Week one part 2

Al

2026-01-13

Reading tabular data

Reading and writing data in R. There are a few principal reading function in 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 worklspaces
- `unserialize`, for reading single R objects in binary form

Writing data

- 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 characer vector indicating the class of each column in the data set.
- `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
- `stringAsFactors`, 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

R will auto - 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 excepot that the default separator is a comma.

Reading Large Tables

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
- Make a rough calc of the memory required to store your dataset. If the dataset is larger than the ammount of RAM of your computer, you can prob stop right there. - Set "comment.char =" " "

Reading in Larger Datasetsd with read.table

- Use the colClasses argument. Specifying this option instead of using the default can make `read.table` ran MUCH faster. If all columns are "numeric" then you can just set `colclasses = "numeric"`.

- Set nrows. This doesn't make R run faster but it helps with memory usage. A mild overestimate is ok. 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 fwe things about your systems - how much RAM
- Other applications
- Other users log in
- What OS
- Is the OS 32 or 64 bit

Calc memory requirement

A dataframe with 1500000 rows and 120 column, all of which is numeric data. Ropughly, how much memory is required to store this data drame?

$$1500000 * 120 * 8 \text{ bytes/numeric} = 1440000000 \text{ bytes} = 144000000/2^{20} \text{ bytes/MB} = 1.34 \text{ GB}$$

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 of 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 git whioch 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 formal is not very space-efficient

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

```
y<- data.frame(a = 1,b = "a")
y
```

```
##   a b
## 1 1 a
```

```
dput(y)
```

```
## structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

```
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. Connection 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"),
##     raw = FALSE, method = getOption("url.method", "default"))
```

- `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, connection 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")
```

But it might be helpful to read parts of a file as in

```
con <- gzfile("words.gz")
x <- readLines( con , 10)
x
```

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

`readLines` can be useful for reading in lines of webpages

```
con <- url("https://www.reddit.com/r/typing/comments/l3loix/which_finger_do_you_press_c_with/", "r")
x<-readLines(con)
```

```
## Warning in readLines(con): incomplete final line found on
## 'https://www.reddit.com/r/typing/comments/l3loix/which_finger_do_you_press_c_with/'
```

```
head(x)
```

```
## [1] ""
## [2] "    <!DOCTYPE html>"
## [3] "    <html lang=\"en-US\" class=\"theme-beta\" dir=\"ltr\" device=\"desktop\">"
## [4] "      <head prefix=\"og: https://ogp.me/ns#\">"
## [5] "        <title>Reddit - The heart of the internet</title>"
## [6] "        <meta name=\"viewport\" content=\"width=device-width, initial-scale=1, viewport-fit=cove
```

Subsetting: Basics

There are a number of operators that can be used to extract subsets of R objects ([,[[, or $).

- [always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- [[is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or a data frame
- $ is used to extract elements of a list or data frame by name; semantics are similar to that of [[

```r
x <- c("a","b","c","c","d","a")
x[1]#Numeric Index
```

```
## [1] "a"
```

```r
x[2]
```

```
## [1] "b"
```

```r
x[1:4]
```

```
## [1] "a" "b" "c" "c"
```

```r
x[x > "a"]#Logical Index
```

```
## [1] "b" "c" "c" "d"
```

```r
u <- x > "a"
u
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

```r
x[u]
```

```
## [1] "b" "c" "c" "d"
```

Subsetting List

```r
x<- list(foo = 1:4,bar=0.6)
x[1]# is a list
```

```
## $foo
## [1] 1 2 3 4
```

```r
x[[1]] # is a vector of numbers
```

```
## [1] 1 2 3 4
```

```r
x$bar # is a numeric object
```

```
## [1] 0.6
```

```r
x[["bar"]] # is a numeric object
```

```
## [1] 0.6
```

```r
x["bar"] # is a list
```

```
## $bar
## [1] 0.6
```

```r
# what if i want more than one object
x <- list(foo = 1:4,bar=0.6, baz = "hello")
x[c(1,3)]
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

The [[operator can be used with computed indices; $ can only be used with literal names.

```r
x <- list(foo = 1:4,bar=0.6, baz = "hello")

name <- "foo"
x[[name]] # computed index for foo
```

```
## [1] 1 2 3 4
```

```r
x$name # Element 'name' doesn't exits
```

```
## NULL
```

```r
x$foo # Element 'foo' does exist
```

```
## [1] 1 2 3 4
```

Subsetting Nested Elements of a List

The [[can take an integer sequence.

```r
x <- list( a = list(10,12,14),b=c(3.14,2.81))
x[[c(1,3)]]
```

```
## [1] 14
```

```r
x[[1]][[3]]
```

```
## [1] 14
```

```r
x[[c(2,1)]]
```

```
## [1] 3.14
```

Subsetting : Matrices

Matrices can be subsetted in the usual way with (i,j) type indeces

```r
x <- matrix(1:6,2,3)
x[1,2]
```

```
## [1] 3
```

```r
x[2,1]
```

```
## [1] 2
```

```r
x[1,]
```

```
## [1] 1 3 5
```

```r
x[,2]
```

```
## [1] 3 4
```

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix. This behavior can be turned off by setting drop = False

```r
x <- matrix(1:6,2,3)
x[1,2]
```

```
## [1] 3
```

```r
x[1,2,drop = FALSE]
```

```
##      [,1]
## [1,]    3
```

Similarly, subsetting a single column or row will give you a vector, not a matrix by default.

Subsetting with names

Partial Matching

Partial matching of names is allowed with [[and $

```
x <- list(aardvark = 1:5)
x$a
```

```
## [1] 1 2 3 4 5
```

```
x[["a"]]
```

```
## NULL
```

```
x[["a", exact = FALSE]]
```

```
## [1] 1 2 3 4 5
```

Removing missing values

A common task is to remove missing values (NAs)

```
x<- c(1,2,NA,4,NA,5)
x
```

```
## [1]  1  2 NA  4 NA  5
```

```
bad <-is.na(x)
bad
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```
x[!bad]
```

```
## [1] 1 2 4 5
```

What if there are multiple things and you want to take the subset with no missing values

```
x <- c(1,2,NA,4,NA,5)
y <- c("a",NA,"b","d",NA,"f")

good <- complete.cases(x,y)
good
```

```
## [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE
```

```
x[good]
```

```
## [1] 1 4 5
```

```
y[good]
```

```
## [1] "a" "d" "f"
```

Vectorized Operations

Feature of R for the command line since there is not gonna be any looping. As in mathematical operations are coordinatewise

Many operations in R are vectorized making code more efficient, concise, and easier to read

```
x <- 1:4; y<- 6:9
x+y
```

```
## [1]  7  9 11 13
```

```
x>2
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

```
x>=2
```

```
## [1] FALSE  TRUE  TRUE  TRUE
```

```
y==8
```

```
## [1] FALSE FALSE  TRUE FALSE
```

```
x*y
```

```
## [1]  6 14 24 36
```

```
x/y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Same for matrix is multiplication by coordinate. Real matrix multiplication is given by `%*%`