# Part IV. Combining multiple datasets

In the previous chapters, we have worked on writing programs that read information from a datafile and calculate something or plot something. For many analyses, however, you need to link multiple datasets to form a new one that is required for your analysis. In the R workflow diagram (Fig. 5.0), these fall under “3b database operation”. In this chapter, we start with some general reflections on databases from the perspective of constructing your own database and working with this. For many applications, you work with an existing database or you essentially construct your own by acquiring two or more datasets that can be linked in a meaningful way. Thus, we focus here on combining data from different sources. The underlying drive of this part (and in general) is: Keep your datafiles in the original shape and use R to carry out all the work. If there are multiple datasets (a database), you keep the original files as they are and combine them in R and process the joined dataset further. In doing so, you have a clear record of what was done to get from your various datafiles to the actual information you used for what is typically the next step: Analyses (operation 4 in the R workflow, Fig 5.0). Because you have a script detailing what you did, you can go back, adjust or repeat it whenever needed as well as have a record of what was done for yourself and for others.

## Chapter 11. Database operations

### 11.1 Relational database: what and why

Connecting datafiles is a general issue which you encounter in most practical applications of programming. For example, when you carry out statistical analyses on data you collected you typically have to first put together the information you need for the analysis, often from multiple source files. Another application is when you wish to connect information in your data files to publicly available datasets, which is common in bio-informatics (e.g. matching genetic sequence data to genes), spatial analyses (e.g. map information to other spatial explicit information) and ecological analyses (e.g. climate data to population counts).

For most practical applications, we need to have more than one file. It is typically not possible, or extremely messy, to have all information in one file. This is because the structure of our data is typically fairly complex and different measures are made on different levels. As an example, consider the following: Suppose we have surveyed 30 meadows each for three times during the flowering season and counted how many of different species of flowers there were in 30 1m2 randomly placed squares on these meadows. We now have data on different levels, which are hierarchically nested.

**1. Meadow**

This includes characteristics of each meadow which are ‘fixed’. That is, they do not vary across different surveys. The meadow’s location, its orientation, its slope, its size, etc.

**2. Visit to each meadow**

Each visit has different characteristics, such as a different date, the weather conditions were probably different and perhaps also who did the survey on that day, etc.

**3. Findings within the squares placed out during each visit to each meadow.**

This is the lowest level of the data, and consists of counts of different species, made for each of the 30 squares.

To place all information in one file requires always that each data line concerns at least the lowest level observed in our data. Thus, for the above example, we could construct a single file, for example in Excel, with for each meadow, 3 times 30 rows of data on each square. This would then repeat the same information for the meadow characteristics (which are ’fixed’) for each of these rows. It would also have to have all the species one could possibly encounter in the columns. In short, this would be a very large datafile, with many rows and columns and lots of repeats (for the levels meadows and visits) and probably lots of missing values (not all species observed in each square). Because the basic data is still fairly restricted in size, it would work reasonably, but it would be inconvenient to work with. In entering the data, you would be prone to make a mistake (e.g. typing in data in the wrong “species” column) and finding such mistakes would be difficult. Furthermore, when you want to plot or analyze the data, you may fairly quickly start to construct what we can consider as ‘sub-files’. This is where things get even more messy and it is possible to rapidly lose track of what and how you handled the data to arrive at a specific answer. This will not do, your findings must be repeatable and transparent!

One alternative is to design a database structure where the information is entered for each level, but where the files can be linked. This is called a relational database. In practice, this requires adding some identification code (ID), which links the levels, whenever relevant. For example, the files could be structured as displayed in Figure 10.1.



*Figure 10.1. Potential construction of a relational data base consisting of three files, where information on each level is denoted only once. The identifiers which link the different files are graphically displayed by arrows.*

The objective of any relational database is that each piece of information is entered only once. Specific files which you may need to address a specific problem or analysis are constructed from the database. The database files themselves are not altered, unless a mistake appears. Any mistake is corrected in the relevant database file (e.g. variable “Size” in “Meadow data”, Fig. 10.1), and the data you need for the analysis are then created again by database operations. Again, if the database is designed properly, the correction is only to be made in one file. These procedures may sound cumbersome, but the advantages of working with a database are:

(1) Entering the information is easier, because redundancy is minimal. The only information which is repeated when you go one level lower is the ID number that links to the higher level. For example, in the “Meadow data” file, each meadow has one row with one “Meadow ID” number. In the “Visit data” file, the “Meadow ID” number needs to be repeated for each of the three rows with different “Visit ID” numbers.

(2) You avoid having several copies of your information in different files which are constructed to allow doing analyses on different levels. The latter is familiar to anyone who has analyzed data where the data manipulation are carried out manually to arrive at specific files for specific analyses. It is amazing how fast you accumulate files!

(3) Because each essential piece of information is entered only once, any mistake found requires only one correction to be made and this correction is done in the relevant database file, not in several files. This makes it much easier to keep track of your data and to be sure it is as correct as possible at any given time.

(4) Because the database is not used for analysis, it does not have to be constructed in a way to allow analyses and is thus much more flexible. For example, the “Census data” file in Fig. 10.1. is fundamentally different from one which you would otherwise need to construct for a statistical analysis. In particular, it could be constructed such that squares in which none of the focal species were found are simply not listed, which may drastically lower the amount of work you need for entering the data, and thus reduces the risk of making a mistake.

Obviously, you can use a database program (e.g. Microsoft Access) to do the database work. These programs are nice and clickable. But, a database program does not analyze anything for you, so you will always need to export it and import it in another program. In fact, my experience is that database programs do not even work particularly well in extracting the file you need for your analysis. One reason is that database programs are not designed well for handling missing values in a flexible manner. Of course, this may be a matter of skill or experience, but it is noteworthy that more demanding database problems typically require you to program it (in SQL or similar). From that perspective, one clear advantage of using R is that you do not need to switch programs and do a lot of clicking, exporting and importing after having made the initial investment of programming. You also do not need to switch programming language. If you find a mistake in the data, you correct the original database. If you find a mistake in what your coding did, you modify the script. In both cases, two key stroke combinations (CTRL+A, CTRL+ENTER) in the R-script editor will run all of it again and redo all of your analyses and plotting. It is important to realize that re-doing analyses because you spotted something odd is really common! Time you lose by sweating on constructing the database and in terms of writing your script to format the purpose-specific file for your analysis, you will gain back, with rent. The more you code, the better you get and the faster you will be able to code the next data extraction challenge. This is perhaps a main reason to stick to R and not confuse yourself with learning many different programs at least in the initial stage.

### 11.2 Using merge() to connect multiple data.frame objects

The function merge() merges data. Selection from the help on merge shows the most important arguments

**Usage**

merge(x, y, ...)

## Default S3 method:

merge(x, y, ...)

## S3 method for class 'data.frame'

merge(x, y, by = intersect(names(x), names(y)),

by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,

sort = TRUE, suffixes = c(".x",".y"),

incomparables = NULL, ...)

**Arguments**

|  |  |
| --- | --- |
| x, y | data frames, or objects to be coerced to one. |
| by, by.x, by.y | specifications of the columns used for merging. See ‘Details’. |
| all | logical; all = L is shorthand for all.x = L and all.y = L, where L is either [TRUE](http://127.0.0.1:13623/library/base/help/TRUE) or FALSE. |
| all.x | logical; if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. The default is FALSE, so that only rows with data from both x and y are included in the output. |
| all.y | logical; analogous to all.x. |

Thus, by default merge() will try to merge two datasets by finding the tag the data.frame x and y have in common. This may work smoothly. For example,

> d1<-data.frame(students=c(“Jukka”, “Harald”, “Sven”, “Bitte”), nationality=c(“F”,”N”,”S”,”D”))

> d2<-data.frame(ages=c(22,22,41,25),students=c(“Harald”, “Jukka”, “Sven”, “Bitte”))

> #merge the data.frames

> merge(d1,d2)

students nationality ages

1 Bitte D 25

2 Harald N 22

3 Jukka F 22

4 Sven S 41

In the above, the tag “students” was there in both data.frame df1 and df2 and R merged them correctly.

In some cases, the naming is perhaps a bit different requiring specification of how merge() should do the trick. For example,

> #merge but different name

> d3<-data.frame(ages=c(22,22,41,25),studs=c(“Harald”, “Jukka”, “Sven”, “Bitte”))

> merge(d1,d3, by.x=”students”,by.y=”studs”)

students nationality ages

1 Bitte D 25

2 Harald N 22

3 Jukka F 22

4 Sven S 41

Here, the first argument (df1) is interpreted as x and the second one (df2) as y (as specified in the help above). Thus, by.x specifies the tag to use in df1 and by.y the tag to use in df2.

It is important to note that merge() will assume the ID really is unique. For example, if we have another student called “Harald”

> d2.b<-rbind(d2,data.frame(ages=21, students=”Harald”))

merge() will not understand the difference between these two Haralds

> merge(d1,d2.b)

students nationality ages

1 Bitte D 25

2 Harald N 22

3 Harald N 21

4 Jukka F 22

5 Sven S 41

Here, both Haralds are automatically assumed to have the nationality “N” although the nationality is specified only for one Harald and we in fact do not know the nationality of the second Harald.

The converse happens rather easily when merging. That is, when information is provided not for all the IDs, merge will by default simply reduce the data. Thus, missing information is by default omitted. In most practical applications, this is a very common situation due to missing values in the data. For example, perhaps we did not have the information for age for student Bitte

> d3.b<-d3[1:3,] #no age for Bitte

> d3.b

ages studs

1 22 Harald

2 22 Jukka

3 41 Sven

> #standard merge (Bitte falls off)

> merge(d1,d3.b, by.x=”students”, by.y=”studs”)

students nationality ages

1 Harald N 22

2 Jukka F 22

3 Sven S 41

Thus, student “Bitte” has disappeared after merging, and hence the information we had on her nationality. We need to specify all.x=TRUE as argument to merge() and override the default (see the help above). Doing so makes sure we have all the information in the first data.frame (argument x) also after the merge operation. Thus,

> merge(d1,d3.b, by.x=”students”, by.y=”studs”, all.x=TRUE)

students nationality ages

1 Bitte D NA

2 Harald N 22

3 Jukka F 22

4 Sven S 41

In this case, the merged data.frame contains NA (missing value: see below) for the student for which we did not have the age in d3.b. Similar procedure works for the second argument in merge() (by specifiying all.y=TRUE) such that it is straightforward to merge two data files which both contain “loose” cases that are unique to either data file. As a rule, it is important to check that cases in your data are not lost after merging. An easy way to check this is either using dim() to obtain the dimension of the data.frame (i.e. number of rows and columns) or nrow() to obtain the number of rows. Note that length() when applied to a data.frame will provide you the number of columns.

### 11.3 Exercises part IV

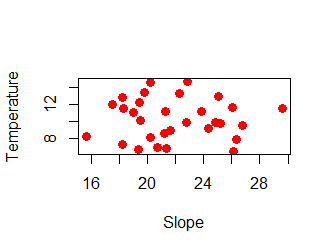
**Exercise IV.1**

Database files which correspond to the structure given in Fig. 10.1 can be found in Moodle. It consists of three “;” separated files called “MeadowsData.txt”, “VisitsData.txt” and “CensusData.txt”.

This data is made up. As detailed above, the data consists of 30 meadows, visited each 3 times. During each visit 30 squares were placed randomly and a count was made of how many of each of 35 species were present in the squares. The information on the different levels (meadow, visit, census) are linked through ID numbers).

Import these three text files in your workspace.

Use the database operations to first link those files you need and then extract the relevant information allowing you to produce a scatter plot of the slope of a meadow (on X) vs. the average temperature on that meadow (on Y) recorded during the three visits. Make the plot look as good (publication-ready) as you can in terms of axes labels, font sizes, plotting symbol, etc. This exercise requires to link information from two database files. Plot should look like:



**Exercise IV.2**

Import the meadows database as described above.

In the meadow database, the lowest level datafile “CensusData.txt” deals with the count of the species found in the squares. There were always 30 squares randomly placed during each visit. If no species were found, the square was not included in the data. That is, zero counts are not included.

a) Produce a histogram using hist() of how many squares were not empty (i.e. had one or more of the focal species in it) per meadow and per visit. That is, you should work out how you count how many squares there are in the census data file per visit (per meadow). Thus, get the columns

Meadow Visit Non-empty.squares.per.meadow.n.visit

1 101 21

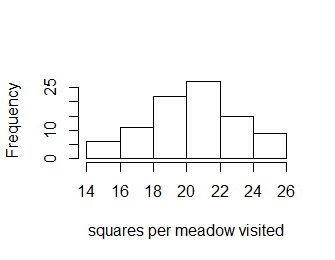
1 102 14

1 103 18

2 201 20

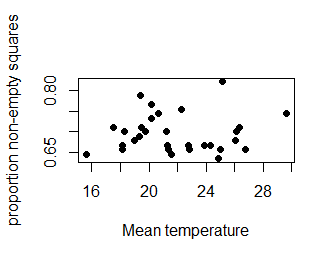
---etc

This is, anyway, what I get ….



b) Add a column where you calculate the number of *empty squares* per visit per meadow, and a column where you calculate the *proportion* of all squares per visit per meadow, which were not empty.

Produce a scatter plot of the average temperature during the 3 visits in a meadow (on X) against the average proportion of non-empty squares during these three visits (on Y). Make the plot look as good (publication-ready) as you can in terms of axes labels, font sizes, plotting symbol, etc. Something like:



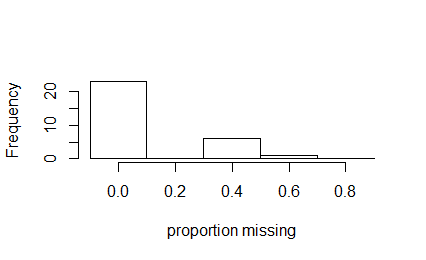
**Exercise IV.3**

In a second survey of the meadows, the thermometer used to measure the temperature was not always working. Thus, some visits had missing data for temperature. This data is in the file “VisitsDataWithNA.txt” which in this exercise replaces the “VisitsData.txt” file in your database.

1. Calculate the average temperature per open and closed meadow and the sample size used to calculate this average
2. One approach to deal with missing data in certain analyses is to assume the missing value is equal to the overall mean. Add a new column to the data where the missing value for temperature has been replaced with the overall mean temperature. Calculate the average temperature per open and closed meadow again and compare to previous estimates.
3. You are worried that one of the observers had real bad luck (or was really sloppy) with the measurement of the temperature. You want to check this. Calculate how many times each observer made a visit, how many times each observer had a missing value for the temperature and

|  |  |  |  |
| --- | --- | --- | --- |
| Who | Nvisits | Nmissed | Propmissed |
| Bob | 20 | 0 | 0 |
| Harriet | 70 | 8 | 0.114285714285714 |

1. Calculate per meadow how many times the temperature was measured and how many times it was missing, as well as the proportion of visits it was missing and produce a histogram of the proportion missing which looks like:



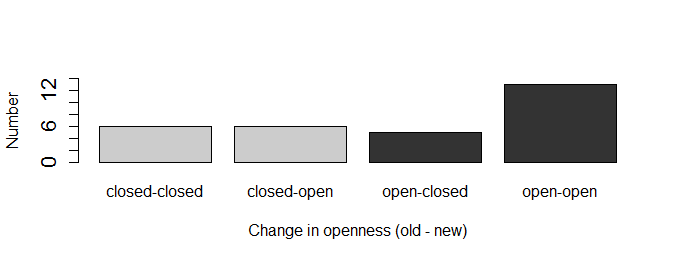
**Exercise IV.4**

Somebody heard about your meadow project and came up with the following interesting information describing for each meadow whether it was “open” or “closed” 20 years ago. The piece of paper she gave your reads:

|  |  |
| --- | --- |
| Mdw | Closed? |
| 1 | yes |
| 2 | no |
| 3 | yes |
| 4 | yes |
| 5 | no |
| 6 | no |
| 7 | no |
| 8 | no |
| 9 | no |
| 10 | yes |
| 11 | no |
| 12 | no |
| 13 | no |
| 14 | yes |
| 15 | no |
| 16 | no |
| 17 | no |
| 18 | no |
| 19 | yes |
| 20 | no |
| 21 | no |
| 22 | yes |
| 23 | no |
| 24 | no |
| 25 | no |
| 26 | yes |
| 27 | Yes |
| 29 | Yes |
| 28 | No |
| 30 | Yes |

Add this data to your database by copy-pasting the above to a text file and importing this text file into a separate data.frame object called newMeadowData

Produce a single bar plot depicting the number of meadows which were open or closed 20 years ago and open or closed now. That is, the numbers of open-open; open-closed; closed-open; closed-closed. Something like below (the order can vary depending on how you coded it):



**Exercise IV.5**

The datafile “AllDataIndividuals.txt” contains data on the body mass of all individuals. It contains the variables 1. individual (ID code for each individual), 2. brood (a unique code for each brood the individual was born in), 3. body mass (of the individual), 4. site (a code for each site).

The datafile “MotherData.txt” contains data on which individual was the mother of all the individuals that were raised in the same brood and contains 1. mother (ID code of the individual that was the mother of a brood), 2. brood (code of the brood the individual was the mother for)

Plot the average body mass of individuals in a brood (on Y) against the body mass of their mother (on X). The data plotted should be like the plot below:

