tut_2

Reading tabular data

There are few main functions used for reading data

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

Reading large data

read.file(. . .):

- *file*, name of a file, or connection
- *header*, logical indicating if the file has a header line, or the dataset in the table starts right away
- *sep*, a string indicating how the columns are separated(, or a space etc.)(for read.csv default is comma, for read.table default is space)
- *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
- *skip*, the number of lines to skip form the beginning
- *stringAsFactors*, should character variables be coded as factors?

You can optimize the reading of the files using following tips:

- *Set comment.char = "" if there are no commented lines in the file*
- *Using nrows, even as a mild over-estimate, will help memory usage.*
- *Use the colClasses argument, so that R doesn't have to spend time and memory on figuring out which type of data is presented in each column. If all columns in the dataset are of the same type you can just assign one type of variable to the colClasses = "numeric"(Ex)**
- **Or you could use only first few rows in order to specify the classes of columns like so:*

don't run this script, just look through it

```
# initial <- read.table("name_of_the_file.txt", nrows = 5)
# classes <- sapply(inital,class)
# tabAll <- read.table("name_of_the_file.txt",  colClasses=classes)
```

Calculating Memory requirements(optional)

This segment is mostly important when working with exeptionally large datasets

If the data is with 1,500,000 rows and 120 columns, all of which are numeric data:

1,500,000 * 120 * 8 bytes/numeric
= 1,440,000,000 bytes
= 1,440,000,000/2^20 bytes/MB
= 1,373.29 MB
~ 1.34 GB

but you might need twice more memory for the reading itself.

Textual data formats

```
i <- data.frame(a1=1,b1="a")
dput(i)
```

```
## structure(list(a1 = 1, b1 = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

```
dput(i, file="y.R") #puts the data frame into the file named y.R if there is no such file creates it
new.y <- dget("y.R") #later on you could get this data frame from the file back using this command
new.y
```

```
##   a1 b1
## 1  1  a
```

Multiple obects could be reconstructed using dump function and read back using source fucntion:

```
dump("i", file="y.R")
rm(i)
source("y.R")
i
```

```
##   a1 b1
## 1  1  a
```

Connections

Data are read in using *connection* interfaces. Connections can be made to files or to other exotic things.

- **file**, opens a connection to a file(text files for ex.)
- **gzfile**, opens a connection to a file compressed with gzip
- **bzfile**, opens a connection to a file compressed with bzip2
- **url**, opens a connections to a webpage

```
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 indication

- "r" read only
- "w" writing(and initializing a new file)
- "a" appending
- "rb", "wb", "ab" reading, writing, appending in binary mode(Windows)

Connections aren't specifically used for regular reading of the files('read' functions exist for this purpose) but rather when there is a goal of reading data in a specific way. For example, if you have a text file, but you need only the first 5 lines of the file to be read:

```
connection <- file("vocab.txt")
voc <- readLines(connection,5)
voc
```

```
## [1] "apple" "bunny" "cave"  "drums" "eagle"
```

```
close.connection(connection)
```

Subsetting

- [] always returns an object of the same class as the original; can be used to select more than one element
- [[]] is used to extract elements of list or data frame; can be used to extract a single element and the class of the returned object would not necessarily be a list or a data frame
- $ used to extract elements of a list or data frame by name;

```
a <- c(1:5)
a[1]
```

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

```
a[1:3]
```

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

```
a[a>3]
```

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

```
y <- a>3
y
```

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

Subsetting lists

```
li <- list(one = 1:4,two = 8, three=0:3)
li[1]
```

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

```
li[[1]]
```

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

```
li$two #the same as
```

```
## [1] 8
```

```
li[["two"]] # the same as
```

```
## [1] 8
```

```
name <- "two"
li[[name]]
```

```
## [1] 8
```

```
#the same as
li[c(1,1)]
```

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

```
#If we have a complex list then navigating in it looks like this:
li2 <- list(list(4,2,6),list(7,8))
li2[[c(1,2)]]
```

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

Subsetting in matrices

```
m <- matrix(1:6,nrow = 3, ncol = 2)
m[1,2] #1st row, 2nd column
```

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

```r
m[1,] #the full 1st row
```

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

```r
m[,2] #the full column
```

```
## [1] 4 5 6
```

Now as you can see this indexing when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather that a 1x1 matrix, which is logical, but sometimes you might need to turn off this default behavior:

```r
m[1, ]
```

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

```r
#see the difference?
m[1, , drop=F]
```

```
##      [,1] [,2]
## [1,]    1    4
```

Partial matching

This is a very useful thing when you are using console line and have a lot of long names for lists/vectors etc. For example:

```r
x<- list(armagedon = 1:5)
x$a #would give you all elements of the list x that start with an a(and we only have one element in the
```

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

```r
#or you could also use brackets, but yuo need to turn of the exact "search" in this case:
x[["a", exact = F]]
```

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

Removing NA values

```r
x<-c(1,2,NA,3,4,NA,5)
bad <- is.na(x)
x[!bad]
```

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

Now let's say you have multiple vectors with missing values on the same places(improtantly vectors must be the same length!):

```r
y<- c("a","b",NA,"c","d",NA,"e")
good <- complete.cases(x,y)
x[good]
```

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

```r
y[good]
```

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

Now let's look at removing NA cases from dataframes

```r
b <- data.frame(ID=1:3, Names =c(NA,"Marry","Elena"),Passed=c(T,F,T))
b
```

```
##   ID Names Passed
## 1  1  <NA>   TRUE
## 2  2 Marry  FALSE
## 3  3 Elena   TRUE
```

```r
good <- complete.cases(b)
b[good,]
```

```
##   ID Names Passed
## 2  2 Marry  FALSE
## 3  3 Elena   TRUE
```

Vectorized operations

This is a small note regardign work with vectors in R:

```r
a<- 1:4; b<-5:8
a + b #note that for ariphmetic operations can only be done for vectors of the same length
```

```
## [1]  6  8 10 12
```

```r
a > 2
```

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

```r
a >= 2
```

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

```r
b == 8
```

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

```
a * b
```

```
## [1]  5 12 21 32
```

```
a / b
```

```
## [1] 0.2000000 0.3333333 0.4285714 0.5000000
```

Noe let's look at the same with matrices:

```
m1 <- matrix(data = 6:9,nrow=2,ncol=2)
m2 <- matrix(data = 1:4,nrow=2,ncol=2)
m1 * m2 #multiplying elements of the metrix by index one by one
```

```
##      [,1] [,2]
## [1,]    6   24
## [2,]   14   36
```

```
m1 / m2
```

```
##      [,1]     [,2]
## [1,]  6.0 2.666667
## [2,]  3.5 2.250000
```

```
m1 %*% m2 #actual algebrais multiplication of matrices
```

```
##      [,1] [,2]
## [1,]   22   50
## [2,]   25   57
```