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`

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 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.

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.

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?

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

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

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

Serialization

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)

Reading and writing files

Importing data in R

Data contained in external text files can be imported in R using one of the following functions:

- scan()
- read.table()
- read.csv()
- read.csv2()
- read.delim()
- read.delim2()

The requirements on the form of data set are sometimes quite strict, so it is better to modify input files to satisfify this requirements.

The function scan()

This function is the most flexible: it can be used to read logical, integer, numeric, complex, character, raw data and *lists*

```
scan(file = " ", what = double(0), n = -1, sep = "", dec =
    ".", skip = 0, na.strings = "NA")
```

file: the name of the file which the data are to be read from;

what: the type of data to be read (logical, integer, numeric, complex, character, raw and list);

n: the maximum number of data values to be read, defaulting to no limit;

scan(file = "", what = double(0), n = -1, sep = "", dec = ".", skip = 0, na.strings = "NA")

sep: the field separator character. If sep = “”, the separator is ‘white space’, that is one or more spaces, tabs;

dec: the character used in the file for decimal points;

skip: the number of lines of the input file to skip before beginning to read data values;

na.strings: character vector. Elements of this vector are to be interpreted as missing (NA) values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields

The function read.table()

This function reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

read.table(file, header = FALSE, sep = "", dec = ".", row.names, col.names)

header: a logical value indicating whether the file contains the names of the variables as its first line

sep: the field separator character. If sep = "" (the default for read.table) the separator is ‘white space’, that is one or more spaces, tabs;

dec: the character used in the file for decimal points;

row.names: a vector of row names;

col.names: a vector of column names.

Input file format

Row names

Column names

	N.Amer	Europe	Asia	S.Amer	Oceania	Africa	Mid.Amer
1951	45939	21574	2876	1815	1646	89	555
1956	60423	29990	4708	2568	2366	1411	733
1957	64721	32510	5230	2695	2526	1546	773
1958	68484	35218	6662	2845	2691	1663	836
1959	71799	37598	6856	3000	2868	1769	911
1960	76036	40341	8220	3145	3054	1905	1008
1961	79831	43173	9053	3338	3224	2005	1076

```
a <-read.table("Worldtelephones.txt")
>str(a)
'data.frame':   7 obs. of  7 variables:
 $ N.Amer  : int  45939 60423 64721 68484 71799 76036 79831
 $ Europe  : int  21574 29990 32510 35218 37598 40341 43173
 $ Asia    : int  2876 4708 5230 6662 6856 8220 9053
 $ S.Amer  : int  1815 2568 2695 2845 3000 3145 3338
 $ Oceania : int  1646 2366 2526 2691 2868 3054 3224
 $ Africa  : int  89 1411 1546 1663 1769 1905 2005
 $ Mid.Amer: int  555 733 773 836 911 1008 1076
```

If only column labels are present add the option "header=T"

The function read.csv() and read.csv2()

- **read.csv()** is intended for reading 'comma separated value' (CSV) files, where the decimal point is "."

- **read.csv2()** is the variant used in countries that use a comma (",") as decimal point and a semicolon (";") as field separator.

```
read.csv(file, header = TRUE, sep = ",", dec=".“)
read.csv2(file, header = TRUE, sep = ";",  dec=",“)
```

The functions read.delim() and read.delim2()

- They are intended to read TAB separated files

```
read.delim(file, header = TRUE, sep = "\t", dec=".", fill =
    TRUE, ...)

read.delim2(file, header = TRUE, sep = "\t", dec=",", fill =
    TRUE,...)
```

sep: the field separator character. “\t” (default for read.delim) stands for TAB separator;

fill: if TRUE then in case the rows have unequal length, blank fields are implicitly added

Exporting Data

R objects can be exported to a text file using the **cat()** function:

```
cat (x , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

x: **R** object

file: character string naming the file to print to. If "" (the default), cat prints to the standard output connection, the console unless redirected by sink.

sep: a character vector of strings to append after each element.

fill: controls how the output is broken into successive lines.

append:logical. If TRUE output will be appended to file; otherwise, it will overwrite the contents of file.

Writing data frames

Often it is useful to write a matrix or a data frame to a file as a grid of elements.

- write() writes out a matrix or vector in a specified number of columns.

- write.table() writes out a data frame (or an object that can be coerced to a data frame) with row and column labels

The function write()

write (x, file = "data", ncolumns =, append = FALSE,
sep = " ")

x	the data to be written out.
file	the file to write to
ncolumns	the number of columns to write the data in.
append	if TRUE the data x are appended to the file.
sep	a string used to separate columns. Using sep = "\t" gives tab delimited output; default is white space " ".

The function write.table()

```
write.table(x, file = "", append = FALSE, sep = " ", na =
    "NA", dec = ".", row.names = TRUE, col.names = TRUE)
```

na:

the string to use for missing values in the data (default is NA)

row.names (col.names):

either a logical value indicating whether the row (column) names of x are to be written along with x, or a character vector of row (column) names to be written.

The functions write.csv() and write.csv2()

write.csv(x, file=“ ”)

write to a file using the comma (“,”) as the field separator

write.csv2(x, file=“ ”)

write to a file using semicolon (“;”) as the field separator and the comma as the decimal point

Reading Excel files

An excel file can contain many sheets, and the sheets can contain formulae, macros and so on: not all readers can manage this.

Instead of importing the data contained in .xls files in R, it would be easier if:

- you export data of the .xls file in a .txt file, in tab-delimited or comma-separated form
- you use R functions *read.delim()* or *read.csv()* to import the .txt file into the R workspace