# R Data I/O and advanced data structures

Alberto Garfagnini

Università di Padova

AA 2022/2023 - R lecture 5



## Saving in R - (1)

### Saving R objects

- sometimes we need to save object created in R
- to save the current R session, so that it can be loaded at a later stage to continue working on it:

```
save(list = ls(all=TRUE), file = "my-session")
```

- a binary file will be produced and saved on disk
- everything can be loaded, at a later stage, with the following command:

```
load(file= "my-session")
```

### Saving R history

- sometimes we need to save only the lines of code that have been typed in an R session

```
savehistory(file = "my-history.R")
```

- a text file with all the command is saved on disk
- to retrieve history, type:

```
loadhistory(file = "my-history.R")
```

# Saving in R - Save workspace image? [y/n/c]

## Saving the full R workspace

- everytime we quit from an R session (function `q()`), we are asked if want to save the current R workspace

  workspace image ➜ a copy of your current environment
- it includes anything that is user defined, from data frames to functions
- R creates a hidden file called `.RData` in your current working directory
- the file (i.e. the environmnt) is loaded the next time R is started

## To save or not to save ...

- never save the workspace since it's

a) unnecessary

b) leads to irreproducible state and hard to diagnose bugs

- if you need to continue to work in R, just leave the R terminal (or RStudio / Jupyter session) open and restart working once ready



# Saving in R - (2)

## Saving graphics

- graphics can be saved in either pdf or JPEG/PNG to include them in a report
- the procedure is to open a new pdf or JPEG/PNG device, with the `pdf()` or `jpeg()/png()` functions
- then all commands needed to create the graphics can be typed in the R session, and once finished, the device has to be closed with the `dev.off()` function

  Example:

```
pdf("my-plot.pdf")
hist(rnorm(10000))
dev.off()
```

## Saving specific data produced within R

- let's suppose we have produced a vector we want to save on disk

  `nbnumbers <- rnbinom(1000, size=1, mu=1.2)`
- and we want to save them in a file, in a single column

  `write(nbnumbers,"nbnumbers.txt",1)`
- if, instead, we want to save them in a matrix like format

```
xmat <- matrix( rpois(100000, 0.75), nrow=1000 )
write.table(xmat, "table.txt", col.names=F, row.names=F)
```

- we have saved 1000 rows each of 100 Poisson random numbers with $\lambda = 0.75$

# Data Input

- numbers can be inputed through the keyboard, from the Clipboard, from an external file on disk, or from an external file on the Web
- use the concatenate function for up to 10 numbers
- and `scan()` for typing or pasting data into a vector

```
y <- c (6,7,3,4,8,5,6,2)

tu <- scan()
# 1: 6
# 2: 3
# 3: 4
# 4: 2
# 5:
# Read 4 items
tu
# [1] 6 3 4 2
```

- but the easiest way is to read data from a file (or from the Web), already shaped in a data frame format



# Data Input using `read.table()`

- the `read.table()` function reads data from a local file and creates a `data.frame`

```
data <- read.table("yield.txt",header=T)

data
#    year wheat barley oats rye corn
# 1  1980   5.9    4.4  4.1 3.8  4.4
# 2  1981   5.8    4.4  4.3 3.7  4.1
# 3  1982   6.2    4.9  4.4 4.1  4.0
...
# 27 2006   8.0    5.9  6.0 6.1  4.5
# 28 2007   7.2    5.7  5.5 5.7  3.9
# 29 2008   8.3    6.0  5.8 6.1  4.4
```

- the parameter `header = T` tells R to use the first row as column names

```
names(data)
# [1] "year"   "wheat"  "barley" "oats"   "rye"    "corn"

str(data)
# 'data.frame':         29 obs. of  6 variables:
#  $ year  : int  1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 ...
#  $ wheat : num  5.9 5.8 6.2 6.4 7.7 6.3 7 6 6.2 6.7 ...
#  $ barley: num  4.4 4.4 4.9 4.7 5.6 5 5.2 5 4.7 4.9 ...
#  $ oats  : num  4.1 4.3 4.4 4.3 4.9 4.6 5.2 4.6 4.6 4.5 ...
#  $ rye   : num  3.8 3.7 4.1 3.7 4.7 4.6 4.7 4.8 4.6 4.8 ...
#  $ corn  : num  4.4 4.1 4 4.1 4.7 4.3 4.3 4.5 4.2 3.8 ...
```

## read.table(): separators and decimal points

- the default field separator character in read.table() is sep=" ": which identifies with one or more spaces, one or more tabs (\t), and one or more newlines (\n)
- for comma-separated fields use read.csv()
- for semicolon-separated fields use read.csv2()
- for tab-delimited fields with decimal points as a commas, use read.delim2()

```
File: bowens.csv
--------------------
|place,east,north  |
|Abingdon,50,97    |
|Admoor Copse,60,70|
|...               |
|Youlbury,48,3     |
--------------------
```

```
str(bw)
# 'data.frame':   733 obs. of  3 variables:
#  $ place: Factor w/ 727 levels "AERE Harwell",..: 2 3 1 4 5 ...
#  $ east : int  50 60 48 70 59 60 60 59 61 60 ...
#  $ north: int  97 70 87 73 65 65 63 66 63 67 ...
```



## read.csv() and read.delim()

- additional functions to read a file in table format exist

```
?read.table
```

```
...
read.delim(file, header = TRUE, sep = "\t", quote = "\"",
           dec = ".", fill = TRUE, comment.char = "", ...)
...
read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
...
read.csv2(file, header = TRUE, sep = ";", quote = "\"",
          dec = ",", fill = TRUE, comment.char = "", ...)
...
read.delim2(file, header = TRUE, sep = "\t", quote = "\"",
            dec = ",", fill = TRUE, comment.char = "", ...)
```

- further detailed instructions in the 'R Data Import/Export' manual:
  https://cran.r-project.org/doc/manuals/r-release/R-data.html

# Data Input from the Web and from DB

- R can read data form the network using HTTP by specifying the file URL

```
wc <- read.table("https://tinyurl.com/murders-txt", header=T)

str(wc)
# 'data.frame':   50 obs. of  4 variables:
#  $ state      : Factor w/ 50 levels "Alabama","Alaska",..: 1 2 ...
#  $ population: int  3615 365 2212 2110 21198 2541 3100 ...
#  $ murder     : num  15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 ...
#  $ region     : Factor w/ 4 levels "North.Central",..: 3 4 4 ...
```

- several packages available on CRAN to help R communicate with DBMSs: combining a unified 'front-end' package with a 'back-end' module, several common relational databases can be accessed (RMySQL, ROracle, RPostgreSQL and RSQLite)
- finally, R can read binary data files: NASA's HDF5 (Hierarchical Data Format, https://www.hdfgroup.org/HDF5/) and UCAR's netCDF data files (network Common Data Form, http://www.unidata.ucar.edu/software/netcdf/)
- and image files



# tidyverse

- it's an opinionated collection of R packages designed for data science.
- all packages share an underlying design philosophy, grammar, and data structures.
- Web Site: https://www.tidyverse.org/


## R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

# tidyverse packages

| | |
|---|---|
| ggplot2 | gplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics |
| dplyr | it provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges |
| tidyr | it provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable |
| readr | it provides a fast and friendly way to read rectangular data (csv, tsv, and fwf) |
| purrr | it enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors |
| tibble | a modern re-imagining of the data frame |
| stringr | it provides a cohesive set of functions designed to make working with strings |
| forcats | it provides a suite of useful tools that solve common problems with factors |




# readr : https://readr.tidyverse.org/

- it provides a fast and friendly way to read rectangular data: csv, tsv, and fwf
  - read_csv() : read and import comma separated (CSV) files
  - read_tsv() : read and import tab separated (TSV) file
  - read_delim() : read and import general delimited files
  - read_fsw() : read and import fixed width files
  - read_log() : read and import web log files

https://readr.tidyverse.org/


## Alternatives

- in baseR : the read.table() function
- in data.table : the function fread() is similar to read_csv()

# dplyr

https://dplyr.tidyverse.org/

- dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

| function | description | SQL equivalent |
|---|---|---|
| `select()` | select on columns (i.e. variables) | SELECT |
| `filter()` | filter a subset of rows | WHERE |
| `group_by()` | group the data | GROUP BY |
| `summarise()` | reduces multiple values down to a single summary | - |
| `arrange()` | changes the ordering of the rows | ORDER BY |
| `join()` | | JOIN |
| `mutate()` | adds new variables that are functions of existing variables | COLUMN ALIAS |

- all function operate on a `data.frame` / `tibble` and the result is a new `data.frame` / `tibble`

➜ dplyr functions never modify their input



# dplyr : select()

- it allows to select a subrange of columns in the data frame

```
select(flights, year, month, day, dep_time)
```

```
# A tibble: 336,776 x 4
#    year month   day dep_time
#   <int> <int> <int>    <int>
# 1  2013     1     1      517
# 2  2013     1     1      533
```

- usual selection rules apply: we can select a range of columns

```
select(flights, dep_time:dep_delay)
```

```
# A tibble: 336,776 x 3
#   dep_time sched_dep_time dep_delay
#      <int>          <int>     <dbl>
# 1      517            515         2
# 2      533            529         4
```

- we can remove columns with the - (minus) sign

```
select(flights, -(year:day))
```

```
# A tibble: 336,776 x 16
#   dep_time sched_dep_time dep_delay arr_time
#      <int>          <int>     <dbl>    <int>
# 1      517            515         2      830
# 2      533            529         4      850
```

# dplyr : filter()

- it allows to subset observations based on their values
- the first argument is the name of the data frame
- the other arguments are the expressions that filter the data frame

```
filter(flights, month == 1, day == 1)
```

```
#  A tibble: 842 x 19
#    year month   day dep_time sched_dep_time dep_delay arr_time
                                                        sched_arr_time
#   <int> <int> <int>   <int>          <int>     <dbl>    <int>
#   <int>
# 1  2013     1     1     517            515         2      830
#     819
# 2  2013     1     1     533            529         4      850
#     830
# 3  2013     1     1     542            540         2      923
#     850
# 4  2013     1     1     544            545        -1     1004
#    1022
# 5  2013     1     1     554            600        -6      812
#     837
# 6  2013     1     1     554            558        -4      740
#     728
# with 836 more rows, and 11 more variables: ...
```

- `nycflights13::flights` is a data frame that contains all 336,776 flights that departed from New York City in 2013
- it's available in the `library(nycflights13)`



# dplyr : arrange()

- it works similarly to `filter()` except that instead of selecting rows, it changes their order
- it takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, month,day, sched_dep_time)
```

```
# A tibble: 336,776 x 19
#    year month   day dep_time sched_dep_time
#   <int> <int> <int>    <int>          <int>
# 1  2013     1     1      517            515
# 2  2013     1     1      533            529
# 3  2013     1     1      542            540
```

- to rearrange a column in descending row, use `desc()`

```
arrange(flights, month,day, desc(sched_dep_time))
```

```
# A tibble: 336,776 x 19
#    year month   day dep_time sched_dep_time
#   <int> <int> <int>    <int>          <int>
# 1  2013     1     1     2353           2359
# 2  2013     1     1     2353           2359
# 3  2013     1     1     2356           2359
```

# dplyr : mutate()

- besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns

```
flights_sml <- select(flights, year:day, ends_with("delay"),
                      distance, air_time)

mutate(flights_sml, gain = dep_delay - arr_delay,
         speed = distance / air_time * 60)
```

```
# A tibble: 336,776 x 9
#    year month  day dep_delay arr_delay distance air_time  gain speed
#   <int> <int> <int>   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
# 1  2013     1     1       2      11    1400     227    -9  370.
# 2  2013     1     1       4      20    1416     227   -16  374.
# 3  2013     1     1       2      33    1089     160   -31  408.
```

- if we want to keep only the new variables, we use `transmute()`:

```
transmute(flights, gain = dep_delay - arr_delay,
          hours = air_time / 60, gain_per_hour = gain / hours)
```

```
# A tibble: 336,776 x 3
#    gain hours gain_per_hour
#   <dbl> <dbl>         <dbl>
# 1    -9 3.78          -2.38
# 2   -16 3.78          -4.23
# 3   -31 2.67         -11.6
```



# dplyr : summarise() and group_by()

- `summarise()` collapses a data frame to a single row
- it is very useful when combined with `group_by()`
- `group_by()` takes an existing data frame and converts it into a grouped data frame where operations are performed by group

```
not_cancelled <- flights |>
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled |> group_by(year, month, day) |>
                 summarise(mean = mean(dep_delay))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
#    year month   day  mean
#   <int> <int> <int> <dbl>
# 1  2013     1     1 11.4
# 2  2013     1     2 13.7
# 3  2013     1     3 10.9
# 4  2013     1     4  8.97
# 5  2013     1     5  5.73
# 6  2013     1     6  7.15
# 7  2013     1     7  5.42
# 8  2013     1     8  2.56
# 9  2013     1     9  2.30
#10  2013     1    10  2.84
```

# The {magrittr} PIPE operator: %>%

- it process a data-object with a sequence of operations by passing the result of one step as input for the next step using infix-operators rather than the more typical R method of nested function calls
- it comes from the magrittr package (%>%) and starting from R version 4.1 it is part of the language as |>

## Syntax

```
lhs %>% rhs # pipe syntax for rhs(lhs)

lhs %>% rhs(a = 1) # pipe syntax for rhs(lhs, a = 1)

lhs %>% rhs(a = 1, b = .) # pipe syntax for rhs(a = 1, b = lhs)

lhs %<>% rhs # pipe syntax for lhs <- rhs(lhs)

lhs %$% rhs(a) # pipe syntax for with(lhs, rhs(lhs$a))

lhs %T>% rhs # pipe syntax for { rhs(lhs); lhs }
```

- lhs = a value or the magrittr placeholder
- hhs = a function call using the magrittr semantics



# The PIPE operators: %>% , |> examples (1)

## Basic use

```
library(magrittr)

1:10 %>% mean
# [1] 5.5

# is equivalent to
mean(1:10)
# [1] 5.5

# this also works
1:10 %>% mean()
# [1] 5.5
```

but using |> RHS must be written as a function

```
1:10 |> mean
# Error: The pipe operator requires a function call as RHS

1:10 |> mean()
# [1] 5.5
```

# The PIPE operators: %>% , |> examples (2)

```
(years <- factor(2008:2012))

# [1] 2008 2009 2010 2011 2012
# Levels: 2008 2009 2010 2011 2012

as.numeric(as.character(years))
# [1] 2008 2009 2010 2011 2012
```

Piping equivalent:

```
years %>% as.character %>% as.numeric
# [1] 2008 2009 2010 2011 2012

years |> as.character() |> as.numeric()
# [1] 2008 2009 2010 2011 2012

grepl("Wo", substring("Hello World", 7, 11))
# [1] TRUE

"Hello World" %>% substring(7, 11) %>% grepl(pattern = "Wo")
# [1] TRUE

"Hello World" |> substring(7, 11) |> grepl(pattern = "Wo")
# [1] TRUE

"Hello World" %>% substring(7, 11) %>% grepl("Wo", .)
# [1] TRUE

"Hello World" %>% substring(7, 11) %>% { c(paste(. ,'Hi', .)) }
# [1] "World Hi World"
```



# Combining multiple operation with the pipe

- the pipe oeprators, %>%, |> are used to rewrite multiple operations in a compact way; it can be read left-to-right, top-to-bottom
- piping improves code readability

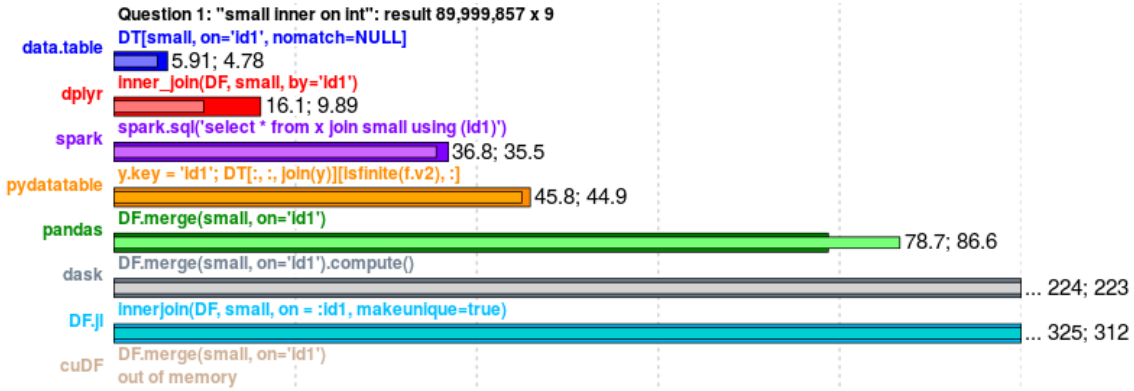
```
select(flights, year:day, ends_with("delay"),
       distance , air_time) %>%
       transmute(gain = dep_delay - arr_delay,
       speed = distance / air_time * 60)
# A tibble: 336,776 x 2
#    gain speed
#   <dbl> <dbl>
# 1    -9  370.
# 2   -16  374.
# 3   -31  408.
# 4    17  517.
# 5    19  394.
# 6   -16  288.
# 7   -24  404.
# 8    11  259.
# 9     5  405.
#10   -10  319.
# ... with 336,766 more rows
```

- behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on

- it provides a high-performance version of base R's `data.frame`
- a `data.table` is created using the `fread()` function for reading data on disk, or provided on the fly with the `data.table()` function

```
DT = data.table(
     id = c("b","a","a","c","c","b"),
     val = c(4,2,3,1,5,6)
)
```


- existing objects can be converted to `data.table` using the `setDT()` and the `as.data.table()` functions
- it is a optimized and runs faster for large data sets (example plot: $10^8$ rows with 7 columns ➜5 GB data) https://h2oai.github.io/db-benchmark/






## data.frame - (1)

- have a 2D matrix like structure: rows and columns. We can:

- subset rows

```
X[X$id != "a"]
```

- select columns

```
X[, "val"]
```

- and do it at the same time:

```
X[X$id != "a", "val"]
```

X

| | id | val |
|---|---|---|
| 1 | b | 4 |
| 2 | a | 2 |
| 3 | a | 3 |
| 4 | c | 1 |
| 5 | c | 5 |
| 6 | b | 6 |

- we can compute on columns:
- sum column valA only for the rows where code != "abd"

```
sum(DF[DF$code != "abd", "valA"])
1.9
```

- we can perform operations on aggregated groups
- sum valA and valB columns for code != "abd" and group by id

```
aggregate(cbind(valA, valB) ~ id,
          DF[DF$code != "abd", ], sum)
```

- we can update values

```
DF[DF$code == "abd", "valA"] <- NA
```

DF

| | id | code | valA | valB |
|---|---|---|---|---|
| 1 | 1 | abc | 0.1 | 11 |
| 2 | 1 | abc | 0.6 | 7 |
| 3 | 1 | abd | NA | 5 |
| 4 | 2 | apq | 0.9 | 10 |
| 5 | 2 | apq | 0.3 | 13 |

| | id | valA | valB |
|---|---|---|---|
| 1 | 1 | 0.7 | 18 |
| 2 | 2 | 1.2 | 23 |



# data.table

- they allow column names to be seen as variables within the [...]
- and computations can be done with them directly
- an additional argument, by is introduced
- a data.table has a row/column data structure, as a data.frame
- subset rows

```
X[id != "a", ]
```

- select columns

```
X[, val]
```

- and compute on columns

```
X[, mean(val)]
```

- subset rows and select/compute on columns

```
X[X$id != "a", mean(val)]
```

- and with a 'virtual 3rd dimension, group by

```
X[X$id != "a", .sum(valA), sum(valB), by=id]
```


X

| | id | val |
|---|---|---|
| 1: | b | 4 |
| 2: | a | 2 |
| 3: | a | 3 |
| 4: | c | 1 |
| 5: | c | 5 |
| 6: | b | 6 |

# equivalence `data.frame` vs `data.table`

- think in terms of basic units: rows, columns and groups