We can for example see that John is older than Mary.

```
kids["John", "age"] > kids["Mary", "age"]
```

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

Let's now try adding a column gender.

```
kids2 <- cbind(kids, gender=c("f", "m"))
kids2
```

```
##      age  weight height gender
## Mary "4"  "15"   "101"  "f"
## John "11" "28"   "132"  "m"
```

Let's see whether John is still older than Sarah.

```
kids2["John", "age"] > kids2["Mary", "age"]
```

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

Not any more. How come? We have added the variable `gender`, which is a character vector. As all the data in the matrix needs to be of the same data type, R had to convert all the columns, including `age` to character strings, and character strings do not compare in the same way as numbers: in a dictionary `11` would be before `4`. We can see from the quotes around the numerical variables that these have been converted to characters (accidental conversion to factors is slightly more difficult to spot, as R prints factors without quotes).

Because of situations like this one it is better to use data frames (or tibbles, which we look at next week) to store data sets.

Creating data frames A matrix can be converted to a data frame using `as.data.frame` and we can convert a data frame back to a matrix using `as.matrix`.

```
kids <- as.data.frame(kids)
kids <- cbind(kids, gender=c("f","m"))
kids
```

```
##      age weight height gender
## Mary   4     15    101      f
## John  11     28    132      m
```

```
kids["John", "age"] > kids["Mary", "age"]
```

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

A data frame can handle columns of different data types, so the numeric column is not converted when adding a character (or factor) column.

Data frames can be created using the function `data.frame`, so we could have created the data set using

```
kids <- data.frame(age=c(4,11), weight=c(15,28), height=c(101,132), gender=c("f", "m"))
rownames(kids) <- c("Mary", "John")
```

or

```
kids <- rbind(Mary=data.frame(age=4, weight=15, height=101, gender="f"),
              John=data.frame(age=11, weight=28, height=132, gender="m"))
```

A data frame has row names and column names like a matrix and these can be set in the same way as for matrices. Data frames can also be subset like matrices. We will come back to this later on.

Data frames also behave like lists Data frames behave not just only like matrices, they also behave like lists (with the columns being the entries). We can use $ to access columns:

```
kids$age
```

```
## [1]  4 11
```

So for a data frame, the following four lines of R commands are all equivalent

```
kids$age
```

```
## [1]  4 11
```

```
kids[,"age"]
```

```
## [1]  4 11
```

```
kids[,1]
```

```
## [1]  4 11
```

```
kids[["age"]]
```

```
## [1]  4 11
```

```
kids[[1]]
```

```
## [1]  4 11
```

As mentioned previously, it is always better to refer to columns by their name rather than their index. The latter is too likely to change as you work with the data.

We can use the same notation to set values in the data frame. Suppose it was John's birthday and we want to change his age to 5. We can use any of the lines below (and there are even more possible ways).

```
kids["John", "age"] <- 5
kids$age[2] <- 5
kids[2,1] <- 5
kids[[2]][1] <- 5
```

Again, the command at the top is probably best, because it is the most "human-readable".

Data manipulation

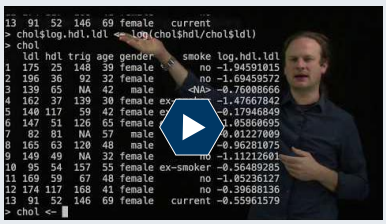
In this section we will now look at various R functions for manipulating data frames. This week we look at the functions available in base R. Next week, we look at the more elegant and powerful functions from tidyverse.

In this section we work with a toy data set called `chol`, which you can load into R using

```
load(url("https://github.com/UofGAnalyticsData/R/blob/main/Week%203/chol.RData?raw=true"))
```

The data set contains (simulated) blood fat measurements from a small number of patients.

Adding new columns



Transformations

https://youtu.be/XJE9x9qW0zk

Duration: 3m58s

Suppose we want to add to our data set a new column called `log.hdl.ldl` which contains the logarithm of the ratio of HDL and LDL cholesterol.

Using what we have seen so far we could use

```
chol <- cbind(chol, log.hdl.ldl=log(chol[,"hdl"]/chol[,"ldl"]))
```

or

```
chol[,"log.hdl.ldl"] <- log(chol[,"hdl"]/chol[,"ldl"])
```

or

```
chol$log.hdl.ldl <- log(chol$hdl/chol$ldl)
chol
```

```
##    ldl hdl trig age gender     smoke log.hdl.ldl
## 1  175  25  148  39 female        no -1.94591015
## 2  196  36   92  32 female        no -1.69459572
## 3  139  65   NA  42   male      <NA> -0.76008666
## 4  162  37  139  30 female ex-smoker -1.47667842
## 5  140 117   59  42 female ex-smoker -0.17946849
## 6  147  51  126  65 female ex-smoker -1.05860695
## 7   82  81   NA  57   male        no -0.01227009
## 8  165  63  120  48   male   current -0.96281075
## 9  149  49   NA  32 female        no -1.11212601
## 10  95  54  157  55 female ex-smoker -0.56489285
## 11 169  59   67  48 female        no -1.05236127
## 12 174 117  168  41 female        no -0.39688136
## 13  91  52  146  69 female   current -0.55961579
```

The first two commands work for data frames and matrices whereas the last one only works for data frames.

All of the above commands look slightly messy and use the name of the data set more than once, which is a potential source of mistakes when you want to change the name of the data set in the future.

It is generally better to use the functions `transform`. Its main advantage is that we do not need to put `chol\$` everywhere.

```
chol <- transform(chol, log.hdl.ldl=log(hdl/ldl))
```

Removing columns

You can use the subsetting techniques for matrices to remove columns. For example,

```
chol <- chol[,-3]
```

removes the column `trig` (third column). Alternatively, you could use list-style syntax:

```
chol[[3]] <- NULL
```

or

```
chol$trig <- NULL
```

Supplementary material:
NULL

The null object in R is `NULL`. You can test whether an object is null by using `is.null(object)`. However, `object==NULL` does *not* work.

Subsetting data sets / Selecting observations

Data frames can be subset just like matrices. For example, to remove all patients who have never smoked and store the result in a data frame called `chol.smoked` we can use

```
chol.smoked <- chol[chol$smoke!="no",]
chol.smoked
```

```
##    ldl hdl age gender     smoke log.hdl.ldl
## NA  NA  NA  NA   <NA>      <NA>          NA
## 4  162  37  30 female ex-smoker  -1.4766784
## 5  140 117  42 female ex-smoker  -0.1794685
## 6  147  51  65 female ex-smoker  -1.0586070
## 8  165  63  48   male   current  -0.9628107
## 10  95  54  55 female ex-smoker  -0.5648928
## 13  91  52  69 female   current  -0.5596158
```

Because there is a missing value in the smoking status, the new data set starts with a row of missing values. This is due to the fact that `chol$smoke!="no"` has as third entry `NA`.

```
chol$smoke!="no"
```

```
##  [1] FALSE FALSE    NA  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE
## [12] FALSE  TRUE
```

For every `NA` in the condition, R will put a row of `NA`'s at the top of the data set.

We then have to remove the row(s) with missing values, which is best done using the function `na.omit`

```
chol.smoked <- na.omit(chol.smoked)
```

We can subset the data frame in one go when we use the function `subset`. Just like `transform` it provides a cleaner solution because it does not require us to use `chol\$` in the condition.

```
chol.smoked <- subset(chol, smoke!="no")
chol.smoked
```

```
##    ldl hdl age gender     smoke log.hdl.ldl
## 4  162  37  30 female ex-smoker  -1.4766784
## 5  140 117  42 female ex-smoker  -0.1794685
## 6  147  51  65 female ex-smoker  -1.0586070
## 8  165  63  48   male   current  -0.9628107
## 10  95  54  55 female ex-smoker  -0.5648928
## 13  91  52  69 female   current  -0.5596158
```

Task 1.

Create a data frame `chol.lowhdl` containing the data for patients with a HDL cholesterol of less than 40 mg/dl.

chol.lowhdl <- subset(chol, hdl<40)

Sorting data sets

The function `order` can be used to sort an entire data set by one column. The data frame chol can be sorted by the age of the patient using

```
permut <- order(chol$age)      # Create the permutation order by age
chol <- chol[permut,]          # Apply this permutation to the entire data set
chol
```

```
##    ldl hdl age gender     smoke log.hdl.ldl
## 4  162  37  30 female ex-smoker -1.47667842
## 2  196  36  32 female        no -1.69459572
## 9  149  49  32 female        no -1.11212601
## 1  175  25  39 female        no -1.94591015
## 12 174 117  41 female        no -0.39688136
## 3  139  65  42   male      <NA> -0.76008666
## 5  140 117  42 female ex-smoker -0.17946849
## 8  165  63  48   male   current -0.96281075
## 11 169  59  48 female        no -1.05236127
## 10  95  54  55 female ex-smoker -0.56489285
## 7   82  81  57   male        no -0.01227009
## 6  147  51  65 female ex-smoker -1.05860695
## 13  91  52  69 female   current -0.55961579
```

We will see more elegant code for sorting data sets when we look at `tidyverse` next week.

Supplementary material:
Attaching data frames

To access the column `height` in the data frame `kids` we have to use `kids$height` (or one of the above equivalent statements). Sometimes it would be easier just to refer to it as `height` (without the `kids$`) as we have done inside `transform` or `subset`. This can be done using the function `attach`. After calling

```
attach(kids)
```

we can access the columns of `kids` as if they were variables, i.e. we can use

```
weight / (height/100)^2
```

```
## [1] 14.70444 16.06979
```

instead of

```
kids$weight / (kids$height/100)^2
```

```
## [1] 14.70444 16.06979
```

To undo the effects of `attach` simply use

```
detach(kids)
```

However, `attach` behaves in an unexpected way if you try to change any of the variables of a data frame you have already attached. The problem with `attach`'ed data is that it exists in R twice. Once as inside the data frame (where it was before you called `attach`) and once as a variable in your current environment. R does however *not* link those two. If you change one of them, the other one does *not* get updated automatically.

Suppose we now want to change the weight from kgs to pounds, i.e. divide it by 0.45359237. If you now use

```
weight <- weight / 0.45359237
```

to change the unit, you have changed the unit only for the `attach`'ed variable.

```
weight
```

```
## [1] 33.06934 61.72943
```

```
kids$weight
```

```
## [1] 15 28
```

`weight` contains the weight in pounds, whereas `kids$weight` still contains the "old" weight in kgs.

If we had used

```
kids$weight <- kids$weight / 0.45359237
```

the opposite would have happened. `weight` would have contained the old version (in kgs), and only `kids\$weight` would have contained the new version (in pounds). In other words, whatever we do, we will end up with inconsistent data.

Thus, it is probably a good idea to avoid using `attach`. If you use `attach` remember this important rule: Never manipulate `attach`'ed data!

The function `with` is a safer alternative to `attach`. We could use

```
with( kids, {
  weight <- weight / 0.45359237
} )
```

to transform the column `weight` from kgs to pounds. In scripts using `with` is actually clearer than `attach`. However in the interactive console, `with` is slightly more awkward to use. Note that in this example it would have been easiest to use `transform`.

```
kids <- transform(kids, weight=weight / 0.45359237)
```

Task 2.

The data frame `cia` contains data about almost all countries taken from the CIA World Factbook.

You can load the data frame into R using

```
load(url(paste("https://github.com/UofGAnalyticsData/R/blob",
               "/main/Week%203/cia.RData?raw=true",sep="")))
```

We will use the following columns.

Variable	Content
`Country`	Name of the country
`Continent`	Geographic region the country is located in
`Population`	Population
`Life`	Life expectancy at birth in years
`GDP`	Gross domestic product in USD
`MilitaryExpenditure`	Military expenditure in USD

(a) Delete all observations for which the population is missing. You might find the functions `is.na` or `complete.cases` useful. (You can get R to show the help for these functions by entering `?is.na` or `?complete.cases`).

(b) Which countries have a population of less than 10,000 inhabitants?

(c) Which countries have a military expenditure of at least 8% of the GDP?

(d) Ignoring missing values, what is the combined GDP of all European countries?

(e) Create a new column `GDPPerCapita`, which contains the per capita GDP (`GDP`) divided by `Population`. Also create a new column `MilitaryExpPerCapita`, which contains the per capita military expenditure `MilitaryExpenditure` divided by `Population`).

(f) Which country has the highest life expectancy?

(g) Which ten countries have the highest life expectancy?

Merging data sets

Often, the data required for a certain analysis is stored in more than one data frame. Many transactional databases are designed using the so-called "third-normal form" design principle, which, despite being optimal from a transactional point of view, requires combining many tables before being able to analyse the data. R has a built-in command `merge`, which allows for merging tables using common keys. Next week, we will look at the package dplyr, which has more powerful functions for combining data sets. When possible, it is however often easier to combine the information required for analysis using SQL and the database engine before importing the data into R.

In this section we will look at the function `merge` built into R.

Consider the following example. In a study of 2,287 eighth-grade pupils (aged about 11) a language test score and the verbal IQ were determined. The question of interest is whether the socio-economic status (SES) of the parents and various characteristics of the class influence the language test score and the verbal IQ. The data is stored in two data frames, one containing the data about the children (`children`) and one containing the data about the different classes (`classes`).

The first few rows of `children` are:

```
load(url(paste("https://github.com/UofGAnalyticsData/R/blob/main",
"/Week%203/children_classes.RData?raw=true",sep="")))
head(children)
```

```
##   lang   IQ SES class
## 1   46 15.0  23   180
## 2   45 14.5  10   180
## 3   33  9.5  15   180
## 4   46 11.0  23   180
## 5   20  8.0  10   180
## 6   30  9.5  10   180
```

The first few rows of `classes` are:

```
head(classes)
```

```
##   class size combined
## 1   180   29     TRUE
## 2   280   19     TRUE
## 3  1082   25     TRUE
## 4  1280   31     TRUE
## 5  1580   35     TRUE
## 6  1680   28     TRUE
```

There are good reasons for storing the data in this format. This way, less redundant information is stored and the information about each class is stored exactly once. This makes it easier to change class properties like for example the number of pupils.

However, for the analysis of the data it is necessary that we "copy" all information from the data frame `classes` into the data frame `children`: for every child we need to look up which class it belongs to and copy the information about

that class into the row belonging to that child. Of course we cannot simply use `cbind`: the child in the second row of the data frame `children` attended class "180", which is described in the first row of `classes`.

The function `merge` can be used for such a task. By default, `merge` merges data sets using the columns both data frames have in common (in our case `class`)

```
data <- merge(children, classes)
```

It is typically better to explicitly specify which column(s) are to be used for merging the data frames. This avoids that columns that happen to have the same name in both data frames, but are not related are used in the merger. The common key(s) to be used for merging can be specified using the argument `by`, provided they have the same names in both data frames.

```
data <- merge(children, classes, by="class")
```

The arguments `by.x` and `by.y` allow merging data frames using columns which do not have the same name in both data frames (in our case they of course do have the same name).

```
data <- merge(children, classes, by.x="class", by.y="class")
```

For each child `merge` has looked up the information about the class and added it to the row in the resulting data frame `data`.

```
head(data)
```

```
##   class lang   IQ SES size combined
## 1   180   46 15.0  23   29     TRUE
## 2   180   45 14.5  10   29     TRUE
## 3   180   33  9.5  15   29     TRUE
## 4   180   46 11.0  23   29     TRUE
## 5   180   20  8.0  10   29     TRUE
## 6   180   30  9.5  10   29     TRUE
```

If there were children in a class for which there is no entry in `classes` R would by default remove these from the resulting data frame. Similarly, data from classes for which there are no pupils in `children` will also not appear in the resulting table. If you are familiar with SQL this corresponds to the default `INNER JOIN`.

If you want the resulting data frame to contain all cases from the first data frame even if there is no matching entry in the second data frame (`LEFT JOIN` in SQL speak), you need to specify the additional argument (`all.x=TRUE`). If you want the resulting data frame to contain all cases from the second data frame even if there is no matching entry in the first data frame (`RIGHT JOIN` in SQL speak), you need to specify the additional argument (`all.y=TRUE`).

Task 3.

Consider two data frames `patients` and `weights`, which you can load into R using

```
load(url(paste("https://github.com/UofGAnalyticsData/R/blob/",
"main/Week%203/patients_weights.RData?raw=true",sep="")))
```

The first few rows of the data sets are

```
head(patients)
```

```
##   PatientID Gender Age   Smoke
## 1         1   male  33      no
## 2         2 female  32      no
## 3         3   male  67      ex
## 4         4   male  36 current
## 5         5 female  47 current
```

```
head(weights)
```

```
##   PatientID Week Weight
## 1         1    1     72
## 2         1    2     74
## 3         1    3     71
## 4         2    1     54
```

```
## 5          2     3      54
## 6          3     1      96
```

Merge the data sets such that there is information about the patient for each weighting.

Importing and exporting data from R

Native .RData files

R has an internal binary data format (".RData files"), which can be used to store one or more objects. Typically `.RData` or `.rda` is used as the file extension. The key advantage of using R's internal format is that it stores objects exactly as they are in R. If you load the objects back in R you are guaranteed that they are exactly reproduced.

If you want to save a object `x` to a file you can use

```
save(x, file="MyX.RData")
```

If you want to save more than one object (say `x` and `y`) you can use

```
save(x, y, file="MyXY.RData")
```

If you want to save all objects in your workspace to a file (`MyVariables.RData`), you can use

```
save.image(file="MyVariables.RData")
```

To load the data you have saved back into R use

```
load(file="MyVariables.RData")
```

When saving the workspace at the end of a session, R simply stores all objects in your workspace in a file `.RData` in your home directory.

R's internal format is a good idea if you only work with R. However, very few other software products support it.

Path names on Windows

Path names in Windows typically contain backslashes (\). In R you need to either use a forward slash instead (for example `c:/Users/Craig/data.RData`) or escape the backslash, i.e. use a double `\\` (for example `c:\\Users\\Craig\\data.RData`).

Table data
Text and CSV files

Reading in data

https://youtu.be/OWR6DJKpz3A

Duration: 5m56s

Importing data into R In most cases data is stored in a table or spreadsheet format. The easiest way of reading external data into R is to use delimited text files. If the data at hand is say an Excel spreadsheet it is typically easier to open it in Excel first and convert it to a text (or CSV) file in Excel, and only then open the text file in R.

Before reading a text file into R, it is always a good idea to look at the file first using a raw text editor and determine the following information, which is easy for you to determine, but hard for R to guess automatically:

- Column names: Does the first line of the file contain data or does it contain the names of the columns ("variables")?
- Delimiter: What character is used to delimit the columns? This is typically white space, tabulator, `,` or `;`.
- Missing values: Determine how missing values are encoded (if there are any). R uses `NA`, but `*` and `.` are common as well.

Below are the first few lines of the file chol.txt.

```
175 25 148 39 female no
196 36 92 32 female no
139 65 NA 42 male NA
```

We can see that:

- The first line contains data and *not* the names of the columns.
- The columns are delimited using white space.
- The data set uses `NA` to encode missing values.

Such white space (or tab) delimited files can be read in using the R function `read.table`. It assumes by default that the first line of the file does *not* contain the column names (i.e. the first line already contains data), that white space is used as delimiter, and that missing values are encoded as `NA`. If this is not the case, you need to use the following additional arguments:

- `header`: Use `header=TRUE` if the first line of the file contains the names of the columns.
- `sep`: If the delimiter of the columns is not white space, but another character, you need to use the additional argument `sep`. For comma-separated data, use `sep=","`. The latter is better read in using the function `read.csv`.
- `na.strings`: If missing values are encoded using strings other than "`NA`", you need to use the additional argument `na.strings`. If for example "`*`" is used to denote missing values, you would use `na.strings="*"`. The argument `na.strings` can be a character vector if more than one string is used to denote missing values. You do not need to use this argument if your data set does not contain any missing values.
- `dec`: You can use the additional option `dec` to set the decimal separator (e.g. `dec=','`)

The file `chol.txt` uses no column names, white space as a separator, and uses "`NA`" for missing values. Thus we do not need to use any additional arguments to read it into R.

```
chol <- read.table("chol.txt")
head(chol)
```

```
##    V1  V2  V3 V4     V5        V6
## 1 175  25 148 39 female        no
## 2 196  36  92 32 female        no
## 3 139  65  NA 42   male      <NA>
## 4 162  37 139 30 female ex-smoker
## 5 140 117  59 42 female ex-smoker
## 6 147  51 126 65 female ex-smoker
```

It is always worth looking at the first few lines of the data you have read in to make sure it was read in correctly.

If, like in our example, the data file does not contain variables, it is a good idea to set them right after you have read in the data.

```
colnames(chol) <- c("ldl", "hdl", "trig", "age", "gender", "smoke")
```

R tries to guess of what type each column/variable is, but might not always get it right. It also is worth checking that each column was read in as the data type you had intended. This can be done using

```
sapply(chol, class)
```

```
##         ldl         hdl        trig         age      gender       smoke
##   "integer"   "integer"   "integer"   "integer" "character" "character"
```

Alternatively we could use

```
str(chol)
```

```
## 'data.frame':    13 obs. of  6 variables:
##  $ ldl   : int  175 196 139 162 140 147 82 165 149 95 ...
##  $ hdl   : int  25 36 65 37 117 51 81 63 49 54 ...
##  $ trig  : int  148 92 NA 139 59 126 NA 120 NA 157 ...
##  $ age   : int  39 32 42 30 42 65 57 48 32 55 ...
##  $ gender: chr  "female" "female" "male" "female" ...
##  $ smoke : chr  "no" "no" NA "ex-smoker" ...
```

If a variable which you had intended to be numeric (or an integer) shows up as a factor it is likely that missing values (or other error codes) were in the data that were not read in correctly.

The file chol.csv contains the same data, however in comma-separated ("CSV") format (Note - you may need to right click "Save As" when opening this link to obtain the chol.csv file). The first few lines of this file are

```
ldl,hdl,trig,age,gender,smoke
175,25,148,39,female,no
196,36,92,32,female,no
```

```
139,65,.,42,male,.
```

We can see that:

- The first line of the file are the column names.
- The columns are delimited using commas.
- The missing values are coded using `.`.

Thus we can read the file into R using

```
chol <- read.table("chol.csv", header=TRUE, sep=",", na.strings=".")
```

or

```
chol <- read.csv("chol.csv", na.strings=".")
```

`read.csv` is a sibling of `read.table` with the main difference that it assumes by default that the data is comma-separated and that the first line contains the variable names. In other words, you do not need to specify `sep=","` and `header=TRUE"` when using `read.csv`.

Task 4.

Read the data files `cars.csv` and `ships.txt` into R.

You can download the two files from:

- https://github.com/UofGAnalyticsData/R/raw/main/Week%203/cars.csv
- https://github.com/UofGAnalyticsData/R/raw/main/Week%203/ships.txt

(Note - you may need to right click "Save As" when opening this link to obtain the cars.csv file)

Exporting data from R Text files can be used as well to export data from R using the function

```
write.table(chol, file="chol.csv", sep=",", col.names=TRUE, row.names=FALSE)
```

The arguments `col.names` and `row.names` can be used to choose whether the column names and row names should be exported as well. The function `write.csv` can be used instead of the additional argument `sep=","`.

```
write.csv(chol, file="chol.csv", row.names=FALSE)
```

Other file formats There are many R packages that allow reading in data stored in file formats used by other software products. In most cases it is best to first convert the file into a text file using the proprietary software the file format corresponds to and then open the exported text file in R. However, there are R packages which allow opening various different file formats.

Package	Functions	Formats that can be read in
`readxl`	`read_excel`	Excel spreadsheets (`.xls` and `.xlsx`)
`xlsx`	`read.xlsx`, `write.xlsx`	Excel spreadsheets (only `.xlsx`)
`foreign`	`read.xport`, `write.xport`	SAS XPORT format
`foreign`	`read.dta`, `write.dta`	Stata binary files
`foreign`	`read.spss`	SPSS files

SQL databases If the data you want to work with data which is stored in a relational database it is typically best to connect straight to that database. The R package DBI provides a high-level interface allowing direct connections to various database management systems: SQLite, MySQL and MariaDB, PostgreSQL, Oracle, etc. It can also work using Microsoft's ODBC or Java's JDBC interface.

We will look at an example using a temporary in-memory SQLite database (so to run the example below you do not need any additional software other than the R packages used). We will use the babynames data as an example. Because we start with an empty database we first move the `babynames` table from R into the SQL database (in practice your data would of course already be in the database). You will learn how to write SQL queries in the *Data Management and Analytics using SAS* course.

```
library(DBI)                                        # Load required packages
library(RSQLite)
con <- dbConnect(RSQLite::SQLite(), ":memory:")     # Connect to temporary database
library(babynames)                                  # Load data package
dbWriteTable(con, "babynames", babynames)           # Store data in database

dbListTables(con)                                   # List tables in database
```

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

```
dbListFields(con, "babynames")                      # List variables in babynames table
```

```
## [1] "year" "sex"  "name" "n"    "prop"
```

```
result <- dbSendQuery(con, "SELECT * FROM babynames WHERE year = 2015 ORDER By prop DESC")
                                                    # Send a query to the database
result.data <- dbFetch(result)                      # Fetch the data
dbClearResult(result)                               # Free up resources again
                                                    # Data is now in R in data frame result.data

head(result.data)                                   # Print top 6 observations from data frame
```

```
##   year sex   name     n       prop
## 1 2015   F   Emma 20435 0.01050471
## 2 2015   F Olivia 19669 0.01011095
## 3 2015   M   Noah 19613 0.00962209
## 4 2015   M   Liam 18355 0.00900492
## 5 2015   F Sophia 17402 0.00894559
## 6 2015   F    Ava 16361 0.00841045
```

```
dbDisconnect(con)                                   # Disconnect from the database
```

Installing R packages

Before you can load an R package with `library(packagename)` you need to install it, either using the RStudio user interface (click on the tab *Packages* in the bottom right pane and then click on `Install` or install the package from command line using the command `install.packages("packagename")`.

Hierarchical data

So far we have only looked at reading in data stored in flat tables. However not every data set can be easily stored in flat tables (this is what lead to the ascent no "NoSQL" databases like MongoDB). If you only work with R, you can simply use R's internal `.RData` format. However it is not widely supported by other software.

Alternative file formats that can represent complex data structures are JSON ("JavaScript object notation") and YAML (a recursive acronym for "YAML Ain't Markup Language"). JSON encodes an object as JavaScript code whereas YAML produces a more or less human-readable representation.

Many public data APIs provide data in JSON format (or XML, see below). We'll look at an example reading in such JSON data. We will get the current disruptions on Transport for London's (TfL) tube lines using TfL's public API.

```
library(jsonlite)                                   # Load required package
uri <- "https://api.tfl.gov.uk/line/mode/tube/status"
data <- read_json(uri,  simplifyVector = FALSE) # Download data and convert
for (line in data) {                                # We'll learn about loops later
  cat("Disruptions on",line$name,"\n")
  print(line$disruptions)                           # Most of the time there are none
}
```

```
## Disruptions on Bakerloo
## list()
## Disruptions on Central
## list()
## Disruptions on Circle
## list()
```

```
## Disruptions on District
## list()
## Disruptions on Hammersmith & City
## list()
## Disruptions on Jubilee
## list()
## Disruptions on Metropolitan
## list()
## Disruptions on Northern
## list()
## Disruptions on Piccadilly
## list()
## Disruptions on Victoria
## list()
## Disruptions on Waterloo & City
## list()
```

YAML files can be read and written in R using the yaml package. XML files also provide a way of importing (and exporting) complex data into R. This can be done using the xml2 package.

Answers to tasks

Answer to Task 1.

```
chol.lowdl <- subset(chol, hdl<40)
chol.lowdl
```

```
##   ldl hdl trig age gender     smoke
## 1 175  25  148  39 female        no
## 2 196  36   92  32 female        no
## 4 162  37  139  30 female ex-smoker
```

Answer to Task 2. You can use the following R code.

```
#-- Part (a) -------------------------------------
cia <- subset(cia, !is.na(Population))

#-- Part (b) -------------------------------------
subset(cia, Population<1e4)
```

```
##                               Country             Continent Population
## 50                  Christmas Island                  Asia       1402
## 52            Cocos (Keeling) Islands                  Asia        596
## 78  Falkland Islands (Islas Malvinas) Central/South America       3140
## 103           Holy See (Vatican City)                Europe        826
## 156                        Montserrat Central/South America       5097
## 170                              Niue                 Other       1398
## 171                  Norfolk Island                 Other       2141
## 183                 Pitcairn Islands                 Other         48
## 191                 Saint Barthelemy Central/South America       7448
## 192                     Saint Helena                Africa       7637
## 196         Saint Pierre and Miquelon         North America       7051
## 218                          Svalbard                 Other       2116
## 229                           Tokelau                 Other       1416
##      Life       GDP MilitaryExpenditure
## 50     NA        NA                  NA
## 52     NA        NA                  NA
## 78     NA 105100000                  NA
## 103    NA        NA                  NA
## 156 72.76        NA                  NA
## 170    NA  10010000                  NA
## 171    NA        NA                  NA
## 183    NA        NA                  NA
## 191    NA        NA                  NA
## 192 78.44        NA                  NA
## 196 79.07        NA                  NA
## 218    NA        NA                  NA
## 229    NA        NA                  NA
```

```
#-- Part (c) -------------------------------------
subset(cia, MilitaryExpenditure>0.08*GDP)$Country
```

```
## [1] Iraq         Jordan       Oman         Qatar        Saudi Arabia
## 255 Levels: Afghanistan Akrotiri Albania Algeria ... Zimbabwe
```

```
#-- Part (d) -------------------------------------
cia.europe <- subset(cia, Continent=="Europe" & !is.na(GDP))
sum(cia.europe$GDP)
```

```
## [1] 4.111738e+13
```

```
#-- Alternative answer to part (d) ---------------
sum(subset(cia, Continent=="Europe")$GDP, na.rm=TRUE)
```

```
## [1] 4.111738e+13
```

```
#-- Part (e) -------------------------------------
cia <- transform(cia, GDPPerCapita=GDP/Population,
                      MilitaryExpPerCapita=MilitaryExpenditure/Population)

#-- Part (f)-------------------------------------
cia[order(cia$Life, decreasing=TRUE)[1],]
```

```
##     Country Continent Population  Life       GDP MilitaryExpenditure
## 138   Macau      Asia     559846 84.36 2.204e+10                  NA
##     GDPPerCapita MilitaryExpPerCapita
## 138     39367.97                   NA
```

```
#-- Alternative answer to part (f) ---------------
cia[which.max(cia$Life),]
```

```
##     Country Continent Population  Life       GDP MilitaryExpenditure
## 138   Macau      Asia     559846 84.36 2.204e+10                  NA
##     GDPPerCapita MilitaryExpPerCapita
## 138     39367.97                   NA
```

```
#-- Part (g) -------------------------------------
cia[order(cia$Life, decreasing=TRUE)[1:10],]
```

```
##        Country     Continent Population  Life       GDP
## 138      Macau          Asia     559846 84.36 2.204e+10
## 6      Andorra        Europe      83888 82.51        NA
## 118      Japan          Asia  127078679 82.12 4.844e+12
## 206  Singapore          Asia    4657542 81.98 1.545e+11
## 199 San Marino        Europe      30324 81.97 8.500e+08
## 105  Hong Kong          Asia    7055071 81.86 2.238e+11
## 16   Australia         Other   21262641 81.63 1.069e+12
## 43      Canada North America   33487208 81.23 1.564e+12
## 82      France        Europe   64057792 80.98 2.978e+12
## 220     Sweden        Europe    9059651 80.86 5.129e+11
##     MilitaryExpenditure GDPPerCapita MilitaryExpPerCapita
## 138                  NA     39367.97                   NA
## 6                    NA           NA                   NA
## 118          3.8752e+10     38118.12             304.9449
## 206          7.5705e+09     33172.00            1625.4282
## 199                  NA     28030.60                   NA
## 105                  NA     31721.86                   NA
## 16           2.5656e+10     50275.97            1206.6234
## 43           1.7204e+10     46704.40             513.7484
## 82           7.7428e+10     46489.27            1208.7210
## 220          7.6935e+09     56613.66             849.2049
```

Answer to Task 3. We can use the following R code:

```
weights.all <- merge(patients, weights, by="PatientID")
head(weights.all)
```

```
##   PatientID Gender Age Smoke Week Weight
## 1         1   male  33    no    1     72
## 2         1   male  33    no    2     74
## 3         1   male  33    no    3     71
## 4         2 female  32    no    1     54
## 5         2 female  32    no    3     54
## 6         3   male  67    ex    1     96
```

Answer to Task 4. The first line of the file `cars.csv` contains the variable names and the fields are separated by commas. Missing values are encoded as asterisks.

```
cars <- read.csv("cars.csv", na.strings="*")
str(cars)
```

```
## 'data.frame':    20 obs. of  5 variables:
##  $ Manufacturer: chr  "Chevrolet" "Oldsmobile" "Dodge" "Chevrolet" ...
##  $ Model       : chr  "Camaro" "Achieva" "Spirit" "Astro" ...
##  $ MPG         : int  19 NA 22 NA 25 18 18 25 16 29 ...
##  $ Displacement: num  3.4 2.3 2.5 4.3 2.2 2.8 3 1.8 4.9 1.5 ...
##  $ Horsepower  : int  160 155 100 165 110 178 300 81 200 81 ...
```

We could have also used the function `read.table`.

```
cars <- read.table("cars.csv", header=TRUE, sep=",", na.strings="*")
str(cars)
```

```
## 'data.frame':    20 obs. of  5 variables:
##  $ Manufacturer: chr  "Chevrolet" "Oldsmobile" "Dodge" "Chevrolet" ...
##  $ Model       : chr  "Camaro" "Achieva" "Spirit" "Astro" ...
##  $ MPG         : int  19 NA 22 NA 25 18 18 25 16 29 ...
##  $ Displacement: num  3.4 2.3 2.5 4.3 2.2 2.8 3 1.8 4.9 1.5 ...
##  $ Horsepower  : int  160 155 100 165 110 178 300 81 200 81 ...
```

The first line of the file `ships.txt` contains the variable names and the fields are separated by whitespace. Missing values are encoded as ".".

```
ships <- read.table("ships.txt", header=TRUE, na.strings=".")
str(ships)
```

```
## 'data.frame':    40 obs. of  5 variables:
##  $ type     : chr  "A" "A" "A" "A" ...
##  $ year     : int  60 60 65 65 70 70 75 75 60 60 ...
##  $ period   : int  60 75 60 75 60 75 60 75 60 75 ...
##  $ service  : int  127 63 NA 1095 1512 3353 0 2244 44882 17176 ...
##  $ incidents: int  0 0 3 4 6 18 0 11 39 29 ...
```