MAP 2112 – Introdução à Lógica de Programação e Modelagem Computacional

1º Semestre - 2020

Prof. Dr. Luis Carlos de Castro Santos

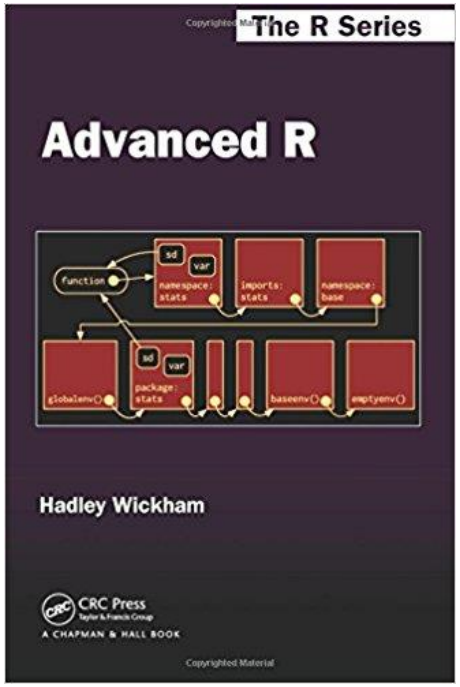
lsantos@ime.usp.br

ROTEIRO

Esse material é fortemente baseado no livro

Advanced R (Chapman & Hall/CRC The R Series)

de Hadley Wickham (http://hadley.nz/) o Cientista-chefe do Rstudio



Seguindo o roteiro do Prof. Roger Peng
http://www.biostat.jhsph.edu/~rpeng/

Quando chegarmos nos tópicos de modelagem e Data Science novas referências serão selecionadas.

Na aula anterior vimos dois objetos armazenadores de dados: os vetores e as listas

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

```
> x <- c(0.5, 0.6)        ## numeric
> x <- c(TRUE, FALSE)     ## logical
> x <- c(T, F)            ## logical
> x <- c("a", "b", "c")   ## character
> x <- 9:29               ## integer
> x <- c(1+0i, 2+4i)      ## complex
```

The most basic object is a vector

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
```

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices (cont'd)

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Console Terminal

```
> m <- matrix(1:6, nrow=2, ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m[2]
[1] 2
> m[2,]
[1] 2 4 6
> m[2,3]
[1] 6
> m[,3]
[1] 5 6
>
```

Matrices (cont'd)

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

cbind-ing and rbind-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Console Terminal

```
> x <- 1:3
> y <- 10:13
> cbind(x,y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
[4,] 1 13
Warning message:
In cbind(x, y) :
  number of rows of result is not a multiple of vector length (arg 1)
> rbind(x,y)
  [,1] [,2] [,3] [,4]
x    1    2    3    1
y   10   11   12   13
Warning message:
In rbind(x, y) :
  number of columns of result is not a multiple of vector length (arg 1)
>
```

RStudio

Console | Terminal

```
> m <- runif(25)
> m
 [1] 0.61535242 0.77510990 0.35556869 0.40584997 0.70664691
 [6] 0.83828767 0.23958913 0.77077153 0.35589774 0.53559704
[11] 0.09308813 0.16980304 0.89983245 0.42263761 0.74774647
[16] 0.82265258 0.95465365 0.68544451 0.50050323 0.27548386
[21] 0.22890394 0.01443391 0.72896456 0.24988047 0.16118328
> dim(m) <- c(5,5)
> m
          [,1]      [,2]       [,3]      [,4]       [,5]
[1,] 0.6153524 0.8382877 0.09308813 0.8226526 0.22890394
[2,] 0.7751099 0.2395891 0.16980304 0.9546536 0.01443391
[3,] 0.3555687 0.7707715 0.89983245 0.6854445 0.72896456
[4,] 0.4058500 0.3558977 0.42263761 0.5005032 0.24988047
[5,] 0.7066469 0.5355970 0.74774647 0.2754839 0.16118328
>
```

Environment | History | Connections

Import Dataset | List

Global Environment

Data

m	num [1:5, 1:5] 0.615 0.775 0.356 0.406 0.707 ...

Files | Plots | Packages | Help | Viewer

runif

R: The Uniform Distribution | Find in Topic

Uniform {stats} R Documentation

The Uniform Distribution

Description

These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `punif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

Usage

```
dunif(x, min = 0, max = 1, log = FALSE)
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
runif(n, min = 0, max = 1)
```

Arguments

Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("foo", "bar", "norf")
> x
foo bar norf
  1   2    3
> names(x)
[1] "foo"  "bar"  "norf"
```

```
> x <- 1:3
> names(x) <- c("a","b","c")
> x$a
Error in x$a : $ operator is invalid for atomic vectors
> x
a b c
1 2 3
> x["a"]
a
1
>
```

Names

Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

```
> x <- list(a=1, b=2, c=3)
> x["a"]
$a
[1] 1
```

Console Terminal

```
> x <- list(1,2,3)
> x
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

>
```

Console Terminal

```
> x <- list(a=1,b=2,c=3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3

> x$a
[1] 1
> x$b
[1] 2
> x$c
[1] 3
>
```

Names

And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

```
> m["a",]
c d
1 3
> m[,"d"]
a b
3 4
```

Factors

general linear models

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*.

- Factors are treated specially by modelling functions like `lm()` and `glm()`
- Using factors with labels is *better* than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Factors

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
no yes
 2   3
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Factors

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
              levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

Missing Values

Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.

- `is.na()` is used to test objects if they are `NA`
- `is.nan()` is used to test for `NaN`
- `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
- A `NaN` value is also `NA` but the converse is not true

Missing Values

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Data frames are used to store tabular data

- They are represented as a special type of list where every element of the list has to have the same length
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
- Data frames also have a special attribute called `row.names`
- Data frames are usually created by calling `read.table()` or `read.csv()`
- Can be converted to a matrix by calling `data.matrix()`

Data Frames

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

2.4.1 Creation

You create a data frame using `data.frame()`, which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Beware `data.frame()`'s default behaviour which turns strings into factors. Use `stringAsFactors = FALSE` to suppress this behaviour:

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

2.4.3 Combining data frames

You can combine data frames using `cbind()` and `rbind()`:

```
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>    x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

Reading Data

Reading Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

Writing Data

There are analogous functions for writing data to files

- write.table
- writeLines
- dump
- dput
- save
- serialize

Reading Data Files with read.table

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset
- `comment.char`, a character string indicating the comment character
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors?

read.table

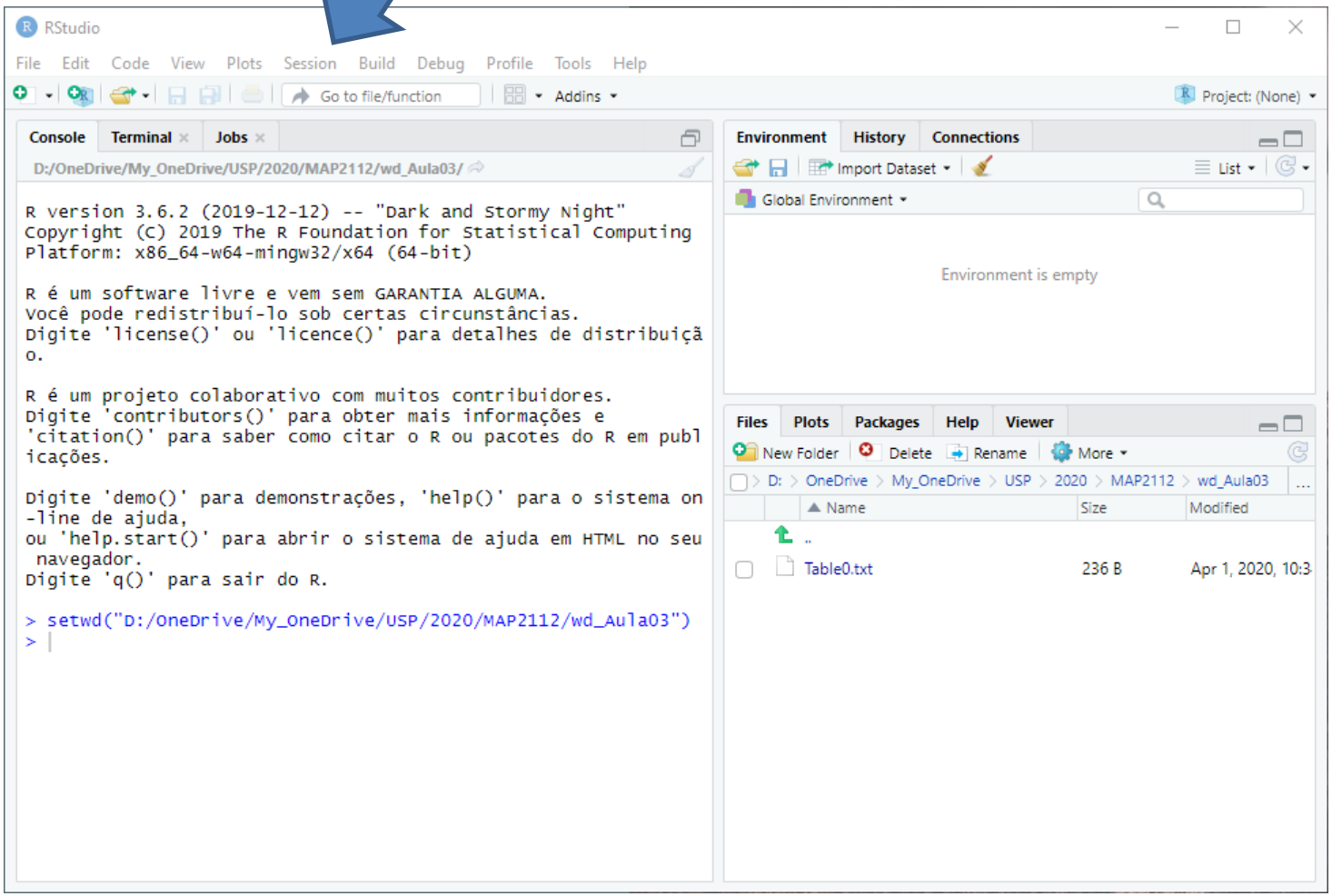
For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
data <- read.table("foo.txt")
```

R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table Telling R all these things directly makes R run faster and more efficiently.
- `read.csv` is identical to read.table except that the default separator is a comma.

Set Working Directory



```
> setwd("D:/OneDrive/My_OneDrive/USP/2020/MAP2112/wd_Aula03")
> x <- read.table("Table0.txt")
> x
        V1 V2  V3 V4 V5
1     Alex 25 177 57  F
2    Lilly 31 163 69  F
3     Mark 23 190 83  M
4   Oliver 52 179 75  M
5   Martha 76 163 70  F
6    Lucas 49 183 83  M
7 Caroline 26 164 53  F
> str(x)
'data.frame':   7 obs. of  5 variables:
 $ V1: Factor w/ 7 levels "Alex","Caroline",..: 1 3 5 7 6 4 2
 $ V2: int  25 31 23 52 76 49 26
 $ V3: int  177 163 190 179 163 183 164
 $ V4: int  57 69 83 75 70 83 53
 $ V5: Factor w/ 2 levels "F","M": 1 1 2 2 1 2 1
>
```

Table0.txt

Table0 - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

```
Alex      25      177      57      F
Lilly     31      163      69      F
Mark      23      190      83      M
Oliver    52      179      75      M
Martha    76      163      70      F
Lucas     49      183      83      M
Caroline  26      164      53      F
```

Ln 1, C 100% Windows (CRLF) UTF-8

Reading in Larger Datasets with read.table

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.

Reading in Larger Datasets with read.table

- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick an dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
> classes <- sapply(x,class)
➢ classes
       a    beta      logic
"integer" "numeric" "logical"
```

Know Thy System

In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

Calculating Memory Requirements

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

1,500,000 × 120 × 8 bytes/numeric

= 1440000000 bytes

= 1440000000 / 2^{20} bytes/MB

= 1,373.29 MB

= 1.34 GB

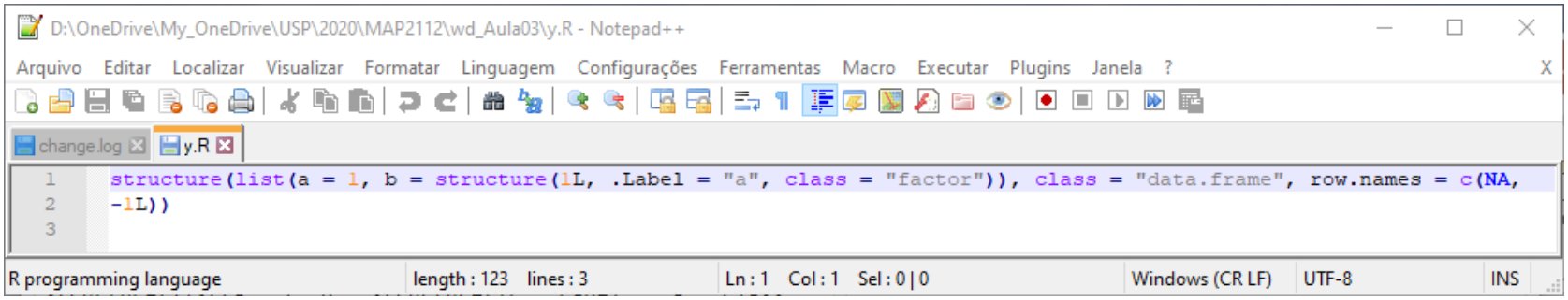
Textual Formats

- `dumping` and dputing are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- `Unlike` writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- `Textual` formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient

dput-ting R Objects

Another way to pass data around is by deparsing the R object with dput and reading it back in using `dget`.

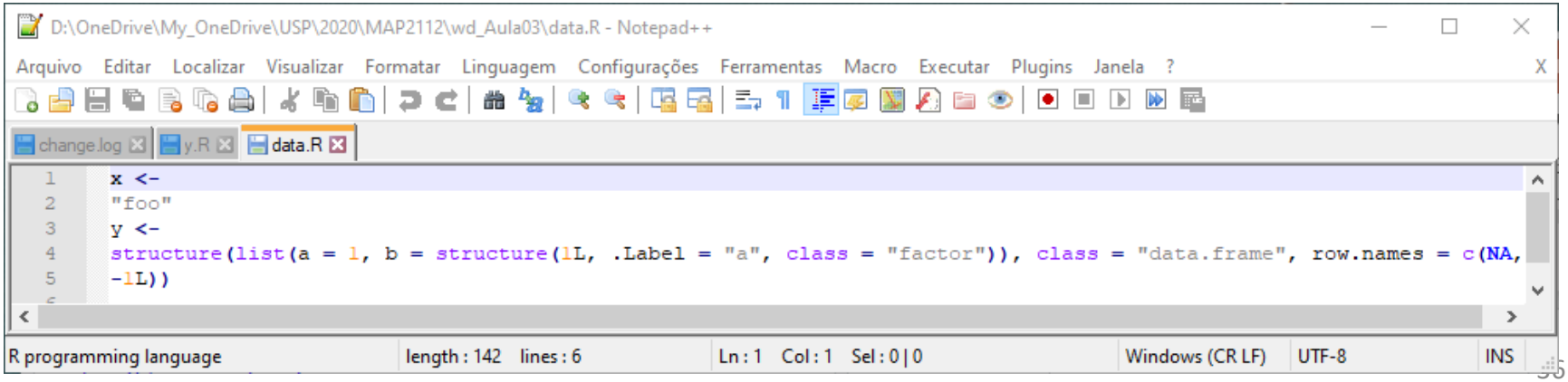
```
> y <- data.frame(a = 1, b = "a")
> dput(y)
structure(list(a = 1,
               b = structure(1L, .Label = "a",
                             class = "factor")),
          .Names = c("a", "b"), row.names = c(NA, -1L),
          class = "data.frame")
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
   a  b
1  1  a
```



Dumping R Objects

Multiple objects can be deparsed using the dump function and read back in using `source`.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a  b
1 1  a
> x
[1] "foo"
```



Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

File Connections

```
> str(file)
function (description = "", open = "", blocking = TRUE,
          encoding = getOption("encoding"))
```

- `description` is the name of the file
- `open` is a code indicating
  - "r" read only
  - "w" writing (and initializing a new file)
  - "a" appending
  - "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

Connections

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

Reading Lines of a Text File

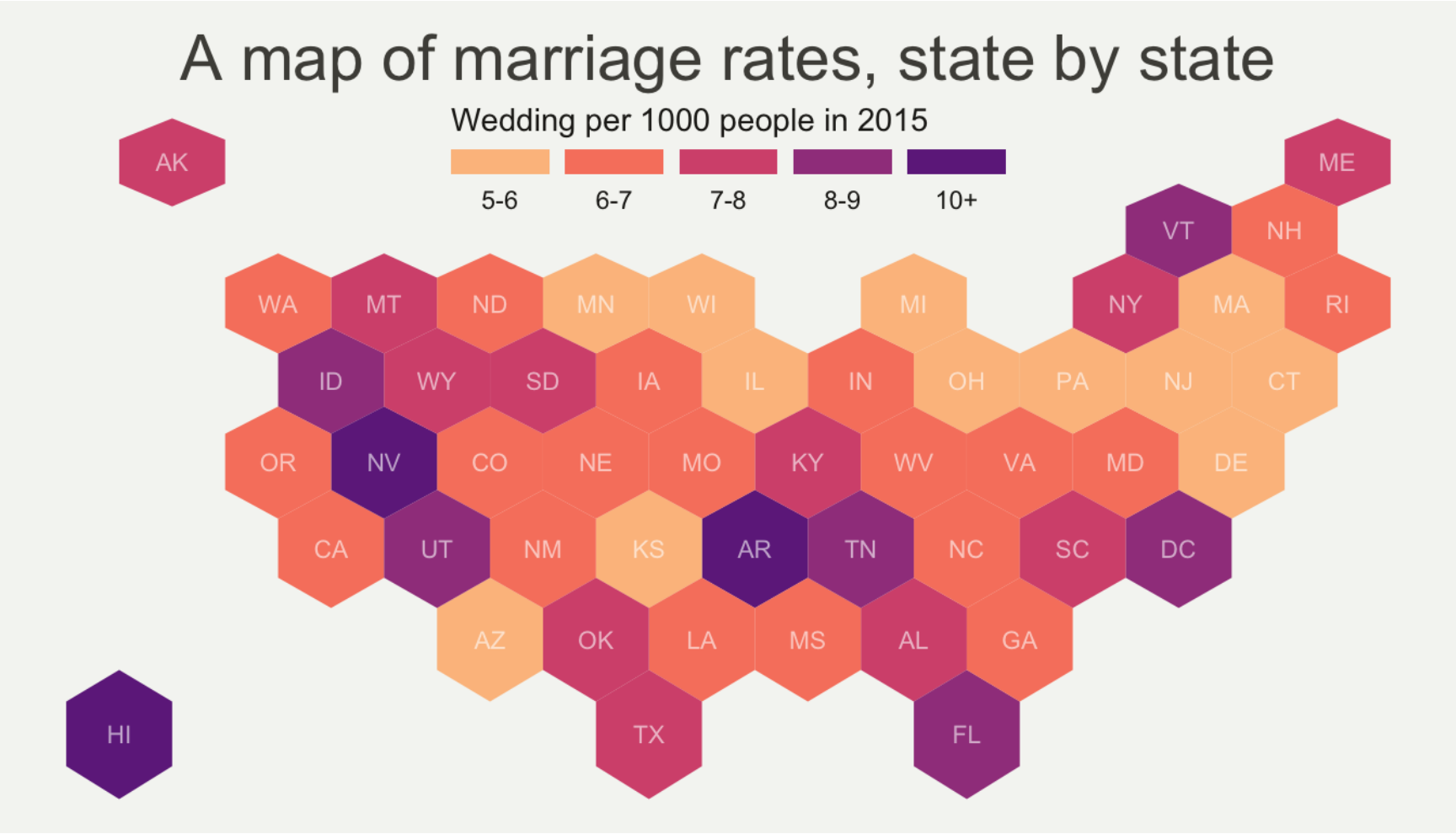
```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"     "10-point" "10th"     "11-point"
 [5] "12-point" "16-point" "18-point" "1st"
 [9] "2"        "20-point"
```

`writeLines` takes a character vector and writes each element one line at a time to a text file.

Reading Lines of a Text File

readLines can be useful for reading in lines of webpages

```
## This might take time
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
> head(x)
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"
[2] ""
[3] "<html>"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8
```



https://www.r-graph-gallery.com/328-hexbin-map-of-the-usa.html