

Manipulating big data in R for vegetation scientists

Viktorija Wagner

Version 1.3 - 23 November 2015

This script was first written as a companion to the course 'Manipulating big data in R for vegetation scientists', taught at the 58th Annual Symposium of the International Association for Vegetation Science, in Brno, Czech Republic, 18-19 July 2015. Feel free to share this document but please acknowledge this work by inserting the citation:

Wagner, Viktoria (2015): *Manipulating big data in R for vegetation scientists* Version 1.2. Unpublished course script. URL: <http://viktoriawagner.weebly.com/>. Accessed on *date*.

The preparation of this document was financed from the SoMoPro II programme. The research leading to this result has acquired a financial grant from the People Programme (Marie Curie Action) of the Seventh Framework Programme of EU according to the REA Grant Agreement No. 291782. The research is further co-financed by the South-Moravian Region. The content of this paper reflects only the author's views; the European Union is not liable to any use that may be made of the information contained therein.

Salza Palpurina kindly provided comments on an earlier draft of the manuscript.



Viktoria Wagner

Department of Botany and Zoology

Masaryk University

Kotlářská 2, CZ-611 37 Brno (Mailing address)

Kamenice 5 , CZ-625 00 Brno-Bohunice (Visiting address)

Czech Republic Phone: + 420 532 146 299, Fax: + 420 532 146 213

wagner@sci.muni.cz, <http://viktoriawagner.weebly.com/>

This file was created with LaTeX in TexShop. This script is continuously updated. If you spot any errors, please contact the author.

Table of contents

1	Introduction	5
1.1	Notes on how to use this manual	5
1.2	Background	5
1.3	Packages	6
1.4	Installation of packages	7
1.5	Loading and detaching packages	7
2	General tools	9
2.1	Time your R-processes	9
2.2	Manage your R-memory	10
2.3	Pipe functions	11
2.4	Aliases	13
3	Read data into R's workspace	14
3.1	Overview	14
3.2	Read delimited files	15
3.3	Encoding	16
4	Write data to file	18
5	Data inspection	20
6	String operations	22
6.1	Paste, split and check encoding	22
6.2	Regular expressions	23
7	Reshaping tables: Wide \leftrightarrow Long	26
7.1	Wide format	26
7.2	Long format	27
7.3	Wide \rightarrow Long: gather()	28
7.4	Long \rightarrow Wide: spread()	28
8	Adding or deleting rows and columns	30
8.1	Adding columns: mutate()	30
8.2	Adding rows: bind_rows()	31
8.3	Deleting columns	32
8.4	Deleting rows	32
9	Manipulate columns and cell entries	33
9.1	Rename columns: mutate(), rename()	33
9.2	Replace values: mutate(), mapvalues(), replace()	34
9.3	Update values by group	35
10	Rearranging order of rows and columns	37
11	Select rows	39

12 Select columns	41
13 Combining datasets	42
13.1 Joining	42
14 Data cleaning	47
14.1 Spot duplicate entries	47
14.2 Spell-checking taxon names	48
14.2.1 General spelling mistakes	48
14.2.2 Spell-checking with reference lists	49
15 Merging cover values	54
16 Data summary per group	57
16.1 Number of species per plot	57

1 Introduction

1.1 Notes on how to use this manual

This manual was written as a companion to the course "Manipulating big data in R for vegetation scientists", held at the 58th Annual Symposium of the International Association for Vegetation Science, in Brno, during 18-19 August 2015. The aim of the course was to familiarize participants with R tools for data manipulation, including reading, reformatting, merging, summarizing, proofreading, and exporting data.

Although R can access data remotely, the course and the manual focus on R's ability to import and handle data in its workspace. Furthermore, although **data.table** is as suitable for the manipulation of large tables as **dplyr**, the manual focuses largely on the **dplyr** package. This is due to **dplyr**'s more intuitive syntax and its suitability for piping. Both points are an advantage for R beginners.

The R code in most sections can be easily reproduced by copy-and-paste. In general, all necessary objects are created adhoc at the beginning of the section and necessary packages are loaded and attached. The R code in the manual is written to increase readability. However, some invisible formatting symbols (spaces, line breaks) could interfere with code execution when the code is pasted into R. To ensure that the code works, copy it from this file and paste it into the RStudio editor tab (or another editor). Do not paste it into the console. Then, submit the code to the console, row by row. Do not try to submit an entire section as it might throw an error.

1.2 Background

Vegetation data store information on species composition, species abundance and related variables for a given area ('plot'). They provide indispensable insights for basic ecology, biogeography, land management, conservation and ecological restoration. In the last decades, the amount of electronically stored vegetation data has steadily increased. For example, Schaminée et al. (2009) have estimated that data for 1.8 million plots are stored electronically in Europe. Consolidation of databases has led to large depositories, e.g. Vegbank: data on 76,000 plots, s-Plot: 1.17 million plots, and the European Vegetation Archive: 1 million plots (Chytrý et al *in press*). The combination of electronically stored vegetation data and biological, environmental and geospatial databases offers exciting opportunities for inference to both science and application. However, large vegetation data pose also novel challenges for data processing and preparation. In particular, memory can become a limiting factor when processing large vegetation databases. Data preparation steps must be carried out fast, with minimum usage of RAM. Furthermore, large databases can contain spelling errors and typos which are difficult to detect manually.

R has become one of the most popular software tools for the analysis and visualization of vegetation data. However, few vegetation scientists know that R is also a versatile tool for processing large data. In the last years, several packages have been released that allow users to manipulate data fast and efficiently. In particular, the **dplyr** and **data.table** packages have advanced R's ability to manipulate large datasets. By using the function in these packages, vegetation scientists can perform some classic manipulations, including spotting duplicate entries and joining vegetation data with biological and environmental data.

References

Chytrý, M., Hennekens, S.M., Jiménez-Alfaro, B. *et al.* (in press) European Vegetation Archive (EVA): an integrated database of European vegetation plots. *Applied Vegetation Science*.

Schaminée, J.H. J., Hennekens, S., Chytrý, M. & Rodwell, J.S. 2009. Vegetation-plot data and databases in Europe: an overview. *Preslia* 81: 173-185.

1.3 Packages

Table 1.1: Useful packages for manipulating large vegetation data, their leading author, year of first release, depositories, and description. Numbers following package names indicate versions used.

Package	Author	First release	Deposit	Description
data.table 1.9.5	M. Dowle	2006	CRAN, github	Extension of data.frame
dplyr 0.4.2	H. Wickham	2014	CRAN, github	A Grammar of data manipulation
foreign 0.8-65	R Core Team	1999	CRAN	Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, ...
magrittr 1.5	S. Milton	2014	CRAN, github	A forward-pipe operator for R
plyr 1.8.3	H. Wickham	2008	CRAN, github	Tools for Splitting, Applying and Combining Data
pryr 0.1.2	H. Wickham	2014	CRAN, github	Tools for computing on the language.
readr 0.1.1	H. Wickham	2015	CRAN, github	Read flat/tabular text files from disk
stringr 1.0.0	H. Wickham	2009	CRAN, github	Simple, consistent wrappers for common string operations
stringi 0.5-5	M. Gagolewski	2014	CRAN, github	Character String Processing Facilities
taxize 0.6.2.9632	S. Chamberlain	2012	CRAN, github	Taxonomic Information from Around the Web
taxizesoap 0.1.0.99	S. Chamberlain	2014	github	Extension of taxize for data retrieved with the SOAP data transfer protocol
Taxonstand 1.7	L. Cayuela	2012	github	Taxonomic Standardization of Plant Species Names
tpl 0.1	G. Carvalho	2015(?)	github	R package (including shiny app) to query The Plant List
vegdata 0.6.9	F. Jansen	2009	CRAN	Access Vegetation Databases and Treat Taxonomy
veggie 0.0.1	V. Wagner	2015	github	Manipulation of Vegetation Data

Note that the **data.table** and **taxize** versions presented here are development versions on

github.

1.4 Installation of packages

(1) **CRAN** is the official repository for R packages (<http://cran.r-project.org/>). To download and install a package from the CRAN online repository, run the `install.packages()` function in R, specifying the package name within quotation marks and parentheses. Under Windows and Mac, the package will be automatically downloaded and installed into your R library. Suppose we want to install the **dplyr** package:

```
install.packages("dplyr")
```

Under Linux, you must specify the directory to the library folder, where your packages should be installed, in the 'lib'-argument. In the example below, I am using the directory on my notebook (admittedly a Mac):

```
lib.vec <- "/Library/Frameworks/R.framework/Versions/3.2/Resources/library"  
install.packages("dplyr", lib = lib.vec)
```

You can locate the library folder on your machine with the function `.libPaths()`.

(2) **github** is the second important online repository for R packages (<https://github.com/>). It has several advantages for package development, like version control, collaborative mode, and a bug-reporting system. Many packages which are found on CRAN have a development version on github; but some are exclusively deposited on CRAN or github. To install a package from github, you need first to install (once) and load the **devtools** package:

```
install.packages("devtools")  
library(devtools)
```

Next, install your desired package from github by specifying the developer's username and package name, for example the **taxize** package from ROpenScience:

```
install_github("ropensci/taxize")
```

1.5 Loading and detaching packages

If you want to work with a certain package that is not part of R base, you need to load and attach it into your session with the `library()` function. Below is an example for loading and attaching the **dplyr** package:

```
library("dplyr")
```

Again, in Linux, you have to specify the library folder where the package is installed (I am using again my directory as an example).

```
lib.loc.vec <- "/Library/Frameworks/R.framework/Versions/3.2/Resources/library"  
library(dplyr, lib.loc = lib.loc.vec)
```

Packages are automatically detached from your workspace once you shut R or RStudio down. However, if you want to remove a package from your current workspace without closing the application, use the `detach()` function. Below is an example for how to detach the `dplyr` package.

```
library(dplyr)
detach(package:dplyr)
```

Packages can be uninstalled with the `remove.packages()` command. You can specify the library location in the 'lib' argument. If that is missing, R will use the directory shown under `.libPaths()`.

```
remove.packages("data.table")
```


2 General tools

2.1 Time your R-processes

When processing large datasets, the time a function takes to execute a task can become a limiting factor. Below are three options how to time R functions, based on an example using the arbitrary function 'some.output <- rnorm(10^6)':

```
# OPTION 1:
ptm <- proc.time()
some.output <- rnorm(10^6)
diff1 <- proc.time() - ptm

diff1      # units in seconds
  user  system elapsed
 0.105   0.001   0.106

# OPTION 2:
system.time(
  some.output <- rnorm(10^6)
)

# units in seconds
  user  system elapsed
 0.082   0.001   0.082

# OPTION 3:
t1 <- Sys.time()
some.output <- rnorm(10^6)
t2 <- Sys.time()

difftime(t2,t1)
Time difference of 0.1059768 secs
```

Depending on the computer you are using, the calculated times might look different. Options 1 and 2 produce three numbers ('user', 'system', and 'elapsed'). To understand the first two numbers, you need to know that R functions can be split up into processes that R is executing and that the OS is executing on behalf of R (e.g. allocating additional memory for processes, reading and writing files). However, these two numbers are usually of little interest. Instead, it is the last number which is most useful: It gives the total amount of elapsed time (in seconds).

All three options are equally well suited to time R functions. Option 3 has a slight advantage in that you can set the unit to be reported ('secs', 'mins', 'hours', 'days', 'weeks'). This is particularly valuable if your R code runs for a long time and you don't want to convert seconds to higher units.

2.2 Manage your R-memory

Once objects are imported or created in R, they use up memory (RAM) for storage of data, meta-data, and attributes. Memory usage can be easily monitored with the **pryr** package.

Memory usage by single objects can be inspected with **object_size()**:

```
library(pryr)

some.output <- rnorm(10^6)

object_size(some.output)
8 MB
```

Memory usage by all objects is returned by **mem_used()** :

```
mem_used()

34.6 MB
```

Objects can share memory, if they are pointing to each other. For example, imagine we want to create a new data frame *df* based on the vector above:

```
df <- data.frame(x = some.output)

object_size(df)
8 MB

object_size(df, some.output)
8 MB

mem_used()
34.8 MB
```

As you can see, both objects take the same amount of memory as each individual object because *some.output* points to *df*.

Sometimes, we want to get rid of a large and unnecessary R-object to free up some space. However, R will only free up memory for an object, if no other objects are pointing to it. Using the example above, removing *some.output* with the function **rm()** will not free up space because *df* is still pointing to it:

```
rm(some.output)

mem_used()
34.8 M
```

It is only after we also have removed *df* that the 8MB memory is released.

```
rm(df)

mem_used()
26.8 MB
```

In general, R is efficient with memory usage. If it needs more memory, it will check for deleted objects and release memory through a process called **garbage collection**. But sometimes we are impatient and want to release memory immediately. This can be enforced with the **gc()** function (again: memory will be released only for those deleted objects to which no other objects

are pointing to).

Another important aspect for memory management is detecting **hidden copy-making**. Some R-functions will duplicate an object without giving any notice. If you run them on large R-objects, their hidden copy-making will use up precious memory. To make hidden copy-making visible, use the `tracemem()` function in the **base** package.

For example, if you have an object `df` and you call `'tracemem(df)'`, your `df` object gets a number. From now on, hidden copy-making in a function is exposed and the respective number for the object is traced. Below is an example, in which we substitute a value within a data frame `df` (cell in row 1 and column 2) by `'NA'`. As we have wrapped the `tracemem()` function around our `df` object before, we can see that the replacement function copies `df` silently in the background.

```
df <- data.frame(x = rnorm(10), y = rnorm(10))
tracemem(df)
[1] "<0x7f95fb6c2e60>"

df[1,2] <- NA
tracemem[0x7f95fb6c2e60 -> 0x7f95fb6a06b0]:
tracemem[0x7f95fb6a06b0 -> 0x7f95fb6a0838]: [<-.data.frame [<-
tracemem[0x7f95fb6a0838 -> 0x7f95fb6a0a30]: [<-.data.frame [<-
```

The specific example above is trivial and will take only a fraction of a second. However, with complex functions and large data frames, copy making use up valuable time.

A strength of the packages **dplyr** and **data.table** is that most of their functions avoid copy-making.

2.3 Pipe functions

Piping is a relatively recent addition to R, introduced in the **magrittr** package. It refers to a syntax that chains individual functions with a pipe symbol (e.g. `%>%`, `%<>%`). By using pipes, you can build more comprehensive and shorter R syntax.

Piping is an excellent companion to **dplyr**. Here an example with the *iris* dataset, in which sepal length (column `'Sepal.length'`) is listed for three *Iris* species (`'Species'`).

First, let's look at a set of data manipulation functions that we will carry out in the classic, *NON-pipe way*:

We will construct a data frame in which we calculate the mean sepal length for each species. Next, we will sort the resulting table in descending order:

```
temp.df <- group_by(iris, Species) # group species
temp.df.1 <- summarize(temp.df,
  mean.length = mean(Sepal.Length)) # calculate mean
temp.df.2 <- arrange(temp.df.1, desc(mean.length)) # sort table
temp.df.2

Source: local data frame [3 x 2]

  Species mean.length
1  virginica      6.588
```

```

2 versicolor      5.936
3      setosa      5.006

```

Note how cumbersome the syntax above is: We've created three new objects, are reusing the data frame name of the previous call and most of all, our syntax is difficult to read. With piping, we can chain all steps into one step, without creating new redundant objects in each line and reusing the previous data frame name.

Here is how we proceed in the *PIPE-way*:

We start the chain of functions by supplying the original data frame name, first, and then sequence all necessary functions with the **forward pipe operator** `%>%`:

```

library(dplyr)
library(magrittr)

iris %>%
  group_by(Species) %>%
  summarize(x = mean(Sepal.Length)) %>%
  arrange(desc(x))
# table name
# group by species
# calculate mean
# sort table

Source: local data frame [3 x 2]

  Species mean.length
1  virginica      6.588
2  versicolor      5.936
3    setosa       5.006

```

Note that I have stretched the functions above across several lines but you can place the entire chain into one single line. The `%>%` symbol indicates that the previously supplied object should be used as an input for the next function. So, there is no need to specify the data frame name `iris` in the following function. For example, `'iris %>% group_by(Species)'` is the same as `'group_by(iris, Species)'`.

In the example above, we've used the forward pipe (`%>%`) which yields a new table and leaves the original table unmodified. By comparison, the **compound assignment pipe operator** `%<>%` replaces the original data frame with a new object (as created in subsequent steps). This is convenient when we want to rename columns or replace values in a large data frame:

```

colnames(iris)
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

```

Let's rename column 'Sepal.Length' to 'L' by using `%<>%`:

```

iris %<>% rename(L = Sepal.Length)

colnames(iris)
[1] "L" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

```

The forward pipe (`%>%`) is currently implemented in **magrittr** and **dplyr** but the compound assignment pipe is included in **magrittr**, only. Piping can be more versatile than shown here. For instance, it can be used with vectors and lists, and there are additional pipes. Check out the documentation in the **magrittr** package. The recently released **pipeR** packages offers even

a wider range of piping solutions.

2.4 Aliases

The **magrittr** package offers additional tools that can be used in conjunction with pipes. For example, we can use the **use_series()** function to extract a vector from a data frame.

```
# Extract a column as a vector
# Show only first 10 hits
iris %>%
  use_series(Petal.Length) %>%
  head()
[1] 1.4 1.4 1.3 1.5 1.4 1.7

# How many distinct values does the vector have?
iris %>%
  use_series(Petal.Length) %>%
  n_distinct()
[1] 43

# Show maximum value for a vector
iris %>%
  use_series(Petal.Length) %>%
  max()
[1] 6.9
```

To see the entire list of aliases, type for example:

```
?use_series
```

3 Read data into R's workspace

3.1 Overview

Vegetation data can be stored in a wide array of different formats, such as relational databases (e.g. MS Access), markup language (e.g. XML) or flat files (e.g. .txt). R is able to open all common formats and import file content in its workspace (Table 3.1).

Table 3.1: Common file formats for storage of vegetation data, corresponding R packages and functions for data import

Type	Ending	R package	Function	Comment
Delimited	.txt	base	read.table()	
	.txt, .csv	base	read.csv(), read.csv2()	
	.txt	readr	read_delim()	for large files
	.txt, .csv	readr	read_csv(), read_csv2()	for large files
		data.table	fread()	for large files
Excel	.xls, .xlsx	XLConnect	loadWorkbook()	One of several packages that can import Excel files
XML	.xml	XML	xmlParse()	for general xml-import, incl. Veg-X (Wiser et al. 2011)
		vegdata	ESveg.obs(db, ...)	import of ESveg standard (species data)
		vegdata	ESveg.site(db, ...)	header data
dBase	.dbf	foreign	read.dbf()	
Access	.mdb	RODBC	odbcConnectAccess() + sqlFetch()	
SQL	.sql	RODBC	odbcConnectAccess() + sqlFetch()	

R is also able to access databases remotely, without importing them into its workspace. This is most convenient for databases that are too large to fit into R's memory. The course will not deal with this topic but check out the **dplyr** package for remote access to SQL databases.

3.2 Read delimited files

Delimited files are the most common type for data storage because they can be easily shared among different operating systems and accessed with different software. Given they can fit into R's memory, large delimited files can be imported in three different ways, using

- (1) the `read.table()` function in **base** (with custom settings),
- (2) the `read_delim()` function in the **readr** package, and
- (3) the `fread()` function in the **data.table** package.

The first two functions will convert the content into a data frame, whereas the `fread()` function can create a data table (an enhanced data frame, default) or data frame.

The `read.table()` function (and similar functions in the **base** package) is the classic option for importing delimited files but it suffers from a long syntax.

`read_delim()` does not only have a very slim syntax but can also display a progress bar.

`fread()` is the fastest (see code below) and most versatile option. For example, it is able to import only a subset of columns. Furthermore, not only can it display a progress bar, it can also be 'chatty' (report progress status and timings of individual steps).

In the example below, I compared the speed with which the three option import a 2.14GB vegetation data file. The file consists of 5,665,866 rows and 50 columns (file not reproduced here):

```
setwd(/Users/Data/)      # my custom working directory
```

(1) `read.table()`:

```
# do not run

ptm <- proc.time()
data <- read.table("mydata.txt",      # File name
                  nrows = 5665866,   # Number of rows
                  colClasses = "character", # Defines column type
                  sep = "\t",        # Separator of columns (here: tab)
                  header = T,        # header (column names) is present
                  na.strings = NA,   # Abbreviation for missing values
                  stringsAsFactors = F) # Don't convert text strings to factors

diff1 <- proc.time() - ptm
```

(2) `read_delim()` in the **readr** package:

```
# do not run

library(readr)

ptm <- proc.time()
data <- read_delim("data.txt",      # File name
                  # column type:
                  col_types = paste(rep("c",50), collapse = ""),
                  delim="\t",      # type of delimiter
                  progress=T)

diff2 <- proc.time() - ptm
```

(3) `fread()` in the `data.table` package:

```
# do not run

library(data.table)

ptm <- proc.time()
data <- fread("data.txt",
             colClasses = rep("character",50), # File name
             sep = "\t", # column type
             nrows = 5650247, # type of delimiter
             header = T,
             stringsAsFactors = F,
             showProgress = T)
diff3 <- proc.time() - ptm
```

Speed comparison (time in s):

```
data.frame(Option = c("read.table()", "read_delim()", "fread()"),
          time = round(c(diff1[3], diff2[3], diff3[3]),2))

  Option  time
1 read.table() 311.13
2 read_delim() 179.32
3      fread()  61.37
```

3.3 Encoding

Characters in vegetation data can include simple ASCII-characters (e.g. A, p, T, 1) or non-ASCII special characters (e.g. 'ö', 'ñ', 'á' or Chinese characters). The latter is often found in columns that store locality information (e.g. 'Moravský Kraj') or author names in taxon strings (e.g. "Luzula hitchcockii Hämet-Ahti").

To what extent R will import and display non-ASCII characters correctly (i.e. as in the original file), depends on three conditions:

(1) **The encoding of your file.** If your delimited file includes non-ASCII characters and you want them to be displayed correctly in the R console (and the tables and figures you are generating with R), the file should be in UTF-8 format. It is also important that characters in the original file are not corrupted. Corruption could have happened e.g. after somebody opened a file with non-ASCII characters in a software that does automatic character conversion and then saved it. In this case, there is not much you can do except to fix the corrupted characters and save the file under a UTF-8 format in text processing software (e.g. in Text Wrangler or Sublime Text).

(2) **The encoding that your operating system is using.** During installation, R will usually adopt the encoding of your operating system. Check out the encoding system that R is using on your machine with this function (in my case, R is using UTF-8):

```
Sys.setlocale()
[1] "en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8"
```

In general, Linux and Mac operating systems employ UTF-8 encoding. This makes the import and display of non-ASCII characters in R comparatively easy (you might have to set the right

encoding also in the import functions, see point 3). By comparison, encoding on Windows can vary and cause problems when non-ASCII characters are imported. For example, if you have non-Chinese encoding on Windows, you might have trouble displaying Chinese characters in R. Note that the developers of RStudio (versions > 0.93) have announced that their software can display non-ASCII characters correctly (See <https://support.rstudio.com/hc/en-us/articles/200532197-Character-Encoding>).

(3) The encoding function that you are using to import a file. Functions `read.table()` and `fread()` (in `data.table` version > 1.9.5) can read and preserve non-ASCII characters if you specify a UTF-8 encoding in arguments 'encoding' and 'fileEncoding'. However, you **need to know whether your file is encoded in UTF-8 or not**. But how to tell what encoding is used in your file? Assuming that your file includes uncorrupted non-ASCII characters (see point 1), the safest option is to open the file in Text Wrangler, Sublime Text (or another text-processing software) and save it again with the encoding set to 'UTF-8'. Then, you can proceed in R to import the file by setting the encoding to UTF-8:

```
# Option 1: read.table()
data <- read.table("mydata.txt",      # File name
                  nrows = 5665866,
                  colClasses = "character",
                  header = T,
                  na.strings = NA,
                  stringsAsFactors = F,
                  encoding="UTF-8",
                  fileEncoding="UTF-8")

# Option 2: fread()
data <- fread("data.txt",
              colClasses = rep("character",50),
              sep = "\t",
              nrows = 5650247,
              header = T,
              stringsAsFactors = F,
              showProgress = T,
              encoding="UTF-8")
```

Note that when the manual was written, the 'encoding' argument in `data.table` was just released (development version 1.9.5 on github). Some changes might have occurred since then.

4 Write data to file

Writing large amounts of data to files can be time-consuming. Unfortunately, neither **data.table** nor **dplyr** currently offer functions for rapid data export. In the case of delimited files, use the base function **write.table()**. Some users have reported a gain in speed by converting a data frame (made up of many character columns) to a matrix first, and then writing the data to the file batches. Last but not least, you can save your table as an **.RDS object** with the base function **writeRDS()** which uses up less disk space (but can be opened only with R). Let's compare the three options. We will use as an example a file with 5,665,866 rows and 50 columns (file not reproduced here):

(1) **write.table()** function:

```
# do not run

ptm <- proc.time()
write.table(data,
            "data.txt",
            sep = "\t",
            row.names=F,
            quote = F)
diff4 <- proc.time() - ptm
```

(2) **write.table()** function (in batches):

```
# do not run

# First, convert your table to a matrix
# First batch (50% of rows)

nr <- nrow(data)
ptm <- proc.time()
data.m <- as.matrix(data)
write.table(data.m[1:(nr/2),],
            "data.txt",
            row.names = F,
            sep = "\t",
            quote = F)

# second batch
write.table(data.m[(nr/2)+1:nr,],
            "data.txt",
            append = T,          # Append batch to existing file
            row.names = F,
            col.names = F,
            sep = "\t",
            quote = F)
diff5 <- proc.time() - ptm
```

(3) **write.rds()** function

```
# do not run

ptm <- proc.time()
saveRDS(data, "data.rds")
diff6 <- proc.time() - ptm
```

Speed comparison (time in s):

```
data.frame(Option = c("write.table()", "as.matrix() + write.table()", "writeRDS"),  
           time = round(c(diff4[3], diff5[3], diff6[3]),2))
```

	Option	time
1	write.table()	90.82
2	as.matrix() + write.table()	115.68
3	writeRDS	128.46

Clearly, the fastest choice is the classic **write.table()**.

5 Data inspection

Data inspection is straightforward when you are dealing with a small table. Type the name of the data frame and it will be printed in the console.

```
# EXAMPLE
# data frame with 4 columns and 7 rows (+header)

spec.vec <- c("Fagus sylvatica", "Acer platanoides",
             "Melica uniflora", "Hordelymus europaeus",
             "Lonicera periclymenum", "Lilium martagon",
             "Lythyrus vernus")

veg.df <- data.frame(plot =c(rep("P1", 7)),
                    spec = spec.vec,
                    layer=c("tr", "tr", rep("h", 5)),
                    cov=c(60,20,5,5,5,1,5))

veg.df

  plot          spec layer cov
1  P1    Fagus sylvatica   tr  60
2  P1    Acer platanoides   tr  20
3  P1    Melica uniflora    h   5
4  P1 Hordelymus europaeus   h   5
5  P1 Lonicera periclymenum   h   5
6  P1    Lilium martagon    h   1
7  P1    Lythyrus vernus    h   5
```

However, R has a limit for how many rows and columns it will display. In case the number of rows in the table exceeds its default limits, it will truncate the printed table.

Beware of submitting just the name of a very large table: R might crash!

Instead, make a habit of transforming your large data frame to a **tbl-data frame**, as soon as you create it. This object is identical to a data frame but differs in its print output to the console: only its head and tail will be printed. Consider this (comparatively small) data frame *df* consisting of 10 million rows and two columns:

```
df <- data.frame(x = rnorm(10^7), y = rnorm(10^7))

library(dplyr)

df <- tbl_df(df)
df

Source: local data frame [10,000,000 x 2]

   x          y
1  0.1929042 -0.59996971
2 -0.7101549  0.43006413
3 -0.1864493  1.43324735
4  1.1486379 -0.89638429
5 -0.4540759 -2.10334832
6  0.5059702 -0.82252682
```

```

7 -0.3152344 0.44769060
8 -0.3619789 0.48927809
9 -0.4754735 1.37828149
10 -0.4859998 0.02161131
..      ...      ...

```

In case of a tbl-data frame, R will print out only as many rows and columns as can fit into your screen (in this case: 10 rows and 2 columns).

Another option is to transform your data.frame to a data table with the `setDT(as.list())` function in the `data.table` package. These objects are also displayed in a compact way but unlike with a tbl-data frame, all columns are shown:

```

df <- data.frame(x = rnorm(10^7), y = rnorm(10^7))

library(data.table)

df.dt <- setDT(as.list(df))

df.dt
      x      y
1: 0.1597989 0.5663999
2: -1.4442060 0.4893772
3: 0.7479281 -0.5184879
4: -0.6818427 1.4947946
5: -1.6210944 1.2190902
---
9999996: -0.7628983 0.5805407
9999997: 0.7230575 -0.7793770
9999998: 0.9685876 -0.1737581
9999999: 0.9798970 -0.6954244
10000000: 1.4770516 -0.7473812

```

The dimensions of your table can be checked with the following functions:

```

dim(df)      # Dimensions
[1] 10000000 2

nrow(df)     # No of rows
[1] 10000000

ncol(df)     # No of columns
[1] 2

```

6 String operations

6.1 Paste, split and check encoding

String operations are indispensable little helper functions that can make manipulation of large vegetation data easier. Unlike many other functions that are introduced in this manual, they do not manipulate entire tables but vectors. One of the most simplest commands is `paste()`, which joins two or more single strings into a more complex string:

```
paste("Fagus", "sylvatica", sep=" ")
[1] "Fagus sylvatica"
```

In the separator ('sep') argument, we specified that a single space should separate the two strings. The function can also be applied to two or more vectors:

```
genus <- c("Fagus", "Oxalis", "Equisetum")
spec <- c("sylvatica", "acetosella", "hyemale")

taxon <- paste(genus, spec, sep=" ")
taxon
[1] "Fagus sylvatica" "Oxalis acetosella" "Equisetum hyemale"
```

The **stringr** package offers several functions for string operations. One of the most important ones is the `str_split()` function. Below we use it to split the newly created vector by a space ('pattern' argument) and thus, separate taxon names back into a genus name and epithet:

```
library(stringr)

str_split(taxon, pattern=" ")
[[1]]
[1] "Fagus" "sylvatica"

[[2]]
[1] "Oxalis" "acetosella"

[[3]]
[1] "Equisetum" "hyemale"
```

The resulting object is a list. Below, we separate the list elements into two vectors, one for the genus and one for the epithet name:

```
sapply(str_split(taxon, pattern=" "), function(x) x[1])
[1] "Fagus" "Oxalis" "Equisetum"

sapply(str_split(taxon, pattern=" "), function(x) x[2])
[1] "sylvatica" "acetosella" "hyemale"
```

Another useful function in the **stringr** package is `str_trim()` which removes redundant white spaces at the beginning and/or end of a string (specify the trim side with the 'side' argument). This is particularly helpful for proofreading taxonomic (see Chapter 15.2) or regional names.

```
vec <- c("Veronica ", "Pyrola ")
str_trim(vec, side = "both")
[1] "Veronica" "Pyrola"
```

Paste in combination with the `str_sub()` function can be used to reorder elements within a string. For example, we can reorder the order of days, months and years in a string for dates. In this case, `str_sub()` will extract elements at the specified positions (first number: start of position, last number: end of position).

```

dates <- c("07.05.1985", "23.04.1999")

str_sub(dates, 1,2) # days
[1] "07" "23"

str_sub(dates, 4,5) # months
[1] "05" "04"

str_sub(dates, 7,10) # months
[1] "1985" "1999"

# Now reorder strings by using paste()
paste(str_sub(dates, 7,10),str_sub(dates, 4,5),str_sub(dates, 1,2), sep=".")

[1] "1985.05.07" "1999.04.23"

```

The encoding of a string can be checked with the `stri_enc_mark()` function in the `stringi` package:

```

library(stringi)
non <- c("Jihomoravský kraj", "Brno")

stri_enc_mark(non)
[1] "UTF-8" "ASCII"

```

6.2 Regular expressions

Regular expressions ('regex') are a set of characters used to match (and replace) patterns. They are implemented in various languages (e.g. R, Perl, Java, Ruby) and software (e.g. Text Wrangler, Sublime Text). Learning regular expressions can be a steep learning curve but it is worthwhile because they can spare you dull programming and paste + copy operations.

Table 6.1: Overview of common regular expressions. See <http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/> for a more comprehensive overview.

Regular expression	Meaning
<code>\s</code>	White space
<code>\S</code>	Not white space
<code>\w</code>	Text (word) character
<code>\W</code>	Not text (word) character
<code>\d</code>	Digit
<code>\D</code>	Not digit
<code>\n</code>	Line ending
<code>\t</code>	Tab delimiter
<code>\$</code>	End of string or line

Several functions in R base and add-on packages use regular expressions. They usually support two types of expressions: the extended (default) and the more elaborate Perl-style regular expressions. Be aware that in R, regular expressions need an additional backslash at the beginning.

First, let's see how we can use regular expressions in the `grepl()` function to match certain string elements within a vector of taxon names:

```
vec <- c("Arundo donax",
        "Typha spec1",
        "Typha pallida",
        "Achillea millefolium agg.",
        "Lotus corniculatus agg.",
        "Hieracium agg. heldreichii")

# Match names with "Typha"
# No regular expression used, yet)
vec[grepl("Typha", vec)]
[1] "Typha spec1" "Typha pallida"

# Match names that include a digit
vec[grepl("\\d", vec)]
[1] "Typha spec1"

# Match names where first character is an "A"
vec[grepl("^\\A", vec)]
[1] "Arundo donax" "Achillea millefolium agg."

# Match only names where "agg." is at the end
vec[grepl("*agg.$", vec)]
[1] "Achillea millefolium agg." "Lotus corniculatus agg."

# Match names where agg. is in the middle (i.e. preceded and followed by a space)
vec[grepl("\\sagg\\.\\s", vec)]
[1] "Hieracium agg. heldreichii"
```

We can use lookarounds to match patterns based on the condition of following (lookaheads) or preceding (lookbehinds) strings elements. Note that you have to switch to Perl-style regular expression in the following functions (argument `'perl = T'`):

```
# Positive lookahead
# Get string elements with "Typha" that are followed by "pallida"
vec[grepl("Typha\\s(?:=pallida)",vec, perl=T)]
[1] "Typha pallida"

# Negative lookahead
# Get string elements with "Typha" that are NOT followed by "spec1"
vec[grepl("Typha\\s(?:!spec1)",vec, perl=T)]
[1] "Typha pallida"

# Lookbehinds
# Get strings with "agg." that are preceded by "Hieracium"
vec[grepl("(?<=Hieracium)\\sagg.", vec, perl=T)]
[1] "Hieracium agg. heldreichii"

# Get strings with "agg." that are NOT preceded by "Hieracium"
vec[grepl("(?!Hieracium)\\sagg.", vec, perl=T)]
[1] "Achillea millefolium agg." "Lotus corniculatus agg."
```

Using the same logic as above, you can also do inverse matching, where names that include a certain string element are excluded. For example, you could match all taxon names that do

NOT include "agg." in their strings:

```
vec[grep("^(?!agg.)*$", vec, perl=T)]
[1] "Arundo donax" "Typha spec1" "Typha pallida"
```

The `gsub()` function can match and replace certain string elements. In the example below, we match a double space and replace it by a single space (see "Lotus corniculatus agg."). Note that the expression "\\s{2,}" matches 'two or more white spaces'.

```
gsub("\\s{2,}", " ", vec)
[1] "Arundo donax" "Typha spec1" "Typha pallida"
"Achillea millefolium agg." "Lotus corniculatus agg." "Hieracium agg. heldreichii"
```

7 Reshaping tables: Wide \leftrightarrow Long

Vegetation tables often include species x plot data, which can be displayed in a wide or long format. The two formats can be converted through reshaping functions `gather()` and `spread()` in the `tidyr` package.

7.1 Wide format

In the wide format, the vegetation table lists plot IDs as a column, with one row for each plot. Abundances are listed across columns, with one column for each species (columns 'spec1' to 'spec4'). Header variables are listed as columns, again with every row representing a plot (latitude: 'lat' and longitude: 'long'). This format is suitable for plot-level operations, e.g. multivariate analysis in the `vegan` package.

```
lat.vec.w <- c(-25.628, 22.504, -3.515,
              1.400, -11.661, 16.936,
              -2.787, -11.767, 10.209,
              -5.747)

long.vec.w <- c(-27.501, -21.998, -48.671,
              17.081, 7.165, -0.839, -6.539, -0.644,
              17.802, NA)

wide.data <- data.frame(plot = c(paste("P", 1:10, sep="")),
                       spec1 = c(20, 5, 20, 1, 0, 0, 0, 20, 5, 10),
                       spec2 = c(0, 0, 40, 1, 0, 0, 0, 5, 0, 5),
                       spec3 = c(10, 50, 0, 5, 10, 0, 1, 0, 5, 10),
                       spec4 = c(5, 0, 10, 10, 30, 5, 0, 0, 50, 30),
                       lat = lat.vec.w,
                       long = long.vec.w)
```

```
wide.data
```

plot	spec1	spec2	spec3	spec4	lat	long
P1	20	0	10	5	-25.628	-27.501
P2	5	0	50	0	22.504	-21.998
P3	20	40	0	10	-3.515	-48.671
P4	1	1	5	10	1.400	17.081
P5	0	0	10	30	-11.661	7.165
P6	0	0	0	5	16.936	-0.839
P7	0	0	1	0	-2.787	-6.539
P8	20	5	0	0	-11.767	-0.644
P9	5	0	5	50	10.209	17.802
P10	10	5	10	30	-5.747	NA

7.2 Long format

In the long format, each row in the vegetation table corresponds to a plot x species observation, with species names and abundances gathered into two columns (note: only first ten rows are displayed, above). Hence, plot ID and other plot-level data (e.g. latitude and longitude for a plot) are listed repeatedly within a plot and are duplicated across the table. This format is the default format for many R operations. It offers the advantage that species names, layers, and abundance values can be more easily manipulated.

```
plot.vec <- c("P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8",
             "P9", "P10", "P1", "P2", "P3", "P4", "P5", "P6",
             "P7", "P8", "P9", "P10", "P1", "P2", "P3", "P4",
             "P5", "P6", "P7", "P8", "P9", "P10", "P1", "P2",
             "P3", "P4", "P5", "P6", "P7", "P8", "P9", "P10")

lat.vec <- c(-25.628, 22.504, -3.515, 1.400, -11.661, 16.936, -2.787, -11.767,
            10.209, -5.747, -25.628, 22.504, -3.515, 1.400, -11.661, 16.936,
            -2.787, 2-11.767, 10.209, -5.747, -25.628, 22.504, -3.515, 1.400,
            -11.661, 16.936, -2.787, -11.767, 10.209, -5.747, -25.628, 22.504,
            -3.515, 1.400, -11.661, 16.936, -2.787, -11.767, 10.209, -5.747)

long.vec <- c(27.501, -21.998, -48.671, 17.081, 7.165, -0.839, -6.539, -0.644,
            17.802, NA, -27.501, -21.998, -48.671, 17.081, 7.165, -0.839, -6.539,
            -0.644, 17.802, NA, -27.501, -21.998, -48.671, 17.081, 7.165, -0.839,
            -6.539, -0.644, 17.802, NA, -27.501, -21.998, -48.671, 17.081, 7.165,
            -0.839, -6.539, -0.644, 17.802, NA)

spec.vec <- c("spec1", "spec1", "spec1", "spec1", "spec1", "spec1", "spec1",
            "spec1", "spec1", "spec1", "spec2", "spec2", "spec2", "spec2",
            "spec2", "spec2", "spec2", "spec2", "spec2", "spec2", "spec3",
            "spec3", "spec3", "spec3", "spec3", "spec3", "spec3", "spec3",
            "spec3", "spec3", "spec4", "spec4", "spec4", "spec4", "spec4",
            "spec4", "spec4", "spec4", "spec4", "spec4")

abund.vec <- c(20, 5, 20, 1, 0, 0, 0, 20, 5, 10, 0, 0, 40, 1, 0, 0, 0, 5, 0, 5,
            10, 50, 0, 5, 10, 0, 1, 0, 5, 10, 5, 0, 10, 10, 30, 5, 0, 0,
            50, 30)

long.data <- data.frame(plot = plot.vec,
                       lat = lat.vec,
                       long = long.vec,
                       spec = spec.vec,
                       abund = abund.vec)

head(long.data)
```

	plot	lat	long	spec	abund
1	P1	-25.628	27.501	spec1	20
2	P2	22.504	-21.998	spec1	5
3	P3	-3.515	-48.671	spec1	20
4	P4	1.400	17.081	spec1	1
5	P5	-11.661	7.165	spec1	0
6	P6	16.936	-0.839	spec1	0

7.3 Wide ->Long: gather()

The transformation of a wide to a long format is known as gathering or melting data. The `gather()` operation converts the wide format to the long format. In the example below, we use this function to combine several species-abundance columns into just two columns, one containing the species names ('species') and one containing their abundances values ('abund'):

```
library(tidyr)
long.data <- gather(wide.data,
                    spec,      # new name of key variable
                    abund,    # new name of abundance variable
                    spec1:spec4, # name of columns to be gathered
                    convert = T) # automatic conversion to modes
```

Display first ten rows of the resulting table:

```
head(long.data, 10)

  plot    lat    long spec abund
1  P1 -25.628 -27.501 spec1    20
2  P2  22.504 -21.998 spec1     5
3  P3  -3.515 -48.671 spec1    20
4  P4   1.400  17.081 spec1     1
5  P5 -11.661   7.165 spec1     0
6  P6  16.936  -0.839 spec1     0
7  P7  -2.787  -6.539 spec1     0
8  P8 -11.767  -0.644 spec1    20
9  P9  10.209  17.802 spec1     5
10 P10 -5.747     NA spec1    10
```

7.4 Long ->Wide: spread()

We can reshape a table from the wide to the long format by using `spread()`. The spread operation is also known as data casting.

```
library(tidyr)
spread(long.data,
        spec, # key variable (will form column names)
        abund) # values for key variables (will form cell content)
```

```
  plot    lat    long spec1 spec2 spec3 spec4
1  P1 -25.628 -27.501    20    NA    10     5
2  P10 -5.747     NA    10     5    10    30
3  P2  22.504 -21.998     5    NA    50    NA
4  P3  -3.515 -48.671    20    40    NA    10
5  P4   1.400  17.081     1     1     5    10
6  P5 -11.661   7.165    NA    NA    10    30
7  P6  16.936  -0.839    NA    NA    NA     5
8  P7  -2.787  -6.539    NA    NA     1    NA
9  P8 -11.767  -0.644    20     5    NA    NA
10 P9  10.209  17.802     5    NA     5    50
```

`spread()` works similar to `gather()` in that a key variable and values are specified but this time, they are spread out into a wide dimension.

As some species occur in specific plots but are missing in others, we obtain missing values

('NA') for some cells in the species columns. To convert the 'NA's to zeroes, we need a little detour and define a function.

First define a function that will carry out the replacement:

```
repl.NA <- function(x) ifelse(is.na(x), 0,x)
```

In `ifelse()`, the first argument (`'is.na(x)'`) checks whether a value is NA, and returns TRUE if it is NA, or FALSE otherwise. The second and third arguments specify what should be written if the test returns TRUE or FALSE.

Now, we will apply the specified function to each species-abundance column by using dplyr's `mutate_each()` function. When custom functions are used inside `mutate_each()`, they must be wrapped in `funs()`. Columns that should not be included in the calculation can be excluded with the minus sign (here: 'plot', 'long', 'lat').

```
wide.data %>%  
mutate_each(funs(repl.NA), -plot, -long, -lat)%>%  
head()
```

	plot	lat	long	spec1	spec2	spec3	spec4
1	P4	1.400	17.081	0	0	5	0
2	P1	-25.628	-27.501	0	0	10	5
3	P1	-25.628	27.501	20	0	0	0
4	P10	-5.747	NA	10	5	10	30
5	P2	22.504	-21.998	5	0	50	0
6	P3	-3.515	-48.671	20	40	0	10

8 Adding or deleting rows and columns

8.1 Adding columns: mutate()

New columns can be added with dplyr's `mutate()` function. In the example below, we want to add a column 'year' to indicate the year in which the vegetation was sampled ('1990') and a column 'country' to indicate the name of the country in which the sample was taken. We will use `magrittr`'s compound assignment pipe operator `%<>%` to incorporate the changes into our data frame.

```
spec.vec <- c("Fagus sylvatica", "Acer platanoides",
             "Melica uniflora", "Hordelymus europaeus",
             "Lonicera periclymenum", "Lilium martagon",
             "Lathyrus vernus")

veg.df <- data.frame(plot = c(rep("P1", 7)),
                    spec = spec.vec,
                    layer = c("tr", "tr", "h", "h", "h", "h", "h"),
                    cov = c(60, 20, 5, 5, 5, 1, 5))
```

Display the data frame:

```
veg.df
```

	plot	spec	layer	cov
1	P1	Fagus sylvatica	tr	60
2	P1	Acer platanoides	tr	20
3	P1	Melica uniflora	h	5
4	P1	Hordelymus europaeus	h	5
5	P1	Lonicera periclymenum	h	5
6	P1	Lilium martagon	h	1
7	P1	Lathyrus vernus	h	5

```
library(dplyr)
library(magrittr)
```

```
veg.df %<>% mutate(year = 1990, country = "Czech Republic")
```

```
veg.df
```

	plot	species	layer	cov	year	country
1	P1	Fagus sylvatica	tr	60	1990	Czech Republic
2	P1	Acer platanoides	tr	20	1990	Czech Republic
3	P1	Melica uniflora	h	5	1990	Czech Republic
4	P1	Hordelymus europaeus f. elatior	h	5	1990	Czech Republic
5	P1	Lonicera periclymenum	h	5	1990	Czech Republic
6	P1	Lilium martagon	h	1	1990	Czech Republic
7	P1	Lathyrus vernus subsp. gracilis	h	5	1990	Czech Republic

8.2 Adding rows: `bind_rows()`

Two data frames can be combined with the `bind_rows()` function in `dplyr`, which is much faster than the solution in R `base`. They must have identical column names and their columns must be arranged in the same order.

In the example below, we add new rows to our existing data frame `veg.df` (the latter created above, including new columns 'year' and 'country'). Here is our second data frame:

```
new.df <- data.frame(plot = "P2",
                     spec = c("Carpinus betulus", "Acer pseudoplatanus",
                               "Viola reichenbachiana"),
                     layer = c("tr", "tr", "h"), cov = c(30,30,5),
                     year = 1994, country = "Austria",
                     stringsAsFactors = F)

new.df

  plot          spec layer cov year country
1  P2   Carpinus betulus   tr  30 1994 Austria
2  P2   Acer pseudoplatanus   tr  30 1994 Austria
3  P2   Viola reichenbachiana   h   5 1994 Austria
```

We check first whether column names are equal and in the same order. Then, we proceed to bind the two data frames.

```
cbind(colnames(veg.df), colnames(new.df))

  [,1]      [,2]
[1,] "plot"  "plot"
[2,] "spec"  "spec"
[3,] "layer" "layer"
[4,] "cov"   "cov"
[5,] "year"  "year"
[6,] "country" "country"
```

Are column names identical?

```
setequal(colnames(veg.df), colnames(new.df))
[1] TRUE
```

Are they in the same order?

```
identical(colnames(veg.df), colnames(new.df))
[1] TRUE
```

As the inspection of column names and their order yielded positive results, we will proceed to bind the two data frames:

```
bind_rows(list(veg.df, new.df))

Source: local data frame [10 x 6]

  plot          spec layer cov year          country
1  P1   Fagus sylvatica   tr  60 1990 Czech Republic
2  P1   Acer platanoides   tr  20 1990 Czech Republic
3  P1   Melica uniflora    h   5 1990 Czech Republic
```

4	P1	Hordelymus europaeus	h	5	1990	Czech Republic
5	P1	Lonicera periclymenum	h	5	1990	Czech Republic
6	P1	Lilium martagon	h	1	1990	Czech Republic
7	P1	Lathyrus vernus	h	5	1990	Czech Republic
8	P2	Carpinus betulus	tr	30	1994	Austria
9	P2	Acer pseudoplatanus	tr	30	1994	Austria
10	P2	Viola reichenbachiana	h	5	1994	Austria

8.3 Deleting columns

Existing column names can be deleted by assigning 'NULL' to them. Below are two options how to carry out the assignment. Suppose we want to remove the 'country' column from the `veg.df` data frame that we created in the two section above:

```
veg.df$country <- NULL

# OR

library(dplyr)
veg.df %<>% mutate(country = NULL)

head(veg.df)
  plot          spec layer cov year
1  P1      Fagus sylvatica   tr  60 1990
2  P1      Acer platanoides   tr  20 1990
3  P1      Melica uniflora    h   5 1990
4  P1 Hordelymus europaeus    h   5 1990
5  P1 Lonicera periclymenum   h   5 1990
6  P1      Lilium martagon    h   1 1990
```

8.4 Deleting rows

See [Chapter 10 'Select rows'](#) on how to remove rows based on a condition.

9 Manipulate columns and cell entries

9.1 Rename columns: mutate(), rename()

Below are some examples of how to change column names and values. All examples are using the compound assignment pipe (`%<>%`) to make immediate changes into the data frame and to avoid creating new data frames. This is particularly convenient with large tables. However, if you are unsure about the outcome, experiment first with the forward pipe (`%>%`) to create some temporary outcome.

First, an example of how to rename a column (from 'spec' to 'species') in a data frame *veg.df* with the function `rename()`:

```
veg.df <- data.frame(
  plot = c(rep("P1",7)),
  spec = c("Fagus sylvatica", "Acer platanoides", "Melica uniflora",
           "Hordelymus europaeus", "Lonicera periclymenum",
           "Lilium martagon", "Lathyrus vernus"),
  layer = c("tr", "tr", "h", "h", "h", "h", "h"),
  cov = c(60,20,5,5,5,1,5))
```

```
veg.df
```

	plot	spec	layer	cov
1	P1	Fagus sylvatica	tr	60
2	P1	Acer platanoides	tr	20
3	P1	Melica uniflora	h	5
4	P1	Hordelymus europaeus	h	5
5	P1	Lonicera periclymenum	h	5
6	P1	Lilium martagon	h	1
7	P1	Lathyrus vernus	h	5

Now, rename the column 'spec':

```
library(dplyr)
library(magrittr)

veg.df %<>% rename(species = spec)
```

```
veg.df
```

	plot	species	layer	cov
1	P1	Fagus sylvatica	tr	60
2	P1	Acer platanoides	tr	20
3	P1	Melica uniflora	h	5
4	P1	Hordelymus europaeus	h	5
5	P1	Lonicera periclymenum	h	5
6	P1	Lilium martagon	h	1
7	P1	Lathyrus vernus	h	5

9.2 Replace values: `mutate()`, `mapvalues()`, `replace()`

To change the names within the species column, we can use `dplyr`'s `mutate()` function in combination with `plyr`'s `mapvalues()`. The latter function replaces original values in a character or factor vector with new values.

Be aware that the `dplyr` and `plyr` packages can interfere. To avoid compatibility problems between them, we need to make sure that we first load `plyr` and then, `dplyr`. If you have worked with `dplyr` in previous operations, you should detach it from your workspace.

```
detach(package:dplyr)

library(plyr)
library(dplyr)
library(magrittr)
```

We will reuse the `veg.df` data frame from the previous section (as obtained after the renaming procedure). Suppose, we want to replace some species names in our column in the following way:

Lathyrus vernus -> *Lathyrus vernus* subs. *gracilis*
Hordelymus europaeus -> *Hordelymus europaeus* f. *elatior*:

```
veg.df %<>% mutate(species = mapvalues(species,
  c("Lathyrus vernus", "Hordelymus europaeus"),
  c("Lathyrus vernus subsp. gracilis",
    "Hordelymus europaeus f. elatior")))

veg.df

  plot      species layer cov
1  P1      Fagus sylvatica   tr 60
2  P1      Acer platanoides   tr 20
3  P1      Melica uniflora    h  5
4  P1 Hordelymus europaeus f. elatior    h  5
5  P1      Lonicera periclymenum    h  5
6  P1      Lilium martagon    h  1
7  P1 Lathyrus vernus subsp. gracilis    h  5
```

In a similar manner, we can change numeric entries based on conditions. Let's say, we want to change (for some arbitrary reason) all cover values of '5' to '10':

```
veg.df %<>% mutate(cov = mapvalues(cov, 5, 10))

veg.df

  plot      species layer cov
1  P1      Fagus sylvatica   tr 60
2  P1      Acer platanoides   tr 20
3  P1      Melica uniflora    h 10
4  P1 Hordelymus europaeus f. elatior    h 10
5  P1      Lonicera periclymenum    h 10
6  P1      Lilium martagon    h  1
7  P1 Lathyrus vernus subsp. gracilis    h 10
```

In addition, we can replace values based on a condition. Here is an example where we replace all cover values < '15' with 'NA', by using the `replace()` function in **base R**:

```
veg.df %<>% mutate(cov = replace(cov, cov<15, NA))
```

```
veg.df
  plot      species layer cov
1  P1      Fagus sylvatica   tr  60
2  P1      Acer platanoides   tr  20
3  P1      Melica uniflora    h   NA
4  P1 Hordelymus europaeus f. elatior   h  NA
5  P1      Lonicera periclymenum    h  NA
6  P1      Lilium martagon        h  NA
7  P1 Lathyrus vernus subsp. gracilis   h  NA
```

9.3 Update values by group

A common problem in data preparation is that plots of a particular group lack values or have values that need to be updated. In the example below, we have soil pH data for five countries. For two plots from Italy, soil pH values are lacking in the initial dataset *env.df* and need to be updated. We can use the **data.table** package to update these values, while leaving other values in the same column intact.

```
# Environmental dataset
env.df <- data.frame(plot = paste("P",1:10,sep = ""),
  soil.ph = c(NA,NA,8.8,6.1,5.5,7.1,6.3,6.6,6.1,5.5),
  region = c(rep("Italy", 3), "Australia", rep("Canada",2),
  rep("Russia",2), rep("Brazil",2)),
  stringsAsFactors = F)
```

```
env.df
  plot soil.ph region
1  P1      NA   Italy
2  P2      NA   Italy
3  P3      8.8   Italy
4  P4      6.1 Australia
5  P5      5.5   Canada
6  P6      7.1   Canada
7  P7      6.3   Russia
8  P8      6.6   Russia
9  P9      6.1   Brazil
10 P10     5.5   Brazil
```

```
# Dataset with updated soil.pH values for Italy
```

```
plot.it <- paste("P",1:2,sep="")
```

```
soil.ph <- c(7.1,7.3)
```

```
italy.df <- data.frame(
  plot = plot.it,
  soil.ph = soil.ph,
  stringsAsFactors = F)
```

```
italy.df
  plot soil.ph
1  P1      7.1
2  P2      7.3
```

Now, load necessary packages:

```
library(data.table)
library(dplyr)
library(magrittr)
```

Next, transform the data frame to data table and add a new column ('id') to highlight the plots that need a new assignment.

```
env.df <- setDT(as.list(env.df))
env.df %<>% mutate(id = ifelse(is.na(soil.ph & region == 'Italy'), 1,0))

italy.df <- setDT(as.list(italy.df))
italy.df%<>% mutate(id = 1,region = 'Italy')
```

Set key columns that will join the two datasets and then update values for plots:

```
setkey(env.df, plot, region, id)
setkey(italy.df, plot, region, id)

env.df[italy.df, soil.ph := i.soil.ph, nomatch = 0]
```

In the resulting dataset you can see that soil pH values have been updated for the two plots from Italy. Note that the joining operation has resulted in a new order of rows (but content has not been altered).

```
env.df
  plot soil.ph  region id
1:  P1    7.1    Italy  1
2: P10    5.5   Brazil  0
3:  P2    7.3    Italy  1
4:  P3    8.8    Italy  0
5:  P4    6.1 Australia 0
6:  P5    5.5   Canada  0
7:  P6    7.1   Canada  0
8:  P7    6.3   Russia  0
9:  P8    6.6   Russia  0
10: P9    6.1   Brazil  0
```

10 Rearranging order of rows and columns

The **order of rows** can be rearranged using the **dplyr**'s **arrange()** function. Let's create an example:

```
data <- data.frame(species = c("Sp1", "Sp1", "Sp2", "Sp3", "Sp1", "Sp3", "Sp3",  
                             "Sp4", "Sp1", "Sp4"),  
                  layer = c(5, 6, 6, 6, 6, 4, 6, 6, 6, 6),  
                  plot = c(rep("P1", 4), rep("P2", 4), rep("P3", 2)),  
                  perc = c(20, 50, 1, 5, 3, 15, 20, 5, 50, 10),  
                  region = c(rep("A", 8), rep("B", 2)))
```

```
data
```

	species	layer	plot	perc	region
1	Sp1	5	P1	20	A
2	Sp1	6	P1	50	A
3	Sp2	6	P1	1	A
4	Sp3	6	P1	5	A
5	Sp1	6	P2	3	A
6	Sp3	4	P2	15	A
7	Sp3	6	P2	20	A
8	Sp4	6	P2	5	A
9	Sp1	6	P3	50	B
10	Sp4	6	P3	10	B

Now, rearrange rows based on increasing cover percentages:

```
arrange(data, perc)
```

	species	layer	plot	perc	region
1	Sp2	6	P1	1	A
2	Sp1	6	P2	3	A
3	Sp3	6	P1	5	A
4	Sp4	6	P2	5	A
5	Sp4	6	P3	10	B
6	Sp3	4	P2	15	A
7	Sp1	5	P1	20	A
8	Sp3	6	P2	20	A
9	Sp1	6	P1	50	A
10	Sp1	6	P3	50	B

The **order of columns** can be rearranged using **dplyr**'s **select()** function. Let's say we want to rearrange columns to match the order of: 'plot', 'species', 'layer', 'perc', 'region'. It's important to include all column names of the table, otherwise they are dropped.

```
select(data, plot, species, layer, perc, region)
```

	plot	species	layer	perc	region
1	P1	Sp1	5	20	A
2	P1	Sp1	6	50	A
3	P1	Sp2	6	1	A
4	P1	Sp3	6	5	A
5	P2	Sp1	6	3	A

6	P2	Sp3	4	15	A
7	P2	Sp3	6	20	A
8	P2	Sp4	6	5	A
9	P3	Sp1	6	50	B
10	P3	Sp4	6	10	B

11 Select rows

Use `dplyr`'s `slice()` function to select rows or delete rows based on their number. Here is an example:

```
data <- data.frame(  
  species = c("Sp1", "Sp1", "Sp2", "Sp3", "Sp1", "Sp3", "Sp3", "Sp4", "Sp1", "Sp4"),  
  layer = c(5, 6, 6, 6, 6, 4, 6, 6, 6, 6),  
  plot = c(rep("P1", 4), rep("P2", 4), rep("P3", 2)),  
  perc = c(20, 50, 1, 5, 3, 15, 20, 5, 50, 10),  
  region = c(rep("A", 8), rep("B", 2))
```

```
data  
  
  species layer plot perc region  
1     Sp1     5   P1   20      A  
2     Sp1     6   P1   50      A  
3     Sp2     6   P1    1      A  
4     Sp3     6   P1    5      A  
5     Sp1     6   P2    3      A  
6     Sp3     4   P2   15      A  
7     Sp3     6   P2   20      A  
8     Sp4     6   P2    5      A  
9     Sp1     6   P3   50      B  
10    Sp4     6   P3   10      B
```

Choose first three rows:

```
slice(data, 1:3)  
  
  species layer plot perc region  
1     Sp1     5   P1   20      A  
2     Sp1     6   P1   50      A  
3     Sp2     6   P1    1      A
```

Omit first three rows:

```
slice(data, -(1:3))  
  
  species layer plot perc region  
1     Sp3     6   P1    5      A  
2     Sp1     6   P2    3      A  
3     Sp3     4   P2   15      A  
4     Sp3     6   P2   20      A  
5     Sp4     6   P2    5      A  
6     Sp1     6   P3   50      B  
7     Sp4     6   P3   10      B
```

Use `dplyr`'s `filter()` function to select or delete rows based on a condition.

Select plots from region "A":

```
filter(data, region == "A")
```

	species	layer	plot	perc	region
1	Sp1	5	P1	20	A
2	Sp1	6	P1	50	A
3	Sp2	6	P1	1	A
4	Sp3	6	P1	5	A
5	Sp1	6	P2	3	A
6	Sp3	4	P2	15	A
7	Sp3	6	P2	20	A
8	Sp4	6	P2	5	A

Omit rows from region "B". Same result as above but different syntax:

```
filter(data, !region == "B")
```

	species	layer	plot	perc	region
1	Sp1	5	P1	20	A
2	Sp1	6	P1	50	A
3	Sp2	6	P1	1	A
4	Sp3	6	P1	5	A
5	Sp1	6	P2	3	A
6	Sp3	4	P2	15	A
7	Sp3	6	P2	20	A
8	Sp4	6	P2	5	A

12 Select columns

To select specific columns, use `dplyr`'s `select()` function. In the example below, we reuse the example of the previous chapter in order to select columns for species name, layer and their cover.

```
data %>% select(species, layer, perc)
```

	species	layer	perc
1	Sp1	5	20
2	Sp1	6	50
3	Sp2	6	1
4	Sp3	6	5
5	Sp1	6	3
6	Sp3	4	15
7	Sp3	6	20
8	Sp4	6	5
9	Sp1	6	50
10	Sp4	6	10

13 Combining datasets

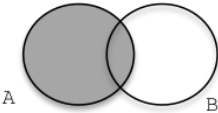
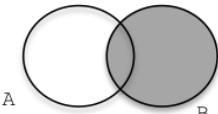
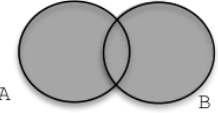
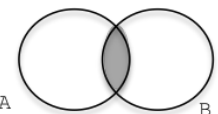
13.1 Joining

Joining (also known as merging) combines two or more datasets based on one or more key variable(s). In vegetation science, an example of a common joining operation is the assignment of traits to species in a vegetation table. In this case, the vegetation and trait datasets are combined based on a key variable for species name.

R offers several functions to join tables. Below we will use functions in the **dplyr** package (Fig. 12.1). Other possibilities would be `merge()` in R **base** and in **data.table**, as well as the more concise `I[J]` in **data.table**.

Datasets can be combined in different ways. In general, outer joins are performed when two datasets do not completely overlap in their key variable and unmatched rows should be appended to the resulting table (with 'NA' listed for cells in which information could not be filled). By comparison, inner joins are carried out to keep only rows with an overlap in the key variable and to drop rows with non-overlap from the resulting table.

Fig. 13.1: An overview of the most common join operations

Join operation		dplyr command
Left outer join		<code>left_join()</code>
Right outer join		<code>right_join()</code>
Full outer join		<code>full_join()</code>
Inner join		<code>inner_join()</code>

Below are two exemplary datasets that will be merged in four different ways:

VEGETATION DATA (Table A)

```
plot.vec <- plot = c(rep("P1",8), rep("P2",3))

spec.vec <- c("Fagus sylvatica", "Acer platanoides",
             "Fagus sylvatica", "Melica uniflora",
             "Hordelymus europaeus", "Lonicera periclymenum",
             "Lilium martagon", "Lathyrus vernus",
             "Fagus sylvatica", "Stellaria holostea",
             "Viola reichenbachiana")

layer.vec <- c("tr","tr", "tr", "h", "h", "h","h","h","tr", "h", "h")

cov <- c(60,30,20,5,5,5,1,5,50,15,5)

veg.df <- data.frame(
  plot = plot.vec,
  spec = spec.vec,
  layer = layer.vec,
  cov = cov.vec,
  stringsAsFactors = F)
```

veg.df

	plot	spec	layer	cov
1	P1	Fagus sylvatica	tr	60
2	P1	Acer platanoides	tr	30
3	P1	Fagus sylvatica	tr	20
4	P1	Melica uniflora	h	5
5	P1	Hordelymus europaeus	h	5
6	P1	Lonicera periclymenum	h	5
7	P1	Lilium martagon	h	1
8	P1	Lythyrus vernus	h	5
9	P2	Fagus sylvatica	tr	50
10	P2	Stellaria holostea	h	15
11	P2	Viola reichenbachiana	h	5

TRAIT DATA (Table B)

```
species.vec.2 <- c("Acer platanoides", "Antennaria dioica",
                 "Dactylis glomerata", "Dentaria bulbifera",
                 "Fagus sylvatica", "Hordelymus europaeus",
                 "Lilium martagon", "Lonicera periclymenum",
                 "Lythyrus vernus", "Melica uniflora",
                 "Poa annua", "Stellaria holostea",
                 "Tragopogon dubius")

life_form.vec <- c("p", "c", "h", "g", "p", "h",
                 "g","l", "g, h", "g, h", "h, t", "c", "h")

pollination.vec <- c("i, w","i", "w", "i, self", "w", "w",
                  "i", "i", "i", "w","w", "i, self", "i, self")

trait.df <- data.frame(species = species.vec.2,
                      life_form = life_form.vec,
                      pollination = pollination.vec,
                      stringsAsFactors = F)
```

```

trait.df

      species life_form pollination
1   Acer platanoides      p      i, w
2  Antennaria dioica      c          i
3  Dactylis glomerata      h          w
4  Dentaria bulbifera      g      i, self
5   Fagus sylvatica      p          w
6  Hordelymus europaeus      h          w
7   Lilium martagon      g          i
8  Lonicera periclymenum      l          i
9   Lythyrus vernus      g, h          i
10  Melica uniflora      g, h          w
11   Poa annua      h, t          w
12  Stellaria holostea      c      i, self
13  Tragopogon dubius      h      i, self

```

(1) The **left outer join** is probably the most important function for manipulating vegetation datasets. In this case, we keep all the rows from table A and add data from B that match the entries in the key variable. If rows in A do not have matching key values in B, then their new values will get an 'NA' assignment. In the example below, we use a left outer join to assign traits to species in a vegetation table. Our key variable is species names. Note that all rows in the vegetation table will be preserved, while only those rows in traits database will be appended that have a matching species name.

```

# Left outer join

library(dplyr)
new.df0 <- left_join(veg.df, trait.df,
                    by = c("spec" = "species"))
new.df0

      plot      spec layer cov life_form pollination
1   P1   Fagus sylvatica  tr  60          p          w
2   P1   Acer platanoides  tr  30          p      i, w
3   P1   Fagus sylvatica  tr  20          p          w
4   P1   Melica uniflora   h   5      g, h          w
5   P1  Hordelymus europaeus h   5          h          w
6   P1  Lonicera periclymenum h   5          l          i
7   P1   Lilium martagon   h   1          g          i
8   P1   Lythyrus vernus   h   5      g, h          i
9   P2   Fagus sylvatica  tr  50          p          w
10  P2  Stellaria holostea  h  15          c      i, self
11  P2  Viola reichenbachiana h   5      <NA>      <NA>

```

Note that joining operations in **dplyr** do not require key variables to have the same name.

You can restrict which columns you want to join by using square-bracket indexing. However, it is important that the key variable is included in each dataset. In the example below, we assign only information on pollination type to the species in our vegetation dataset.

```

new.df1 <- left_join(veg.df, trait.df[,c("species", "pollination")],
                    by = c("spec" = "species"))

```

```
new.df1
```

	plot	spec	layer	cov	pollination
1	P1	Fagus sylvatica	tr	60	w
2	P1	Acer platanoides	tr	30	i, w
3	P1	Fagus sylvatica	tr	20	w
4	P1	Melica uniflora	h	5	w
5	P1	Hordelymus europaeus	h	5	w
6	P1	Lonicera periclymenum	h	5	i
7	P1	Lilium martagon	h	1	i
8	P1	Lythyrus vernus	h	5	i
9	P2	Fagus sylvatica	tr	50	w
10	P2	Stellaria holostea	h	15	i, self
11	P2	Viola reichenbachiana	h	5	<NA>

(2) In a **right outer join**, we keep all the rows from table B and add data from A that match the entries in the key variable. A right join might not make so much sense for the example above, but let's inspect it nonetheless to understand the outcome:

```
# Right outer join

new.df2 <- right_join(veg.df, trait.df, by = c("spec" = "species"))
new.df2
```

	plot	spec	layer	cov	life_form	pollination
1	P1	Acer platanoides	tr	30	p	i, w
2	<NA>	Antennaria dioica	<NA>	NA	c	i
3	<NA>	Dactylis glomerata	<NA>	NA	h	w
4	<NA>	Dentaria bulbifera	<NA>	NA	g	i, self
5	P1	Fagus sylvatica	tr	60	p	w
6	P1	Fagus sylvatica	tr	20	p	w
7	P2	Fagus sylvatica	tr	50	p	w
8	P1	Hordelymus europaeus	h	5	h	w
9	P1	Lilium martagon	h	1	g	i
10	P1	Lonicera periclymenum	h	5	l	i
11	P1	Lythyrus vernus	h	5	g, h	i
12	P1	Melica uniflora	h	5	g, h	w
13	<NA>	Poa annua	<NA>	NA	h, t	w
14	P2	Stellaria holostea	h	15	c	i, self
15	<NA>	Tragopogon dubius	<NA>	NA	h	i, self

(3) In an **outer join**, all rows of table A and B are listed in the resulting dataset, even if their values for the key variable do not match. Rows in table A and B that intersect in the key variable (i.e. are found in both datasets) will be combined into one row, while rows that do not intersect will be appended separately at the bottom, with 'NA' entries in new columns.

```
new.df3 <- full_join(veg.df, trait.df, by = c("spec" = "species"))
new.df3
```

	plot	spec	layer	cov	life_form	pollination
1	P1	Fagus sylvatica	tr	60	p	w
2	P1	Acer platanoides	tr	30	p	i, w
3	P1	Fagus sylvatica	tr	20	p	w
4	P1	Melica uniflora	h	5	g, h	w
5	P1	Hordelymus europaeus	h	5	h	w
6	P1	Lonicera periclymenum	h	5	l	i
7	P1	Lilium martagon	h	1	g	i
8	P1	Lythyrus vernus	h	5	g, h	i

9	P2	Fagus sylvatica	tr	50	p	w
10	P2	Stellaria holostea	h	15	c	i, self
11	P2	Viola reichenbachiana	h	5	<NA>	<NA>
12	<NA>	Antennaria dioica	<NA>	NA	c	i
13	<NA>	Dactylis glomerata	<NA>	NA	h	w
14	<NA>	Dentaria bulbifera	<NA>	NA	g	i, self
15	<NA>	Poa annua	<NA>	NA	h, t	w
16	<NA>	Tragopogon dubius	<NA>	NA	h	i, self

(4) An **inner join** combines only the rows which have matching values in the key variable; rows with non-matching values will be dropped in each table, respectively. In our example, rows in *veg.df* with 'spec' == "Viola reichenbachiana" and rows in *trait.df* with 'species' == "Tragopogon dubius", "Poa annua", "Dactylis glomerata" and "Antennaria dioica" will be dropped because these species do not occur in both datasets. Be careful when you use this joining operation because you might accidentally loose rows in your table (!).

```
new.df4 <- inner_join(veg.df,trait.df, by = c("spec" = "species"))
new.df4
```

	plot	spec	layer	cov	life_form	pollination
1	P1	Fagus sylvatica	tr	60	p	w
2	P1	Acer platanoides	tr	30	p	i, w
3	P1	Fagus sylvatica	tr	20	p	w
4	P1	Melica uniflora	h	5	g, h	w
5	P1	Hordelymus europaeus	h	5	h	w
6	P1	Lonicera periclymenum	h	5	l	i
7	P1	Lilium martagon	h	1	g	i
8	P1	Lythyrus vernus	h	5	g, h	i
9	P2	Fagus sylvatica	tr	50	p	w
10	P2	Stellaria holostea	h	15	c	i, self

14 Data cleaning

14.1 Spot duplicate entries

Duplicate entries can be a data quality concern that need to be dealt with before analyzing data. In the example below, the same species appears twice in a plot, in the same tree layer.

```
veg.df <- data.frame(
  plot = c(rep("P1",8), rep("P2",3)),
  spec = c("Fagus sylvatica", "Acer platanoides",
           "Fagus sylvatica", "Melica uniflora",
           "Hordelymus europaeus", "Lonicera periclymenum",
           "Lilium martagon", "Lythyrus vernus",
           "Fagus sylvatica", "Stellaria holostea",
           "Dentaria bulbifera"),
  layer = c("tr","tr", "tr", "h", "h", "h","h","h","tr", "h", "h"),
  cov = c(60,30,20,5,5,5,1,5,50,15,5))
```

```
veg.df
  plot      spec layer cov
1  P1  Fagus sylvatica   tr  60
2  P1  Acer platanoides   tr  30
3  P1  Fagus sylvatica   tr  20
4  P1  Melica uniflora    h    5
5  P1  Hordelymus europaeus h    5
6  P1  Lonicera periclymenum h    5
7  P1  Lilium martagon    h    1
8  P1  Lythyrus vernus    h    5
9  P2  Fagus sylvatica   tr  50
10 P2  Stellaria holostea h   15
11 P2  Dentaria bulbifera h    5
```

Using the example above, we can spot duplicate entries within plots by specifying the grouping structure using **dplyr**'s **by_group()**. In our case, the grouping structure is the combination of 'plot', 'spec' and 'layer'. Then, we use **dplyr**'s **filter()** to extract combinations that occur more than once.

```
# Show duplicate combinations

veg.df %>%
  group_by(plot, spec, layer) %>%
  filter(n()>1)

  plot      spec layer cov
1  P1  Fagus sylvatica   tr  60
2  P1  Fagus sylvatica   tr  20
```

There is no standard way of how to deal with duplicate plot x species x layer entries. The most rigorous options is to remove plots from your dataset with the **filter()** function. In this case, it might be helpful to get a vector of plot numbers in which duplicate entries were found:

```
veg.df %>%
  group_by(plot, spec, layer) %>%
  filter(n()>1) %>%
  as.vector(unique(extract(plot)))

[1] "P1"
```

You could also remove one of the two rows by using `filter()` or you can merge their cover values with `veggie`'s `merge_cov()` function (Chapter 14).

14.2 Spell-checking taxon names

The correct spelling of taxon names is an important prerequisite for many data processing steps, such as search and replace functions and join operations. When dealing with small vegetation datasets, taxon names can be checked manually for mistakes. However, with large vegetation datasets, this task becomes inefficient, non-reproducible and prone to errors. In this section, you learn how to use R tools to spell-check taxon names.

14.2.1 General spelling mistakes

There are some R functions that can deal with general spelling mistakes, like redundant white spaces and invisible white characters. Below, we use (1) the `stringr` package and (2) the `gsub()` function in **R base** in combination with regular expressions (see chapter 6.2) to spot general mistakes in taxon names.

Imagine we have the following string of taxon names that includes redundant spaces before, in the middle, and at the end of taxon names (see entries for *Betula pendula*, *Vaccinium myrtillus*, and *Cladonia rangiferina*).

```
species <-c("Pinus sylvestris",
  "  Betula pendula",
  "Vaccinium  myrtillus",
  "Deschampsia flexuosa",
  "Luzula pilosa",
  "Dicranum scoparium",
  "Cladonia rangiferina ")
```

White spaces at the beginning and end of taxon names can be deleted with `stringr`'s `str_trim()` function:

```
library(stringr)

species2 <- str_trim(species)

species2

[1] "Pinus sylvestris"      "Betula pendula"
[3] "Vaccinium  myrtillus" "Deschampsia flexuosa"
[5] "Luzula pilosa"        "Dicranum scoparium"
[7] "Cladonia rangiferina"
```


The `gsub()` function can substitute white spaces across the entire string. It consists of three components, the regular expression for matching (here: match two or more spaces; `"\\s{2,}"`), a specification how the matched expression should be replaced (by a white space (" "), in our case), and the name of the vector for which the operation should be carried out ('species2').

```
gsub("\\s{2,}", " ", species2)

[1] "Pinus sylvestris"      "Betula pendula"      "Vaccinium myrtillus"
[4] "Deschampsia flexuosa" "Luzula pilosa"      "Dicranum scoparium"
[7] "Cladonia rangiferina"
```

Alternatively, we can pipe all above functions into a single line with the `magrittr` package. The third element within `gsub()` should include a dot instead of the vector name:

```
species %>% str_trim %>% gsub("\\s{2,}", " ", .)

[1] "Pinus sylvestris"      "Betula pendula"      "Vaccinium myrtillus"
[4] "Deschampsia flexuosa" "Luzula pilosa"      "Dicranum scoparium"
[7] "Cladonia rangiferina"
```

In some rare cases, taxon names can include invisible whitespace characters, like a middle dot (".") that is by default neither displayed in the R console, file nor in the editor. Two taxon names that differ in having a middle dot or regular space (and are otherwise identical) will not be recognized as identical by R.

By definition, it is impossible to create a reproducible and visible object, here. But in general, you can match invisible whitespaces with the regular expression `"\\p{Zs}"` and substitute them by normal whitespaces with the `gsub()` function. The argument `'perl=TRUE'` specifies that Perl-style regular expressions should be used:

```
gsub("\\p{Zs}", " ", species, perl=TRUE)]
```

14.2.2 Spell-checking with reference lists

Many spelling mistakes in taxon names can only be detected by using a reference list that includes correctly spelt names. R offers three packages (`tpl`, `Taxonstand`, `taxize`) that reference global taxonomic databases and match names via fuzzy algorithms.

The choice for one of these three package depends on the taxonomic breadth of your database. Both `Taxonstand` and `tpl` rely on The Plant List and thus, can only spell-check vascular plant and bryophyte taxon names. `Taxonstand` is more limited because it references The Plant List on the web, which only allows to spell-check species epithets (and not genus names). By contrast, `tpl` downloads the entire The Plant List database on your computer (via the `tpldata` package). This gives `tpl` the freedom to also spell-check genus names.

However, the most versatile tool is the `gnr_resolve()` function in the `taxize` package. It calls the Global Names Resolver (GNR), which employs a modified taxamatch algorithm (Rees 2014) and references multiple global taxonomic databases. Consequently, it can detect spelling mistakes in names of vascular plants, bryophytes, fungi, lichens, and algae, at different taxonomic levels (family, genus, species, subspecies, variety). In the following, you can see how `gnr_resolve()` works and how you can incorporate its results through a semi-automatic approach.

I created some arbitrary data (from temperate vegetation in Europe). Hashtags indicate spelling mistakes.

```
library(taxize)
library(dplyr)
library(magrittr)

veg.df <- data.frame(taxa = c(
  "Scorsonera villosa", # correct spelling: Scorzonera villosa
  "Abietinela",        # Abietinella
  "Acer tatarica",     # Acer tataricum
  "Stipa johannis",    # Stipa joannis
  "Anthriscus nitidum", # Anthriscus nitida
  "Xanthopaemelia",   # Xanthoparmelia
  "Bolettus",         # Boletus
  "Fagus sylvatica",
  "Ompalina",         # Omphalina
  "Pooaceae",         # Poaceae
  NA),
  abund = c(12, 5, 20, 1, 1, 1, 10, 15, 10, 5, 2),
  plot=c(99, 12, 100, 15, 99, 70, 88, 201, 9, 49, 52))
```

You can use 94 different sources to match your name against. In the next line we can inspect the full list of available sources. (Uncomment hashtag to see full list):

```
# gnr_datasources()
```

Next, specify the reference databases you want to use. If you don't indicate databases, GNR will match your names against all sources.

```
src <- c("EOL", "The International Plant Names Index",
  "Index Fungorum", "ITIS", "Catalogue of Life",
  "Tropicos - Missouri Botanical Garden")
```

Show specified sources:

```
subset(gnr_datasources(), title %in% src)
  id title
1  1 Catalogue of Life
3  3 ITIS
5  5 Index Fungorum
12 12 EOL
83 165 Tropicos - Missouri Botanical Garden
85 167 The International Plant Names Index
```

Submit request to the Global Names Resolver. You need an internet connection for this step. The important arguments are:

- stripauthority: author names should be excluded in display
- with_canonical_ranks: include ranks (e.g. "subsp.")

```
result.long <- veg.df$taxa %>%
  gnr_resolve(data_source_ids = c(1, 5, 12, 165, 167),
    with_canonical_ranks=T)
```

Inspect full results table. The columns indicate the following:

- submitted_name: original/supplied taxon names
- matched_name2: taxon names matched by GNR
- score: matching score (with 1.0 being the best score)
- data_source_title: name of reference database

```
head(result.long)

  submitted_name      data_source_title score matched_name2
6      Abietinela      Catalogue of Life  0.50  Abietinella
7      Abietinela      EOL                0.50  Abietinella
12     Abietinela Tropicos - Missouri Botanical Garden 0.50  Abietinella
13     Abietinela Tropicos - Missouri Botanical Garden 0.50  Abietineae
14  Acer tatarica      Catalogue of Life  0.75  Acer tataricum
15  Acer tatarica      EOL                0.75  Acer tataricum
```

I use a ‘semi-automatic’ approach for cleaning spelling mistakes in taxon names, which has three advantages:

- 1) I control what I want to implement. This is particularly important when you think that algorithms can suggest different names with same score but only one matching result makes sense. For example, EOL and Tropicos suggested that the name ‘Poaceae’ could be matched either to “Poaceae” or “Podoceae”. The latter name is a synonym of Anacardiaceae (according to USDA ARS GRIN Taxonomy), family which is mostly distributed in the tropics and subtropics. Given that I am working with data from temperate Europe, ‘Poaceae’ is my choice no. 1, here.
- 2) In case GNR matching results are not suitable or matching failed, I can specify and implement alternative names (or use placeholders).
- 3) I make sure that my observations are not multiplied. With a fully automatic (‘blind’) implementation, I run the risk that the number of observations and the number of taxon names in my database is multiplied. For example, joining by “Poaceae” will create two observation for this taxon names, one for “Podoceae” and one for “Poaceae”. In other words, my observation would be forked into two observations.

Next, export your results to a .txt file (modify content in quotes to match your path and insert name of object to be exported).

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

Insert three new columns (change names as you like) and insert text:

- ‘implement’ - should the name suggested by GNR be used? (TRUE/FALSE)?
- ‘alternative’ - write an alternative name here
- ‘dupl’ - Is this entry a duplicate (TRUE/FALSE)?

Save the file under a new name (I created ‘result.short.COMMENTS.txt’) and import it into R:

```
corr.df <- read.table("result.short.COMMENTS.txt",
                     sep="\t", header=T, stringsAsFactors = F)
```

It's important to discard duplicates in the imported table (highlighted in the new 'dupl' column) so that joining does not result in a multiplication of rows.

```
corr.df %<>% filter(!dupl ==T)
```

Now, join with original dataset and implement changes. The function below will implement changes through three ifelse() statements:

```
taxa.df.2 <- veg.df %>%
  left_join(corr.df, by=c("taxa" = "submitted_name")) %>%
  mutate(new.taxon = ifelse(implement == T, matched_name2,
    ifelse(implement == F & is.na(alternative)==T, taxa,
    ifelse(implement == F & is.na(alternative)==F, alternative))))
```

A warning might pop up (as above). It just tells you that the key columns are of different class (character vs. factor). No reason to be worried.

Now, check your results and create a new object. If you work with a very large object, you can directly implement changes in your dataframe by using the compound assignment pipe operator `%<>%`.

```
veg.df.2 <- veg.df %>%
  left_join(corr.df, by=c("taxa" = "submitted_name")) %>%
  mutate(new.taxon = ifelse(implement == T, matched_name2,
    ifelse(implement == F & is.na(alternative)==T, taxa,
    ifelse(implement == F & is.na(alternative)==F, alternative)))) %>%
  # Discard columns
  select(-matched_name2, -score, -implement, -alternative, - dupl, -taxa) %>%
  # Rename your column storing the new taxon names
  rename(taxon = new.taxon) %>%
  # Sort column in the original order (optional)
  select(taxon, abund, plot)
```

And here is our new data frame with correctly spelt taxon names:

```
veg.df.2
```

	taxon	abund	plot
1	Scorzonera villosa	12	99
2	Abietinella	5	12
3	Acer tataricum	20	100
4	Stipa joannis	1	15
5	Anthriscus nitida	1	99
6	Xanthoparmelia	1	70
7	Boletus	10	88
8	Fagus sylvatica	15	201
9	Omphalina	10	9
10	Poaceae	5	49
11	<NA>	2	52

An important topic that is not covered in this manual is taxonomic standardization. There are R packages that can retrieve synonyms from taxonomic databases, including the already mentioned **tpl**, **Taxonstand** and **taxize** packages, as well as the packages **taxizesoap** package which references **the Euro+Med PlantBase**. However, incorporating their results into vegetation data is tricky and requires taxonomic knowledge. I am currently trying to figure out a semi-automatic workflow that won't mess up the data.

References

Rees, T. 2014. Taxamatch, an Algorithm for Near ('Fuzzy') matching of scientific names in taxonomic databases. *PloS One*: 0107510.

15 Merging cover values

In addition to species x plot data, vegetation tables can also include information for the layer in which a species occurs in. Below is an example with the data frame *veg.df*:

```
plot.ID.vec <- paste("P",c(1,1,1,1,1, 2,2,2,2,2,2,2),sep = "")

species.name.vec <- c(rep("Spec1", 3),"Spec2",
                      "Spec3",rep("Spec1", 2) "Spec5",
                      "Spec4", rep("Spec6",3))

layer.vec <- c(6,5,4,6,6, 4,5,2,3,4,5,5)

abund.vec <- c(0.4,0.2,0.2,0.1,0.8,0.6,0.4,0.4,0.7,0.1,0.6,0.2)

region.vec <- region=c(rep("Region1",5), rep("Region2",7))

veg.df <- data.frame(plot.ID = plot.ID.vec,
                    species.name = spec.name.vec,
                    layer = layer.vec,
                    abund = abund.vec,
                    region = region.vec)
```

```
veg.df

  plot.ID species.name layer abund region
1      P1      Spec1     6  0.4 Region1
2      P1      Spec1     5  0.2 Region1
3      P1      Spec1     4  0.2 Region1
4      P1      Spec2     6  0.1 Region1
5      P1      Spec3     6  0.8 Region1
6      P2      Spec1     4  0.6 Region2
7      P2      Spec1     5  0.4 Region2
8      P2      Spec5     2  0.4 Region2
9      P2      Spec4     3  0.7 Region2
10     P2      Spec6     4  0.1 Region2
11     P2      Spec6     5  0.6 Region2
12     P2      Spec6     5  0.2 Region2
```

We can merge cover values for species across its layers in a plot. This operation can be carried out under the assumption of independent overlap among layers or under the assumption of no overlap (Tichy & Holt 2006, Fischer 2014). In the latter case, the resulting cover is the sum of the covers by each layer.

In the following example, we use **veggie**'s `merge_cov()` function to merge cover for identical species within a plot, irrespective of the layer they occur in. By default, the function assumes independent overlap among layers:

```
library(veggie)

merge_cov(veg.df, cover = "abund", plot = "plot.ID", taxacol = "species.name",
          layercol = "layer")

Source: local data frame [7 x 5]
Groups: plot.ID, species.name, layer.new, new.cov
```

plot.ID	species.name	region	layer.new	new.cov
1	P1	Spec1 Region1	new_layer_veggie	0.616
2	P1	Spec2 Region1	6	0.100
3	P1	Spec3 Region1	6	0.800
4	P2	Spec1 Region2	new_layer_veggie	0.760
5	P2	Spec5 Region2	2	0.400
6	P2	Spec4 Region2	3	0.700
7	P2	Spec6 Region2	new_layer_veggie	0.712

Cover values will be merged under the assumption of no overlap, when we specify 'method' = sum:

```
merge_cov(veg.df, cover = "abund", plot = "plot.ID", taxacol = "species.name",
          layercol = "layer", method = "sum")
```

Source: local data frame [7 x 5]
 Groups: plot.ID, species.name, layer.new, new.cov

plot.ID	species.name	region	layer.new	new.cov
1	P1	Spec1 Region1	new_layer_veggie	0.8
2	P1	Spec2 Region1	6	0.1
3	P1	Spec3 Region1	6	0.8
4	P2	Spec1 Region2	new_layer_veggie	1.0
5	P2	Spec5 Region2	2	0.4
6	P2	Spec4 Region2	3	0.7
7	P2	Spec6 Region2	new_layer_veggie	0.9

The function can also merge cover values for specific layers if you supply the target layers in the argument 'layer names', e.g. for layers "4"and "5":

```
merge_cov(veg.df, plot = "plot.ID", taxacol = "species.name",
          layercol = "layer", layernames = c("4","5"), cover = "abund")
```

Source: local data frame [8 x 5]
 Groups: plot.ID, species.name, layer.new, new.cov

plot.ID	species.name	region	layer.new	new.cov
1	P1	Spec1 Region1	6	0.400
2	P1	Spec1 Region1	new_layer_veggie	0.360
3	P1	Spec2 Region1	6	0.100
4	P1	Spec3 Region1	6	0.800
5	P2	Spec1 Region2	new_layer_veggie	0.760
6	P2	Spec5 Region2	2	0.400
7	P2	Spec4 Region2	3	0.700
8	P2	Spec6 Region2	new_layer_veggie	0.712

You can customize the output even further, by narrowing the operation to specific species and layers. In the example below, we want to merge cover values only across layer "4"and "5"for "Spec1"within plots:

```
merge_cov(veg.df, plot = "plot.ID", taxacol = "species.name", taxonname = "Spec1",
          layercol = "layer", layernames = c("4", "5"), cover = "abund")
```

Source: local data frame [10 x 5]

plot.ID	species.name	region	layer.new	new.cov
1	P1	Spec1 Region1	6	0.40
2	P1	Spec1 Region1	new_layer_veggie	0.36

3	P1	Spec2 Region1		6	0.10
4	P1	Spec3 Region1		6	0.80
5	P2	Spec1 Region2	new_layer_veggie		0.76
6	P2	Spec5 Region2		2	0.40
7	P2	Spec4 Region2		3	0.70
8	P2	Spec6 Region2		4	0.10
9	P2	Spec6 Region2		5	0.60
10	P2	Spec6 Region2		5	0.20

References

Fischer, H.S. 2014. On the combination of species cover values from different vegetation layers. *Applied Vegetation Science* 18: 169-170.

Tichý, L. & Holt, J. 2006. *JUICE program for management, analysis and classification of ecological data*. Program manual. Masaryk University Brno, Czech Republic.

16 Data summary per group

A common step in the preparation of vegetation tables are summaries across groups. Examples include the calculation of number of plots per region, number of species per plot, mean indicator values per vegetation type. In **dplyr**, summaries per group are carried out by first specifying the grouping structure with **group_by()** and then, calculating a function for groups with **summarise()**. Both functions can be piped together.

In contrast to the **mutate()** function, which does not change the number of rows in the table, **summarise()** collapses the table, so that each row corresponds a group.

The next two sections include examples for summary-by-group operations for vegetation data. They are based on the following table, with 'species': species name, 'layer': vegetation layer, plot: plot ID, perc: abundance in percentage, region: region, in which the plot was sampled.

```
data <- data.frame(species = c("Sp1", "Sp1", "Sp2", "Sp3",
                             "Sp1", "Sp3", "Sp3", "Sp4",
                             "Sp1", "Sp4"),
                  layer = c(5, 6, 6, 6,
                             6, 4, 6, 6,
                             6, 6),
                  plot = c(rep("P1", 4), rep("P2", 4), rep("P3", 2)),
                  perc = c(20, 50, 1, 5,
                          3, 15, 20, 5,
                          50, 10),
                  region = c(rep("A", 8), rep("B", 2)))
```

```
data
  species layer plot perc region
1     Sp1     5   P1   20      A
2     Sp1     6   P1   50      A
3     Sp2     6   P1    1      A
4     Sp3     6   P1    5      A
5     Sp1     6   P2    3      A
6     Sp3     4   P2   15      A
7     Sp3     6   P2   20      A
8     Sp4     6   P2    5      A
9     Sp1     6   P3   50      B
10    Sp4     6   P3   10      B
```

16.1 Number of species per plot

The **n_distinct()** wrapper counts the number of unique species names per plot (not their replications!). Providing a name for the new column is not obligatory but recommend. This can save some trouble later, if we want to reuse the new column.

```
library(dplyr)

data %>%
  # specify the grouping structure
  group_by(plot) %>%
```

```
# carry out the summary function and name new column
summarise(nr.spec = n_distinct(species))
```

In our next example, we want to obtain mean numbers of species across two nested grouping variables: plot and region.

First, we create a grouping that includes plot x region combinations, with plots nested in regions. Then, we calculate again the number of species per plots and name the new column. Next, we group again, this time, at the higher level of region. Finally, we calculate the mean of our previously generated variable 'nr'.

```
data %>%
  group_by(plot, region) %>%
  summarise(nr = n_distinct(species)) %>%
  group_by(region) %>%
  summarise(mean.sp.nr = mean(nr))
```