

Module 5 (File processing)

Contents

| | | |
|----------|---|-----------|
| 1 | Operations on files and directories | 2 |
| 1.1 | Introduction | 2 |
| 1.2 | Paths | 2 |
| 1.2.1 | Absolute paths | 2 |
| 1.2.2 | Relative paths | 4 |
| 1.3 | Information on files and directories | 4 |
| 1.4 | Basic file system operations | 6 |
| 1.5 | Listing directory contents | 6 |
| 1.6 | Summary | 6 |
| 1.7 | Bibliography | 6 |
| 2 | Text files and connections | 7 |
| 2.1 | Introduction | 7 |
| 2.2 | Reading and writing text files | 7 |
| 2.3 | Connections | 9 |
| 2.4 | Summary | 12 |
| 2.5 | Bibliography | 12 |
| 3 | Common file formats | 12 |
| 3.1 | Introduction | 12 |
| 3.2 | Serializing and deserializing R objects | 12 |
| 3.3 | Handling tabular data | 13 |
| 3.4 | Data exchange over the Web | 14 |
| 3.5 | Summary | 15 |
| 3.6 | Bibliography | 15 |
| 4 | Markdown and knitr | 15 |
| 4.1 | Introduction | 15 |
| 4.2 | Markdown | 16 |
| 4.2.1 | Markdown syntax | 16 |
| 4.3 | knitr | 17 |
| 4.4 | Summary | 19 |
| 4.5 | Bibliography | 19 |

1 Operations on files and directories

1.1 Introduction

In order to perform data analysis we have to... possess some data to analyze. Explaining how to get such data is not within the scope of this course. So just let us assume that we have a data set. Most likely, it is stored:

- as a disk file (perhaps a text file),
- somewhere over the internet (JSON, XML, HTML, YAML,,),
- in a database (maybe an SQL one).

Unfortunately, we do not have time to cover database access issues. The readers who already know some SQL will certainly be interested in the following R packages:

- RSQLite
- RPostgreSQL,
- RMySQL,
- ROracle,
- RJDBC.

The DBI package provides a common abstract interface to all the supported SQL databases. More details on that can be found in: R Development Core Team, *R Data Import/Export Manual*.

In this part we will thus be interested in accessing disk files and internet resources. Moreover, we will focus mainly on files that fit into available RAM (in-memory data). However, we will also discuss how to read only a chunk of a file at a time – that may help with large files. More advanced solutions see e.g.: the rhdfs (Hadoop HDFS access), bigmemory, MonetDB.R packages.

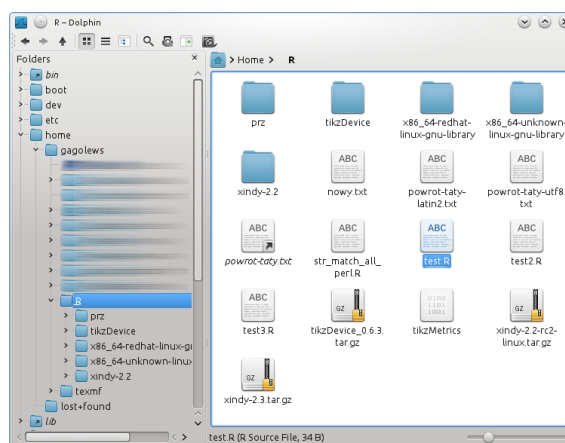
1.2 Paths

A **path** specifies a unique location in a file system. It is used to point to a specific file or directory.

1.2.1 Absolute paths

An **absolute path** is valid in any context. Note that in Unix/Linux/OS X “/” denotes a directory separator. For example, `/home/gagolews/R/test.R` denotes a file named `test.R` in:

```
/(root dir)
home/(home dir – all users)
gagolews/(my home dir)
R/(R dir in my home dir)
```



The UNIX file system forms a connected hierarchy. Various devices are mounted as branches of the file system's tree. For example, on my laptop, `/home/gagolews/` resides on a separate encrypted partition. On the other hand, on my other PC, `/home5/` is an NFS network share. What is more, most paths are **case-sensitive**.

Here are some of my platform's settings:

```
.Platform$file.sep
## [1] "/"
```

```
.Platform$OS.type
## [1] "unix"
```

See also: `?Sys.info`.

On the other hand, Windows uses “\” for a directory separator. For instance, `C:\Users\gagolews\R\test.R` denotes a file named `test.R` on the `C:` drive (possibly some hard disk partition) in the `\Users\gagolews\R` directory. Here, most paths are **case-insensitive**.

Luckily, R accepts “/” as directory separators on Windows. Thus, for portability reasons, this is the separator of choice.

The `normalizePath()` function indicates which paths R treats as valid ones. On Linux:

```
normalizePath("/home/gagolews/R/test.R")
## [1] "/home/gagolews/R/test.R"
normalizePath("/home/gagolews/R/nosuchfile.R")
## Warning: path[1]="/home/gagolews/R/nosuchfile.R": No such file or directory
## [1] "/home/gagolews/R/nosuchfile.R"
normalizePath("/home/////gagolews/R////test.R") # OK
## [1] "/home/gagolews/R/test.R"
normalizePath("\\home\\gagolews\\R\\test.R") # wrong
## Warning: path[1]="\\home\\gagolews\\R\\test.R": No such file or directory
## [1] "\\home\\gagolews\\R\\test.R"
```

On Windows:

```
normalizePath("C:\\Users\\gagolews\\R\\test.R")
## [1] "C:\\Users\\gagolews\\R\\test.R"
normalizePath("C:/Users/gagolews/R/test.R")
## [1] "C:\\Users\\gagolews\\R\\test.R"
```

By the way, there is no need to call `normalizePath()` in R scripts explicitly. Paths are normalized automatically.

“~” points to the home directory.

```
normalizePath("~")
## [1] "/home/gagolews"
normalizePath("~/R/test.R")
## [1] "/home/gagolews/R/test.R"
```

For example, on my Windows “~” points to `C:\Users\gagolews`.

Given a file path, `basename()` extracts the file’s name and `dirname()` extracts the directory’s name.

```
basename("~/R/test.R")
## [1] "test.R"
dirname("~/R/test.R")
## [1] "/home/gagolews/R"
```

These are quite simple string operations. They do not check if a file exists.

```
basename("~/R/nosuchfile.R")
## [1] "nosuchfile.R"
file.exists("~/R/nosuchfile.R")
## [1] FALSE
```

The `file.path()` function may be used to construct a file path in a portable way.

```
file.path("~", "R", "test.R")
## [1] "~/R/test.R"
file.path("/home/gagolews/R", "test.R")
## [1] "/home/gagolews/R/test.R"
file.path("/home/gagolews/R/", "test.R") # // is OK, see above
## [1] "/home/gagolews/R//test.R"
```

You should never use `paste()` with `sep=""` to create file paths.

Moreover, in interactive R sessions the `?file.choose` functions may be used to ask the user for a file path.

The `tempdir()` function gives the directory in which current R session stores temporary files:

```
tempdir()
## [1] "/tmp/Rtmp0xMAMX"
```

`tempfile()` generates a random file name which may be used to store temporary data:

```
tempfile()
## [1] "/tmp/Rtmp0xMAMX/filee5b4c401fc"
tempfile()
## [1] "/tmp/Rtmp0xMAMX/filee5b5d435b1"
```

Note that the contents of the temporary directory will most likely be vanished after quitting current R session.

1.2.2 Relative paths

Relative paths are context-sensitive. They are relative to the **current working directory**:

```
getwd()
## [1] "/home/gagolews/Dydaktyka/adasdr-ipi/textbooks"
```

`setwd()` may be used to change the current working directory:

```
normalizePath("test.R")
## Warning: path[1]="test.R": No such file or directory
## [1] "test.R"
setwd("~/R/")
normalizePath("test.R")
## [1] "/home/gagolews/R/test.R"
```

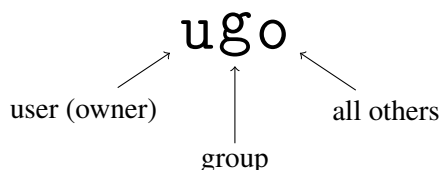
See also *Session → Set Working Directory* in RStudio. However, you should generally avoid changing the current working directory in your R scripts, for i.a. portability reasons.

“.” points to the parent directory of the current working directory and “..” points to the current working directory itself.

```
setwd("~/R/")
normalizePath("test.R")
## [1] "/home/gagolews/R/test.R"
normalizePath("../test.R")
## [1] "/home/gagolews/R/test.R"
normalizePath("../")
## [1] "/home/gagolews"
normalizePath("../R/../../gagolews/R/test.R")
## [1] "/home/gagolews/R/test.R"
```

1.3 Information on files and directories

Let us discuss the file and directory access permissions issues. Here is how we represent file and directory **modes** on UNIX-like systems:



Each of `u`, `g`, `o` $\in \{0, 1, \dots, 7\}$ (octal digits). Each octal digit may be represented as $4r + 2w + 1e$; $r, w, e \in \{0, 1\}$.

In case of file modes:

- *r* – has read access
- *w* – has write access (e.g. may modify contents)
- *e* – has execute access

On the other hand, in case of directory modes:

- *r* – can read contents
- *w* – can add new/remove contents
- *e* – can access contents (if path is known)

For example, consider a user *maciek* who is a member of user group *students*. He owns a directory *dir/* and a file *dir/test.txt*. Additionally, *dir/*'s and *dir/test.txt*'s group is set to *students*. *dir* mode is set to 751 and *dir/test.txt* mode to 644. In such a scenario, *maciek* can e.g. add new files to *dir/* and view *dir/*'s listing. He can also read and modify *dir/test.txt*.

All *students* may see what is out there in *dir/*. They can also read *dir/test.txt*. All *tutors* and *maciek*'s girlfriend (member of *accountants*) cannot see *dir/*'s contents. But if they know that *dir/* includes *test.txt*, they will be able to read it.

Note the meaning of a file's execute permissions. Let *~/test.R* be a file with such a permission (call e.g. `chmod u+x ~/test.R` in the terminal). The contents of the file are as follows:

```
#!/usr/bin/Rscript --vanilla
cat("2 + 2 = ", 2+2, "\n")
```

In such a situation, we may run the script directly in the terminal:

```
[gagolews@eurydike ~]$ ./test.R
2 + 2 = 4
```

This is R in the batch mode.

To test file access permissions in R we call the `file.access()` function. For example, the *~/R/test.R* file has mode of 664. The current user is the file's owner:

```
file.access("~/R/test.R") # does it exist?
## ~/R/test.R
## 0
file.access("~/R/test.R", 1) # e?
## ~/R/test.R
## -1
file.access("~/R/test.R", 2) # w?
## ~/R/test.R
## 0
file.access("~/R/test.R", 4) # r?
## ~/R/test.R
## 0
```

By the way, most often we do not test for file access permissions. An R function will just return an error if it encounters any problems; cf. `tryCatch()`.

On the other hand, `file.info()` returns a data frame with basic file information:

```
file.info("~/R/test.R")
##           size isdir mode          mtime          ctime
## ~/R/test.R   34 FALSE  664 2012-12-22 21:46:57 2014-07-24 12:32:46
##           atime uid gid    uname grname
## ~/R/test.R 2014-07-31 12:52:34 1000 100 gagolews users
```

Compare with the result of executing `ls -l` in the terminal:

```
[gagolews@eurydike R]$ ls -l test.R
-rw-rw-r--. 1 gagolews users 34 2012-12-22 test.R
```

1.4 Basic file system operations

The `file.create()` function creates a new, empty file (if it exists, its contents will be vanished). `dir.create()` creates a new directory. `file.remove()` removes a file. `unlink()` removes a directory and all its contents. `file.copy()` copies a file. `file.rename()` renames a file. See also the `download.file()` function.


 **Exercise:** Let `urls` be a character vector with n URLs. Write a function `massDownload()` to download the files located at given `urls`.

The files should be saved as `1.html`, `...`, `n.html` in a given `outdir`.


1.5 Listing directory contents

`list.files()` returns a list of files in a given directory, represented as a character vector.

```
list.files("~/R", pattern = "^test.*\\.R$")
## [1] "test.R" "test2.R" "test3.R"
list.files("~/R", pattern = glob2rx("test*.R"))
## [1] "test.R" "test2.R" "test3.R"
list.files("~/R", pattern = glob2rx("test*.R"), full.names = TRUE)
## [1] "/home/gagolews/R/test.R" "/home/gagolews/R/test2.R"
## [3] "/home/gagolews/R/test3.R"
list.files("~/R/prz", recursive = TRUE)
## [1] "przA/przA2/test4.R" "t.test2.txt" "test1.R"
## [4] "test2.R" "test3.R" "test5.txt"
```

 **Exercise:** Write a function `mergeAll()` to merge all files with extension `ext` (defaults to `.txt`) located in a given directory `dir`. The output file should be saved as `outfname`.


Hint: use `file.copy()` and `file.append()`


 **Exercise:** Determine total size of all files in e.g. `/pub/music`.

A solution:

```
fnames <- list.files("/pub/music", recursive=TRUE, full.names=TRUE)
sum(file.info(fnames)$size)/1e9 # [GB]
## [1] 22.82181
```

By the way, 1 kB (a kilobyte) is equal to 1000 B. On the other hand, 1 KiB (a kibibyte) is equal to 1024 B.

 **Exercise:** Write a function to find 10 oldest files in a given directory (scan its subdirectories recursively).

 **Exercise:** You are running out of free disk space. Write a function to suggest which files should be removed (or moved to another drive) in order to gain about `x` (defaults to 10) gigabytes of mass storage.

1.6 Summary

Relative paths reference to the current working directory.

“/” is a portable directory separator.

A file's `execute` permission enables us to run R scripts in batch mode.

1.7 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 14
- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 10
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 10

2 Text files and connections

2.1 Introduction

Most often, data are stored in text files. Here are some popular file formats:

- R, C++, Java, LaTeX, ... source files
- HTML web pages, XML (e.g. OpenOffice Writer and Calc documents)
- JSON, YAML
- PDF, PostScript (w/o bitmap images)
- ...

By the way, sometimes such files are additionally compressed, e.g. with zip (Lempel-Ziv-based methods).

In this section we will discuss how to deal with text files in R.

2.2 Reading and writing text files

If a text file is of reasonable size (say, < 100 MB), we may just read its contents into RAM and process it as a character vector/string.


Let us read an exemplary text file:

```
testfile <- readLines("test-utf8.txt") # relative path
head(testfile, 9)
## [1] "\"Pójdźcie, o dzieci, pójdźcie wszystkie razem"
## [2] "Za miasto, pod słup na wzgórek,"
## [3] "Tam przed cudownym klękajcie obrazem,"
## [4] "Pobożnie mówcie paciorkom."
## [5] ""
## [6] "Tato nie wraca; ranki i wieczory"
## [7] "We łzach go czekam i trwam;"
## [8] "Rozlały rzeki, pełne zwierza bory"
## [9] "I pełno zbójców na drodze\"."
```

Here, each text file line is read into a separate string. By the way, each file should end with a newline character. Otherwise, a warning will be given.

Basic statistics:

```
length(testfile) # number of text lines
## [1] 104
library("stringi")
sum(!stri_isempty(stri_trim(testfile))) # non-empty lines
## [1] 84
sum(!stri_isempty(stri_trim(testfile)))/4 # poem's verses
## [1] 21
stri_stats_general(testfile)
##      Lines LinesNEmpty      Chars CharsNWhite
##      104          84      2689      2326
```

 **Exercise:** Write a function `words()` that reads the contents of a given text file, `fname`.

The function returns a data frame with 2 columns:

- `word` – all the unique words occurring in the text file,
- `count` – total number of occurrences of each corresponding word.

The data frame should be sorted from the most to the least common word.

Writing a text file:

```
test <- c("first line", "second line")
fname <- tempfile()
writeLines(test, fname)
```

The resulting file's contents are as follows:

```
first line
second line
```

Here is how we may append lines to a text file:

```
cat(c("third", "fourth"), sep = "\n", file = fname, append = TRUE)
```

Note that if a file does not exist, it will be created from scratch. The above results in:

```
first line
second line
third
fourth
```

Among other functions useful for preparing an output file we find e.g. `format()`, `sprintf()`, `stri_wrap()`, `stri_pad()`. Interestingly, e.g. a spell check may be done (requires `aspell` installed):

```
errors <- aspell("test-utf8.txt", control = list("-l pl"))
errors$Original # unrecognized words
## [1] "paciórek" "paciórek" "Zdrowaś" "litanią" "tarkot" "radośne"
## [7] "rozynki" "moję" "paciórek"
errors$Suggestions[[1]] # suggestions for 'paciórek'
## [1] "Paciorek" "paciorek" "maciorek" "paciorki" "piórek" "pachówek"
## [7] "panierek" "papierek" "zacierek" "łaciarek" "maciorka" "piórko"
## [13] "panierka" "zacierka" "pachówka" "łaciarka"
```

There are at least 2 possible portability issues connected with reading text files in R.

The first one concerns a file's encoding. Recall that a text file (as well as any string) is just a sequence of bytes. `readLines()` does not re-encode the contents of an input file. The encoding argument only affects encoding marks of output strings (an R string may be marked as e.g. unknown (native), latin1, or UTF-8). In other words, setting `encoding="UTF-8"` or `"latin1"` enables us to read an UTF-8- or latin1-encoding file on any platform. However, we will not be able (at least directly) to load a latin2-file on a system with `native=cp1250` or UTF-8.

Possible solutions:

- read as a binary file, re-encode manually:

```
fname <- "test-latin2.txt"
fsize <- file.info(fname)$size
f <- readBin(fname, what = "raw", size = 1, n = fsize)
f <- stri_encode(f, "latin2", "")
f <- stri_split_regex(f, "\\r\\n|\\r|\\n")[[1]]
f[1]
## [1] "\"Pójdźcie, o dziatki, pójdźcie wszystkie razem"
```

This also enables the use of `stri_enc_detect()`.

- use connections (`file(fname, encoding="...")`), see below.

The second potential portability issue is connected with end-of-line marking.

- UNIX-like systems use “`\n`” to denote an EOL,
- Windows systems use “`\r\n`”,
- MacOS (v ≤ 9) used to use “`\r`”.

Luckily, `readLines()` understands both:

```
fname <- tempfile()
cat("line1\r\nline2\r\n", file = fname) # another way to write
readLines(fname)
## [1] "line1" "line2"
```


2.3 Connections

A call to:

```
readLines("test-utf8.txt")
```

does in fact the following:

1. create a new file connection,
2. open the connection for reading,
3. read from the connection,
4. close the connection.

We may perform the above manually:

```
con <- file("test-utf8.txt") # create a file connection
class(con)
## [1] "file"          "connection"
typeof(con)
## [1] "integer"
unclass(con)
## [1] 5
## attr(,"conn_id")
## <pointer: 0xa6>
con
##      description      class      mode      text
## "test-utf8.txt"      "file"      "r"      "text"
##      opened      can read      can write
##      "closed"      "yes"      "yes"
```

And now:

```
open(con, "r") # open for reading
f <- readLines(con)
close(con) # close
f[1]
## [1] "\"Pójdźcie, o dziatki, pójdźcie wszystkie razem"
```

Cf. `print(readLines)`.

Such a manual approach enables us to re-encode the input:

```
con <- file("test-latin2.txt", open = "r", encoding = "latin2")
f <- readLines(con)
close(con)
f[1]
## [1] "\"Pójdźcie, o dziatki, pójdźcie wszystkie razem"
```

Open modes include:

- "r" – open for reading only
- "w" – open for writing only; if a file exists, its contents will be **erased**; otherwise, it will be created from scratch
- "a" – open for appending only; if a file does not exist, it will be created

Three connections are always open:

- `stdin` – standard input (read-only, default=keyboard),
- `stdout` – standard output (write-only, default=R console),
- `stderr` – error output (write-only, default=R console).

```
cat("test\n", file = stdout()) # stdout() is default anyway
## test
cat("test\n", file = stderr())
## test
showConnections(TRUE)[1:3, ]
##      description class      mode text      isopen      can read can write
```

```
## 0 "stdin"      "terminal" "r"  "text" "opened" "yes"  "no"
## 1 "stdout"     "terminal" "w"  "text" "opened" "no"   "yes"
## 2 "stderr"     "terminal" "w"  "text" "opened" "no"   "yes"
getConnection(2)
## description      class      mode      text      opened    can read
## "stderr"         "terminal" "w"       "text"    "opened"  "no"
## can write
## "yes"
```

By the way, maximal number of connections is equal to ca. 128.

Note that e.g. `stdout` may be redirected to a text file:


```
sink("/tmp/output.txt")
print(1:10)
sink() # revert changes
readLines("/tmp/output.txt")
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

An `url` connection enables us to read from a network share.

```
con <- url("http://www.R-project.org", open = "r")
f <- readLines(con)
close(con)
f[1:7]
## [1] "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" \"http://www.w3.org/TR/html4/frameset.dtd\">"
## [2] "<html>"
## [3] "<head>"
## [4] "<title>The R Project for Statistical Computing</title>"
## [5] "<link rel=\"icon\" href=\"favicon.ico\" type=\"image/x-icon\">"
## [6] "<link rel=\"shortcut icon\" href=\"favicon.ico\" type=\"image/x-icon\">"
## [7] "<link rel=\"stylesheet\" type=\"text/css\" href=\"R.css\">"
```

In fact, `readLines()`'s first argument understands URLs directly.

 **Exercise:** `cran.rstudio.com/src/base/R-2/` is an example of a directory listing generated by the Apache web server (a HTML document). Write a function `getApacheDirListing()`, which downloads a given directory listing (`URL` arg), parses its contents, and returns a data frame with the following columns:

- `url` – an URL of each file,
- `name` – files' names,
- `modtime` – last modification times (POSIXct),
- `size` – approximate sizes in bytes.

By the way, I discouraged you from parsing HTML files with regular expressions. This time, luckily, the data are quite “regular”.

We can also create text connections:

```
con <- textConnection("outobj", open = "w")
cat("testing testing testing", file = con)
close(con)
outobj
## [1] "testing testing testing"
```

Moreover, we can also read/write gzip or bzip2 compressed files, see `?gzfile`.


What is important, we do not have to read all data from a connection:

```
writeLines(strrep("line ", 1:5), "/tmp/test.txt")
con <- file("/tmp/test.txt", open = "r")
readLines(con, n = 1)
## [1] "line 1"
```

```
readLines(con, n = 2)
## [1] "line 2" "line 3"
close(con)
```

An example: printing a file's contents on the screen, line by line.


```
writeLines(str_i_paste("line ", 1:5), "/tmp/test.txt")
con <- file("/tmp/test.txt", open="r")
while (length(x <- readLines(con, n=1)) > 0)
  cat(x, "\n", sep="")
## line 1
## line 2
## line 3
## line 4
## line 5
close(con)
```

 **Exercise:** Extract a webpage's (e.g. www.R-project.org) title.

```
...
<title>here's the title</title>
...
```

Here is a solution:

```
con <- url("http://www.R-project.org", open="r")
while (length(x <- readLines(con, n=1)) > 0) {
  t <- str_i_extract_first_regex(x,
    "(?<=<title>)(.)(?<=</title>)"
  )
  if (!is.na(t[1])) break
}
close(con)
t
## [1] "The R Project for Statistical Computing"
```

 **Exercise:** Write a function `extractUrls()` to extract all URLs from a given HTML file on the internet. Return a character vector.

The push-back buffer is useful if you have read too many lines incidentally.

```
writeLines(str_i_paste("line ", 1:5), "/tmp/test.txt")
con <- file("/tmp/test.txt", open = "r")
readLines(con, n = 1)
## [1] "line 1"
pushBack("lineX", con)
readLines(con, n = 1)
## [1] "lineX"
readLines(con, n = 1)
## [1] "line 2"
close(con)
```

Connections also enable us to write only a portion of a file at a time:

```
writeLines(str_i_rand_strings(6, 10, "[a-z]"), "/tmp/testI.txt")
readLines("/tmp/testI.txt")
## [1] "aexslabzg" "bxxhgfkryz" "onqecgpqls" "ownqkyjhzi" "rpdunjeta"
## [6] "eyydsudkp"
conI <- file("/tmp/testI.txt", open="r")
conO <- file("/tmp/testO.txt", open="w")
while (length(x <- readLines(conI, n=3)) > 0)
  writeLines(str_i_trans_toupper(x), conO)
close(conI)
```

```
close(con0)
readLines("/tmp/test0.txt")
## [1] "AEIXSLABZG" "BXXHGFKRZY" "ONQECGPQLS" "OWNQKYJHZI" "RPMJUNJETA"
## [6] "EYYYSUDKP"
```

2.4 Summary

Relatively small files may be read into RAM, and then processed as ordinary character vectors. For larger files, use text connections.

2.5 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 7
- R Core Team, *R data import/export*, 2014
- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 10
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 10

3 Common file formats

3.1 Introduction

Now it is time to discuss popular file formats that you are quite likely to encounter while playing with R:

- RDS – R data serialization (for single R objects)
- CSV – text representation of tabular data
- JSON, XML, YAML – data exchange over the Web

3.2 Serializing and deserializing R objects

The easiest and fastest way to store a single R object for future reference is to use the `saveRDS()` function.

```
fname <- tempfile()
saveRDS(chickwts, fname)
```

By default, the output file is compressed.

```
file.info(fname)$size
## [1] 392
object.size(chickwts)
## 2352 bytes
```

The object may be deserialized with the `readRDS()` function.

```
obj <- readRDS(fname)
identical(chickwts, obj)
## [1] TRUE
```

A set of R objects may be serialized by calling `save()`. The `load()` function gives a reverse operation.

The `save.image()` function may be called to create an image of the “user’s workspace”. It is equivalent to a call to `save(list=ls(all=TRUE), file=".RData")`. Unfortunately, non-R users will not understand these file formats.

Some R users may be interested in a function that creates a textual representation of a given object:

```
test <- c(a = 1, b = 2, c = 3)
dput(test)
## structure(c(1, 2, 3), .Names = c("a", "b", "c"))
```

Its output may simply be copy-pasted to the R console.

3.3 Handling tabular data

Tabular data (represented in R as data frames) are perhaps most often used in data analysis tasks. Excel, SPSS, Statistical, Statgraphics, Stata, R, ... – all of them use different native formats. But all of them may read and write **CSV files**. CSV, at least originally stands for “comma separated-values”. It is a plain text file formatting scheme.

Here are the contents of the `mexico_soccer_managers.csv` file.

```
## "Nationality";"Name";"Career";"Won";"Drawn";"Lost"
## "Mexico";"Adolfo Frías Beltrán";1923;4;1;1
## "Mexico";"Alfonso Rojo de la Vega";1928;0;2;0
## "Spain";"Juan Luque de Serrallonga";1930;0;3;0
## ...
```

Here, 1st row gives column names. Note that we have semicolon-separated fields.

The `read.table()` function may read such kind of tabular data. This function has many variants: `read.csv()`, `read.csv2()`, `read.delim()`, `read.delim2()`, see `?read.table`. Each of them calls `read.table()`. The only difference is within the list of default arguments:

- `header` – does 1st row consist of column names?
- `sep` – column separator
- `dec` – dot or comma used for fractional parts of numbers?
- `comment.char` – which character starts a comment line?


On the other hand, `write.table()` (and related functions) save a given data frame. By the way, I recommend setting `row.names=FALSE` here.


```
head(read.table("mexico_soccer_managers.csv",
  header=TRUE, sep=";"))
##   Nationality      Name      Career Won Drawn
## 1    Mexico Adolfo Frías Beltrán    1923   4     1
## 2    Mexico Alfonso Rojo de la Vega    1928   0     2
## 3     Spain Juan Luque de Serrallonga    1930   0     3
## 4    Mexico Rafael Garza Gutiérrez 1934, 1937-1938, 1949  14     1
## 5   England Alfred C. Crowle      1935   5     0
## 6   Hungary Jorge Orth      1947   2     0
## Lost
## 1    1
## 2    0
## 3    0
## 4    1
## 5    0
## 6    0
```

`read.table()` may be slow for large CSV files. It is because it calls `scan()` and then `type.convert()` to guess column types.

An example:

```
n <- 1000000
test <- data.frame(
  id=str_i_rand_strings(n, 10),
  vals1=runif(n),
  vals2=rnorm(n),
  class=sample(letters, n, replace=TRUE)
)
fname <- tempfile()
write.csv(test, fname, row.names=FALSE)
read.csv(fname) # slow...
```

 **Exercise:** Play with the `colClasses` argument to gain 10x+ speedup.

 **Exercise:** Are you able to open this file in Excel/Calc?

 **Exercise:** What if $n=10000000000$? Calculate mean and variance of `vals1` and `vals2`.

 **Exercise:** Download all data from `cran-logs.rstudio.com/`. What are the top 10 most often downloaded R packages in each month? What are the top 50 most often downloaded R packages last week?

3.4 Data exchange over the Web

Here are some file formats that are tailored for data exchange over the Web:

- **JSON** (packages `rjson`, `jsonlite`, `RJSONIO`)
- **XML** (package `XML`)
- **YAML** (package `yaml`)


An example of a JSON file:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21_2nd_Street",
    "city": "New_York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    { "type": "home", "number": "212_555-1234" },
    { "type": "office", "number": "646_555-4567" }
  ]
}
```

An example of an XML file:

```
<?xml version="1.0"?>
<quiz>
  <qanda seq="1">
    <question>
      Who was the forty-second president
      of the U.S.A.?
    </question>
    <answer>
      William Jefferson Clinton
    </answer>
  </qanda>
</quiz>
```

By the way, HTML is tightly related to XML.

 **Exercise:** Write a function `removeHtmlTags()` to create a text file named `outfile`, which results from removing all HTML tags from a HTML file named `infile`. We should only be interested in data between `<body>` and `</body>`. Do not use regular expressions here. Play with the `XML` package.

An example:

```
<html><head><title>My homepage</title></head>
<body><h1>Hello everybody</h1>
<p>I'm really <em>glad</em> you popped by.</p>
</body></html>
```

```
Hello everybody
I'm really glad you popped by.
```

 **Exercise:** Read the following feeds to data frames:

```
finance.yahoo.com/webservice/v1/symbols/allcurrencies/quote?format=xml
finance.yahoo.com/webservice/v1/symbols/allcurrencies/quote?format=json
```

Here is a YAML file:

```
-----
receipt:      Oz-Ware Purchase Invoice
date:         2012-08-06
customer:
  given:      Dorothy
  family:     Gale

items:
  - part_no:   A4786
    descrip:   Water Bucket (Filled)
    price:     1.47
    quantity:  4
```

3.5 Summary

We have discussed the following file formats:

- RDS – R data serialization (for single R objects)
- CSV – text representation of tabular data
- JSON, XML, YAML – data exchange over the Web

3.6 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 7
- R Core Team, *R data import/export*, 2014
- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 10
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 10

4 Markdown and knitr

4.1 Introduction

*Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on **explaining to humans what we want the computer to do.***

[Donald E. Knuth, *Literate Programming*, 1984]

In order to write reproducible reports, we will need to know some **markup language**. Examples of such languages include:

- T_EX
- HTML
- Markdown

All of them may store an annotated document in a plain text file. Markdown is the simplest and most portable.

We will need the knitr package. Moreover, set up RStudio to use Tools → Global Options → Sweave Weave Rnw files with...knitr

4.2 Markdown

Markdown allows you to write using an easy-to-read, easy-to-write plain text format. (...)

The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. (...) The single biggest source of inspiration for Markdown's syntax is the format of plain text email.

[daringfireball.net/projects/markdown/]

A Markdown document is a plain text file that is readable as-is. It can easily be converted to HTML (and then the output may be copy-pasted to Word/Writer). In this tutorial, we will discuss the R Markdown flavor.

Create an .md file in RStudio. Select File → New File → R Markdown; Default output format: HTML. Press (CTRL+SHIFT+Y) or (CTRL+SHIFT+K) to preview.

4.2.1 Markdown syntax

Sectioning:

```
Title (H1)
=====

Subtitle (H2)
-----
```

or:

```
# Title (H1) #
## Subtitle (H2) ##
### Subsubtitle (H3) ###
```

and so on (until H6). By the way, save the file using UTF-8.

Text input and basic formatting:

```
This is the first paragraph. A newline
is just like any other whitespace here.

And now for something completely different:
the second paragraph. italic and bold
or italic and bold.
And now our rising star: \*.
And a strikeout.
```

```
A nice horizontal line appears above
and below.
```

```
*****
```

Block quote:

```
They say:
> You have to learn the rules of the game.
> And then you have to play better than anyone else.
```

Link:

Click [here](<http://www.internetlastpage.com/>).

Enumerated list:

1. Primo
2. Secundo

Itemized list:

- * this
- * that

List nesting:

- * Primo
 - * this
 - * that
- * Secundo

Pre-formatted text (e.g. source codes):

```
Evaluating `x <- 5` gives you...

Let's consider the following code chunk:

```
x <- 5
print(x)
```
```

Alternatively (4 spaces + a newline):

```
Let's consider the following code chunk:

    x <- 5
    print(x)
```


Table:

| First column | Second column |
|--------------|---------------|
| Giovanni | 32 |
| Maciek | 27 |
| Jose | 59 |

Mathematical equation: (T_EX-like syntax):

```
$f(x)=\sqrt{x}$

$$f(x)=\frac{1}{2x}$$
```

 **Exercise:** Write your CV in Markdown. Use `pandoc` to convert it to HTML and PDF (via L^AT_EX).

 **Exercise:** Check out how to prepare Markdown slides in RStudio. Click `File` → `New file` → `R presentation`.

4.3 knitr

knitr allows for dynamic, fast, and reproducible report generation. It is ideal for: preparing data analysis reports and documenting your software. Interestingly, this textbook is prepared in knitr and L^AT_EX.

An `.Rmd` file may include R code chunks. Knitr converts them automatically to a `.md` file (CTRL+SHIFT+H in RStudio).

Here is how we insert an R code chunk:

```
```{r}
some
R
code
```
```

The chunk will be evaluated on a document's knitting.

For example:

```
```{r}
1:5
cat(":-)\n")
```
```

results in:

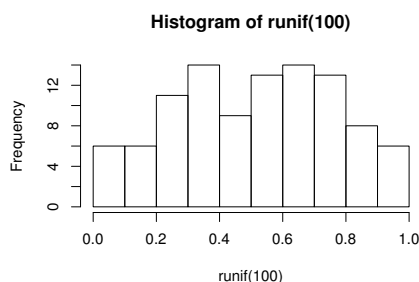
```
1:5
## [1] 1 2 3 4 5
cat(":-)\n")
## :-)
```


Another example:

```
```{r}
hist(runif(100))
```
```

The result:

```
hist(runif(100))
```



 **Exercise:** The `Cars93` data frame from the `MASS` package provides data on 93 car models.

```
library(MASS)
data(Cars93)
```

Conduct an exploratory data analysis of this data set. Generate a knitr report.


Some code chunk options may be set:


```
```{r, chunkName, opt1=val1, opt2=val2}
...
```
```

A few examples:

1. `eval=FALSE` – don't eval contents
2. `echo=FALSE` – don't print code
3. `results="hide"` – don't print results
4. `results="asis"` – allow the results to be interpreted as Markdown code
5. `cache=TRUE` – cache eval results

See yihui.name/knitr/options for more information.

 **Exercise:** Write a function to convert a given data frame to a Markdown table (just `cat()` it to the console). Generate a knitr report in which you demonstrate that the function actually works (using `results="asis"`). Use `cars`, `as.data.frame(WorldPhones)`, `Orange`, `Puromycin`, and `iris` as exemplary data frames.

 **Exercise:** Generate a knitr report summarizing the contents of the data set returned by a call to `available.packages()`. Preferably, you should write the report in HTML (if you already know it; anyway, you can learn it in a few minutes) or \LaTeX . It would be nice e.g. to see some clickable cross-references between the packages' dependencies (`Depends`, `Imports`, `LinkingTo`, `Suggests`, `Enhances` columns).

4.4 Summary

Markdown is a very intuitive and portable markup language. E.g. pandoc may be used to convert a Markdown document to other common file formats. knitr allows for dynamic, fast, and reproducible report generation.

4.5 Bibliography

- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 13
- Xie Y., *Dynamic documents with R and knitr*, CRC press, 2013
- daringfireball.net/projects/markdown/basics
- www.rstudio.com/ide/docs/authoring/using_markdown