

# Introduction to the R Language

## Reading and Writing Data

Biostatistics 140.776

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

There are analogous functions for writing data to files

- `write.table`
- `writelnLines`
- `dump`
- `dput`
- `save`
- `serialize`

# 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

```
> 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)

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 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?

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.



# 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 and 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.

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?

$$\begin{aligned} 1,500,000 \times 120 \times 8 \text{ bytes/numeric} &= 1440000000 \text{ bytes} \\ &= 1440000000 / 2^{20} \text{ bytes/MB} \\ &= 1,373.29 \text{ MB} \\ &= 1.34 \text{ GB} \end{aligned}$$

# Saving Data in Non-tabular Forms

For temporary storage or for transport, it is more efficient to save data in (compressed) binary form using `save` or `save.image`.

```
x <- 1
y <- data.frame(a = 1, b = "a")
save(x, y, file = "data.RData")
load("data.RData")  ## overwrites existing x and y!
```

Binary formats are not great for long-term storage because if they are corrupted, recovery is usually not possible.

# Deparsing 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"
```

# Textual Formats

- `dump` and `dput` 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 work much better with programs like `git` which can only track changes meaningfully in text files
- Textual formats adhere to the “Unix philosophy”



# Reading Lines of a Text File

The `readLines` function can be used to simply read lines of a text file and store them in a character vector.

```
> 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 Transiti
[2] ""
[3] "<html>"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/ht
```

Serialization is the process of taking an R object and converting into a representation as a “series” of bytes.

- The `save` and `save.image` functions serialize R objects and then save them to files
- The `serialize` function can be used to serialize an R object to an arbitrary connection (database, socket, pipe, etc.)
- `unserialize` reads from an arbitrary connection and inverts a serialization, returning an R object

# Serialization

```
> x <- list(1, 2, 3)
> serialize(x, NULL)
 [1] 58 0a 00 00 00 02 00 02 06 01 00 02 03 00 00
[16] 00 00 13 00 00 00 03 00 00 00 0e 00 00 00 01
[31] 3f f0 00 00 00 00 00 00 00 00 00 0e 00 00 00
[46] 01 40 00 00 00 00 00 00 00 00 00 00 0e 00 00
[61] 00 01 40 08 00 00 00 00 00 00
```

# Serialization

```
> con <- gzfile("foo.gz", "wb")
> serialize(x, con)
NULL
> close(con)
>
> con <- gzfile("foo.gz", "rb")
> y <- unserialize(con)
> identical(x, y)
[1] TRUE
```

# Data Output Summary

- `write.table`, `write.csv` — readable output, textual, little metadata
- `save`, `save.image`, `serialize` — exact representation, efficient storage if compressed, not recoverable if corrupted
- `dput`, `dump` — textual format, somewhat readable, metadata retained, not usable for more exotic objects (environments)