

Input & Output

Input & Output

- ◉ This is a very important aspect which allows students to work with rich, complex data from many different formats.
- ◉ We want to be able to both
 - ◉ read data from various domains & sources
 - ◉ generate new types of content/output for presenting statistical results, often graphically with interactive applications such as Google Maps, Google Earth, "AJAX", Flash/SVG, Web forms, and
 - ◉ be part of the visual analytics pipeline.

Topics

also

- ◉ We'll talk about
 - ◉ text
 - ◉ tools for regular formats
 - ◉ complex, irregular formats
 - ◉ patterns and regular expressions
 - ◉ combining specialized tools (e.g. shell tools)
 - ◉ binary formats
 - ◉ character encoding

- ◉ Databases
- ◉ XML - eXtensible Markup Language
standard formats & scraping HTML pages
- ◉ JSON - JavaScript Object Notation
- ◉ HTTP requests and dynamic queries
- ◉ Mastering these will leave a student in a very good position to be able to work with all sorts of data and are increasingly necessary for working in impact areas,
climate, bioinformatics, visual analytics, ...

Self-describing & schema

- ◉ One of the main differences between {databases, XML, JSON} and simple text files is that the former are self-describing
- ◉ Software can read the file and identify which values are real, integer, factors, strings; what are the variables, what is the encoding; what identifies missing-values.
- ◉ These data sources have a schema definition of the structure and content
- ◉ Text files require the user to specify the data types, whether the first row gives the names of the variables, whether there are row names or not, ...

Perspective

- ◉ Again, it is important to try to get the students
 - ◉ to compare these different formats and
 - ◉ understand why each of them exists,
 - ◉ understand their history and evolution,
 - ◉ when one should be preferred over another
 - ◉ what are possible future innovations and evolutions
 - ◉ the role of standards.
- ◉ Students can master each, but need to see the higher-level view.

This session

- ◉ Talk about R's input and output facilities.
- ◉ Mostly covered in
 - ◉ R Data Import/Export manual (R-data.html)
 - ◉ Data Manipulation with R (Phil Spector)
 - ◉ Most R books.

- ◉ Coarse grained:
 - functions for reading entire content of a document atomically.
- ◉ Finer grained:
 - Connections - user programmable "streams" from which we can extract/insert content at resolution of bytes, values, lines, ...
 - Allow us to go past record-per-line or read entire data, but can skip around the content in dynamic, context-specific manner.

- Connections also provide us a way to work with sources other than local text files, e.g.
 - compressed files
 - URLs
 - pipes to & from other processes/running applications
 - data given as literal text, e.g. computed dynamically & so avoiding the file system
 - sockets & FIFOs

Reading regular text files

- Rectangular/tabular data generated from other applications in a form that can be read by most applications
 - each record on its own line
 - each line has values for the same number of variables, in the same order
 - one value per variable
- Regular structure with only the delimiter needing to be specified - `,` or (invisible) TAB or space
- `read.table(file, sep = "my delimiter")`
- Maps to a data frame in R. (Use `scan` for `matrix()`)

Extensions

- Variable names on first row
- Types of the variables
(1, 2, 2, 3 - factor, integer or reals that happen to be integers)
- Comments within the file
- Missing values
- Encoding

Fixed width Format

- Record per line
- Each variable starts in the same column/position in each line and has a fixed maximum width.
- No separator/delimiter
- Application that writes this format needs to know all the values ahead of time, or at least the maximum number of characters for each value for each variable.

```
read.table(file, header = FALSE, sep = "", quote = "\"\"",
  dec = ".", row.names, col.names,
  as.is = !stringsAsFactors,
  na.strings = "NA", colClasses = NA, nrows = -1,
  skip = 0, check.names = TRUE,
  fill = !blank.lines.skip,
  strip.white = FALSE, blank.lines.skip = TRUE,
  comment.char = "#",
  allowEscapes = FALSE, flush = FALSE,
  stringsAsFactors = default.stringsAsFactors(),
  encoding = "unknown")
```

- ⦿ Important to use colClasses
 - ⦿ improves speed
 - ⦿ avoid implicit coercion
 - ⦿ can skip variables
- ⦿ Default turns character variables into factors.
 - ⦿ control with colClasses or stringsAsFactors = FALSE

- ⦿ skip & nrows parameters of read.table allow one to read arbitrary contiguous blocks of a "file".
- ⦿ Can use this to read chunks of the file in sequence and do computations on these smaller chunks.
 - ⦿ one approach to dealing with massive data.

Richer content

- ⦿ read.table("filename", ...) is "atomic"
 - reads all of the relevant portion of the file in the specified manner/format and returns the result as a data frame.
- ⦿ There is no opportunity for
 - ⦿ dynamically interpreting the content/format
 - ⦿ working with something other than a data frame.

Non-rectangular data

- Consider longitudinal data
- John Doe, 28, 123-45-6789
100 101 98 . 100
27 34 45 . .
- `read.table()` won't do
only for the "body" of the record, 2nd & 3rd lines
- So have to use customized input code.

- `readLines()` to get all the lines in the file as a character vector
then group elements for individual records and interpret.
- Alternative is to read lines for a single record
process these
move on to next record.
- Non-atomic "input" operation
loop over { read, process }
- Want to be able to traverse the input stream and
interleave additional R code.
- Use connections

Connections

- Connection is a data stream with a persistent (across separate accesses) of where the current point/mark/cursor is.
- Abstract concept that has files, compressed files, URLs, pipes, FIFOs, sockets and in memory text strings as specific types.
- Open and close connections
consume input and move the cursor.
- Pass an open connection to a function, it leaves it open
Pass a closed connection, it opens & then closes it.

- Get an open stream for reading data from a URL
`con = url("http://eeyore/data/longitudinal", "r")`
- Read 3 lines
`lines = readLine(con, n = 3)`
`records[[i]] = createRecord(lines)`
`i <- i + 1`
- And loop until at the end of the stream
e.g. `length of lines < 3`

Using functions with connections

- How do we extract the values from the 3 lines? like to use `scan()`, `read.table()`, and not write special tools to recognize reals, integers, factors, ...
- We have the content for a record as 3 strings want to use something like
`scan(lines, what = "", strip.white = TRUE, nmax = 3)`
- But `scan()` will treat lines as the name of a file
- So use a `textConnection()`
`con.tmp = textConnection(lines)`
- Now reading the sequence/stream of characters

- The values of the repeatedly measured variables are in tabular form
- We can continue to read from the open text connection, by passing it to
`read.table(tmp.con, na.strings = ".")`
- Now we have the elements of our record (What data structure we use to store them is another matter!)
- But must close the text connection so that it is garbage collected.

- There are a fixed number of open connections at any time (1024) and if you don't close them, they will accumulate and eventually can't open another.
- So when writing functions, make certain to obey the close if connection was closed on entry use `on.exit()` to guarantee this.
- The concept of a connections applies to both input and output, and both count to the total number allowed.

String manipulation

- If the structure of the data are not amenable to `scan()` and `read.table()`, we have to break the text up ourselves.
- e.g. the Mannheim wireless data

```
†=1139692477303;id=00:02:2D:21:0F:33;pos=0.0,0.05,0.0;degree=130.5;00:14:bf:bl:97:8a=-43,2437000000,3;00:0f:a3:39:e1:c0=-52,2462000000,3;00:14:bf:3b:c7:c6=-62,2432000000,3;00:14:bf:bl:97:81=-58,2422000000,3;00:14:bf:bl:97:8d=-62,2442000000,3;00:14:bf:bl:97:90=-57,2427000000,3;00:0f:a3:39:e0:4b=-79,2462000000,3;00:0f:a3:39:e2:10=-88,2437000000,3;00:0f:a3:39:dd:cd=-64,2412000000,3;02:64:fb:68:52:e6=-87,2447000000,1;02:00:42:55:31:00=-85,2457000000,1
```
- Elements are separated by ;
within this have named variables given by `name=value`,
value is different real, triple of reals, triple of integers

- Regular expressions are very powerful for this sort of data.
- However, can do the work with string manipulation tools.
- Good to get students started early in a course and a motivation for more general language for expressing text patterns.

- Read the text,
split the string based on ';'
process each element, by breaking the string on '='
process each value based on context (the variable name)
- `strsplit(character-vector, pattern = ";")`
(pattern can be a regular expression)
- pattern can also be "" to get individual characters
e.g. `strsplit("abc", "") --> c("a", "b", "c")`

- Also, `nchar()`, `substring()`
- `paste()` to concatenate strings back together
Often needed when `strsplit()` using a character that occurs in more than one place.
- And even though they are used for regular expressions, the functions
`grep()`, `regexpr()`, `gsub()`
can be used with `fixed = TRUE` to apply to literal pattern matching rather than treating the contents of the pattern as from the regular expression language.

Common Formats & Data Exchange

- ◉ We often want to use two or more of R, S-Plus, SAS, MATLAB, State, SPSS, ...
- ◉ These store their data in their own special binary format.
- ◉ Want to read the contents directly without having to have these applications running
 - ◉ Could read with low-level R tools (connections & readBin())
- ◉ or use package foreign to read and write data from/to appropriate format.
 - ◉ read.spss(), read.dta(), ...
 - ◉ write.foreign()

Data summaries for reading data

- ◉ When reading data, students need to validate the results,
 - ◉ i.e. that the resulting R object matches the data in the source.
- ◉ How do we verify?
 - ◉ random samples
 - ◉ looking at summaries from both sources
 - ◉ e.g. number of records, counts of particular word,
 - ◉ plots
 - ◉ look at NAs

- Uses different approaches to getting this validating information.
- Use shell tools via `system()` such as
 - `wc -l`
 - `grep | sort | uniq -c`
 - `head -n 20 file | tail -n 1`

Output

- Generate output from R
 - messages on the console (messages to user, simple debugging)
 - saving R objects for use in a different session
 - to share data/values with another application
 - to create input content for another application
 - e.g. HTML pages,
 - Google Earth, SVG,
 - commands for MATLAB, Python, ...

Output to Console

- In functions, instead of writing message to console, use `warning()` or `stop()` so that these can be managed by the user.
- `print()` and `cat()` used for messages on console
 - `print()` for objects
 - `cat()` for control over text.
- Both can write to a connection (open or closed)
 - `cat(..., file = <connection>)`
- But `print()` uses `stdout()`, so use `sink()` to redirect standard out to a connection and then `print()`, and reset `sink(stdout())`

Saving R objects

- `cat()` is flexible, but sometimes create text first by computations to yield a character vector with the correct format.
- C-like tools for formatting numbers, strings, `formatC()`, `sprintf()`
 - Padding, decimal places, etc.
- Often need to create content as collection of strings and then rearrange and write to stream, rather than writing each bit as we see it.
- Also, good to avoid writing objects to console in a function, but rather return data structure with its own print method.

- Three ways to store/serialize R objects
 - `save()`,
 - `dput()`, `dump()`
- `save()` has different formats
 - ASCII or R's own binary format based on XDR (external data representation)
 - XDR is machine independent (i.e. deals with byte endianness) and almost always one can transfer an XDR file from one machine to another and be able to `load()` it directly into R.

Generating Content for Apps

- Use higher level of abstraction than just writing out the content character/element at a time.
- Compose an object and have a serialization method for the target format.
- HTML - R2HTML package
- XML - XML package
- But use a document model which serializes to e.g. HTML via the R2HTML package rather than generating the HTML directly.

Binary files

- Binary files are often used for efficient, self-describing data,
e.g. R's XDR format
- Either format is known ahead of time and code to read and write it is developed using this "schema"
- Or dynamically determine what is coming next in the stream of bytes
e.g. read integer which tells us whether the next value is a real or an integer, and read 8 or 4 bytes respectively to get the value.

- Almost always need to use connections in order to maintain location in the stream and interleave code to interpret the contents.
- Open connection with `file(..., open = "rb")` (or explicitly call `open()`)
rb - read and binary
- Alternatively, can read from a "raw" vector where the bytes of the stream are in memory (like a text connection)
- `readBin(con, what = ...)` and `readChar()`

Other Issues

- Portability
 - `\r\n` or `\n` for line endings
 - file systems, paths (with spaces), `~` and locating files locally
 - use of packages and `system.file()` is good way to make data available via invariant access.
 - Encoding and internationalization.
Latin-1, Unicode, UTF-8