STAT40730 Data Programming with R (online)

Isabella Gollini

Lecture 9 - Input/output and string manipulation

Input/output and string manipulation

- `scan`
- Printing to the screen
- Reading and writing files
- Accessing the internet
- String manipulation
- Regular expressions
- Examples

Why are we studying this?

- Input/Output (IO) is an often underemphasised topic in data analytics.
- It is vital to be able to input and output data in a manner that is most convenient for the users of the programs that you use and write.
- Similarly, string manipulation has a strong role in dealing with real-world messy data sets.
- This week we will also cover how to access the internet through R, and show how to do some web scraping.

Keyboard and monitor

- Suppose we have four files called `z1.txt`, `z2.txt`, `z3.txt` and `z4.txt` - you can find all these files in the Blackboard folder.

- We can read them in using the `scan` function (remember to set the working directory first):

```
scan('z1.txt')
```

```
## [1] 123   4   5   6
```

```
scan('z2.txt')
```

```
## [1] 123.0   4.2   5.0   6.0
```

```
scan('z3.txt')
```

```
## Error in scan("z3.txt"): scan() expected 'a real', got 'abc'
```

```
scan('z3.txt', what = 'character')
```

```
## [1] "abc" "de"  "f"   "g"
```

```
scan('z4.txt', what = 'character')
```

```
## [1] "abc" "123" "6"   "y"
```

Some notes about `scan`.

- `scan` will read in elements individually.
- By default it will separate elements by a space or a new line and expect them to be of mode double.
- The extra `what = 'character'` argument allows you to specify that it should be character.
- The extra argument `sep` allows you to change how it separates elements.
- Normally we would return the value to an object:

```
x1 <- scan('z3.txt', what = 'character')
x1
```

```
## [1] "abc" "de"  "f"   "g"
```

```
x2 <- scan('z3.txt', what = 'character', sep = '\n')
x2
```

```
## [1] "abc"  "de f" "g"
```

Some more notes about `scan`.

- `scan` is a quick and simple way of reading in an entire file.
- It deals well with irregularly-shaped data (i.e. where it isn't in a neat rectangular format).
- You can also use `scan` to read in from the R console:

```
v <- scan('')
v
```

- The argument `quiet = TRUE` stops scan from telling you how many items it has read.

Reading in from the keyboard

- If you just want to read in a single line from the keyboard, the `readline` function is useful:

```
z <- readline()
z
```

```
inits <- readline('Type your
  initials: ')
inits
```

- The related function `readLines` is useful for reading in single lines from files.

Printing to the screen

- We have already met the `print` function numerous times.

```
x <- 1:3
print(x^2)
```

```
## [1] 1 4 9
```

- More useful is `cat`, which presents the output in a neater fashion:

```
cat('The values of x are', x, '\n')
```

```
## The values of x are 1 2 3
```

- Note that `cat` requires a new line character `'\n'` so that R starts writing on the next line.
- By default `cat` introduces spaces between the different elements, use `sep = ''` to stop this happening.

Reading and writing files

Reading in using `read.table` Text and binary files Connections Writing data Reading from the clipboard Other file types

Reading in files

- We have already met `read.table` and `read.csv` as methods to read in data frames and matrices.
- Another related function is `read.delim` which reads in tab-delimited files by default.
- All files read in this way need to have the same number of columns in each row but can be of mixed mode in each column.
- There is no easy way to read in a matrix directly from a text file. You will usually have to read in the file and use `as.matrix` on it.

Reading in files

- There are lots of useful extra arguments to the `read.XXXX` functions, including:
  - `header` which if `TRUE` sets the column names to be the top row.
  - `sep` which changes the delimiter type (useful in `read.table`).
  - `skip` which skips out some of the top lines.
  - `nrows` which will read in only a set number of rows.
  - `colClasses` which will set modes for each different column.

  - `fill` which will over-ride the need to have equal numbers of columns for each row.
  - `stringsAsFactors` which will read in character columns as factors (though is over-ridden by `colClasses` which is more powerful).

Text and binary files

- We will call a text file one which can be opened and read by us in a text editor such as notepad or Word. A binary file by contrast is one which is stored in a more efficient, non-human readable format, such as jpeg or executable program files.
- R will save and load data in either format but it is usually preferable (especially for big data sets) to store in binary format
- The `save` and `load` functions we met in Lecture 8 store objects in binary format, whereas the `read.table` functions we have just discussed read in text format data.
- A very powerful way of reading in data (especially from the web) is to create a connection with a file first and then read it in in chunks - see `?connection` for details of the functions we will use to do this.

Creating a connection

- We can create a connection to a file with the `file` function:

```
y <- file('z5.txt','r')
readLines(y, n = 1)
```

```
## [1] "John 25"
```

```
readLines(y, n = 1)
```

```
## [1] "Mary 28"
```

```
readLines(y, n = 1)
```

```
## [1] "Jim 19"
```

```
readLines(y, n = 1)
```

```
## character(0)
```

- Each call to `readLines` now reads in one line of the file. When we reach the end of a file R gives a blank result.
- The `seek` function can be used to rewind the file and close to close it:

```
seek(con = y, where = 0)
```

```
## [1] 23
```

```
readLines(y, n = 1)
```

```
## [1] "John 25"
```

```
close(y)
```

Writing data

- We have already met `save` which saves R data in binary format.
- Another useful function is `write.table` which saves data in text format but contains some rather odd default options.

- Try:

```
kids <- c('Jack', 'Jill')
ages <- c(12, 10)
d <- data.frame(kids, ages, stringsAsFactors = FALSE)
write.table(d, file = 'd1.txt')
write.table(d, file = 'd2.txt', quote = FALSE, row.names = FALSE)
```

- The second of these files will look far neater.
- You can similarly use `cat` to send information to a file

```
x <- 2:4
cat('abc\n ', x, file = 'd3.txt', sep = '')
cat(x, 'de\n', x, file = 'd4.txt', append = TRUE)
```

Getting file and directory information

- R has a variety of functions for accessing and manipulating files (first introduced in lecture 4), including:
  - `file.info()` which gives the file size, creation time, etc.
  - `list.files()` which gives the names of all the files in the specified directory.
  - `file.exists()` which will return a Boolean vector indicating whether the given file(s) are available.
  - `file.create()` or `file.remove()` create/remove files of the given names.
  - `file.rename()` or `file.copy()` rename/copy files from one location to another.

Reading and writing from the clipboard

- The (Windows-only) functions `readClipboard` and `writeClipboard` allow you to use whatever you have copied or pasted in R.
- It will only work with character vectors:

```
x <- "hello world"
writeClipboard(x)
x <- 3.14
writeClipboard(x)
x <- readClipboard()
```

- This can be a quick and dirty way of getting data into R.

Reading in other common file-types

- If you have data in another standard format there is usually an R package to read it in:
  - Excel data.
  - SAS data.
  - json data.
  - Matlab data.
- The R package `foreign` covers lots of these topics. Also lots of packages to read in database files.

Accessing the internet

Reading in web data `RCurl` `XML`

Reading in web data

- The functions `read.table` and `scan` will also accept web URLs as arguments.
- A simple example (from the excellent, free, *Elements of Statistical Learning* book) is:

```
prostate.url <-
  'https://web.stanford.edu/~hastie/ElemStatLearn/datasets/prostate.data'
prostate <- read.table(prostate.url, header = TRUE, sep = '\t', row.names = 1)
head(prostate)
```

```
##       lcavol  lweight age      lbph svi       lcp gleason pgg45       lpsa
## 1 -0.5798185 2.769459  50 -1.386294   0 -1.386294       6     0 -0.4307829
## 2 -0.9942523 3.319626  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 3 -0.5108256 2.691243  74 -1.386294   0 -1.386294       7    20 -0.1625189
## 4 -1.2039728 3.282789  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 5  0.7514161 3.432373  62 -1.386294   0 -1.386294       6     0  0.3715636
## 6 -1.0498221 3.228826  50 -1.386294   0 -1.386294       6     0  0.7654678
##   train
## 1  TRUE
## 2  TRUE
## 3  TRUE
## 4  TRUE
## 5  TRUE
## 6  TRUE
```

Getting data from the web

- Besides `scan` and `read.table`, R has a number of other functions for accessing and sending data over the internet.
- To use these, you need some background knowledge of TCP/IP, port numbers, and sockets, which we will not cover here.
- These functions include:
  - `readLines` and `writeLines` which allow you to read and send data line by line.
  - `serialize` and `unserialize` which send R objects
  - `readBin` and `writeBin` which send and receive data in binary format.
- We will not go through these, as R has many packages that allow us to read in web data more simply...

`RCurl`.

- The `RCurl` package allow easy downloading of webpages which can then manipulated like character strings.
- On Windows you first need to download Curl (it's usually already in OS X and Linux)

```
library(RCurl)
oct17 <- getURL("https://stat.ethz.ch/pipermail/r-help/2017-October/date.html")
webpage <- strsplit(oct17,"\n")[[1]]
webpage[1:14]
```

```
##  [1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">"
##  [2] "<HTML>"
##  [3] "  <HEAD>"
##  [4] "     <title>The R-help October 2017 Archive by date</title>"
##  [5] "     <META NAME=\"robots\" CONTENT=\"noindex,follow\">"
##  [6] "     <META http-equiv=\"Content-Type\" content=\"text/html; charset=us-ascii\">"
```

```
##  [7] "  </HEAD>"
##  [8] "  <BODY BGCOLOR=\"#ffffff\">"
##  [9] "      <a name=\"start\"></A>"
## [10] "      <h1>October 2017 Archives by date</h1>"
## [11] "      <ul>"
## [12] "         <li> <b>Messages sorted by:</b>"
## [13] "\t        <a href=\"thread.html#start\">[ thread ]</a>"
## [14] "\t\t<a href=\"subject.html#start\">[ subject ]</a>"
```

- This downloads the webpage data for the given URL and then splits it according to each new line.
- This object is still a bit of a mess (we need the string manipulation tools we cover later).

XML.

- The `XML` package includes functions to read in data from tables on a webpage.

```
library(XML)
myurl <- getURL('https://en.wikipedia.org/wiki/Comparison_of_statistical_packages')
mytable <- readHTMLTable(myurl)[[2]]
head(mytable)
```

```
##           V1      V2     V3    V4  V5   V6    V7
## 1    Product Windows Mac OS Linux BSD Unix Cloud
## 2   ADaMSoft     Yes    Yes   Yes Yes  Yes    No
## 3 Analyse-it     Yes     No    No  No   No    No
## 4       BMDP     Yes                          No
## 5   Dataplot     Yes    Yes   Yes Yes  Yes    No
## 6       ELKI     Yes    Yes   Yes Yes  Yes    No
```

- Here `readHTMLTable` extracts all of the tables, and then we pull out the second one.

String manipulation

Why this is helpful Common string functions Printing and concatenating Finding patterns

Overview

- When R cannot work out the format of a data set it tends to convert it into mode character.
- It's therefore very useful to be able to manipulate character vectors.
- Rather than go through these all individually (there are lots of them)
- see the screencast associated with this section for a longer review.

Searching and splitting

- `grep(pattern,x)` finds the specified pattern inside the vector `x`
- `nchar(x)` finds the length of the string `x`. Note that it includes spaces (and has some funny behaviour for non character variables)
- `substr(x, start, stop)` extracts the part of the string from value `start` to value `stop`.
- `strsplit(x, split)` splits the current string into a list of substrings by the character(s) in `split`.

Printing/concatenating strings

- `paste(...)` will concatenate several strings together, with the result returned as one long string. The extra argument `sep` gives the character by which to join them (by default just a space).
- `sprintf(...)` assembles strings from parts in a formatted manner. It is very similar to C functions used to print out strings.

Finding patterns

- `regexp(pattern, text)` gives the character position of the first occurrence of `pattern` in `text`, as well as the length of the `pattern`.
- `gregexp(pattern, text)` gives all the instances of pattern in text (and also the length of the pattern).
- `gsub(pattern, replacement, text)` will replace the values in `replacement` with `pattern` in object `text`

Regular expressions

Introduction Some examples

An introduction to regular expressions

- When manipulating strings, it is often very helpful to be able to flexibly find and replace elements of a string.
- A regular expression is a special form of syntax that allows you to 'pattern match' in a very broad way.
- They are often complicated to read and we do not cover them in any real detail.
- However, it is worth understanding a few basics.
  - You can put a regular expression into `grep`, `regexp`, `gregexp` and many other functions.

Example regular expressions

- A regular expression of the form `'[au]'` finds strings with either `a` or `u` in them

```
grep('[au]', c('Equator', 'North Pole', 'South Pole'))
```

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

- A regular expression of the form `'[o.e]'` finds strings with the letter `o` and `e` separated by any other character:

```
grep('o.e', c('Equator', 'North Pole', 'South Pole'))
```

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

```
grep('N..t', c('Equator', 'North Pole', 'South Pole'))
```

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

The `.` is a wildcard character

Example regular expressions 2

- If you wish to search for a `.` in a string, you need to be careful:

```
grep('.', c('abc', 'de', 'f.g'))
```

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

```
grep('\\.', c('abc', 'de', 'f.g'))
```

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

- You can use ^ to search for the beginning of a string and $ for the end

```
state.name[grep('ana$', state.name)]
```

```
## [1] "Indiana"   "Louisiana" "Montana"
```

```
state.name[grep('^South', state.name)]
```

```
## [1] "South Carolina" "South Dakota"
```

Example regular expressions 3

- You can also do this with whole words rather than just characters:

```
words <- c('cat', 'bat', 'dog', 'rat')
grep('bat|cat', words)
```

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

```
words[grep('(b|c)at', words)]
```

```
## [1] "cat" "bat"
```

```
words[grep('(b*|c*)at', words)]
```

```
## [1] "cat" "bat" "rat"
```

- The * qualifier here specifies 'at least zero' of this b or c but not both.

Examples

Creating files names Listing files of a certain type Web scraping

Example 1: Creating file names

- Create some histograms of normally-distributed data with mean 0 and standard deviation i, create histograms and then store them in a file.

```
for(i in 1:5) {
  fname <- paste('N(0,', i, ').pdf', sep = '')
  pdf(fname)
  hist(rnorm(100, sd = i))
  dev.off()
}
```

- Note that paste here is being used (with sep = '') to create the filenames

Example 2: List files of certain types

- List the files in a chosen directory which follow a certain regular expression

```
list.files(getwd(), '\\.txt$')
```

```
##  [1] "d1.txt"            "d2.txt"            "d3.txt"
##  [4] "d4.txt"            "file1.txt"         "file2.txt"
##  [7] "file3.txt"         "file4.txt"         "prostate.info.txt"
## [10] "z1.txt"            "z2.txt"            "z3.txt"
## [13] "z4.txt"            "z5.txt"            "z6.txt"
```

- This lists the files in the working directory which end with `.txt`.

Example 3: web scraping

- Download the names of the contributors to the R stats mailing list and calculate how often they give help

Example 3: web scraping


```
library(RCurl)
oct17 <- getURL("https://stat.ethz.ch/pipermail/r-help/2017-October/date.html")
webpage <- strsplit(oct17, "\n")[[1]] # download the webpage see slide 21
# get the lines that contains the authors
# [the authors are in italic, so html code starts with "<I>"]:
authorsraw <- grep("<I>", webpage, value = TRUE)
authors <- gsub("<I>", "", authorsraw, fixed = TRUE) # only keep the names
# create a table that contains the number of contributions for each author,
# and sort it in decreasing order:
author_counts <- sort(table(authors), decreasing = TRUE)
author_counts[1:5]
```

```
## authors
## David Winsemius     Bert Gunter     Eric Berger  Jeff Newmiller
##              32              23              23              23
##      PIKAL Petr
##              14
```

- See the screencast for a walkthrough of this example.

Lessons from this week

- `scan` and the other `read.XXX` functions are really useful for getting access to lots of different data sets.
- Getting data from the web is easy if it's already formatted for R (we can use `read.table` or similar), though there are packages that makes things simpler.
- Getting non-formatted data off the web usually requires some string manipulation techniques such as `grep`, `gsub`, etc.