Character Manipulation in R

Clay Ford

Spring 2017

Workshop agenda

- How to recognize character data in `R`
- Character data vs Factor data
- Base `R` functions for working with character data
- Intro to Regular Expressions
- Intro to `stringr`, `qdapRegex` and `lubridate` packages
- Work through three extended examples

Examples of character manipulation

- Getting rid of whitespace (`" Texas "`)
- Converting text to UPPERCASE or lowercase
- Extract parts of a string (Eg, get extract "23" from "23")
- Split "`First Last`" into "`First`" and "`Last`"
- Combine "`First`" and "`Last`" into "`First Last`"
- Identify patterns of text for purpose of extracting or subsetting data

Character data in R

- Character data have quotes when printed to console
- But data with quotes does not mean it's character!
- use `is.character()` to find out.
- Character data need to be surrounded with quotes (either single or double) when used in `R` code

```
(x <- c("a","b","c","12"))
```

```
## [1] "a"  "b"  "c"  "12"
```

```
is.character(x)
```

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

Beyond letters and numbers

Character data includes apostrophes, quotes, other punctuation, line breaks, carriage returns, spaces, and tabs. These can look different depending on the system we use and whether we're printing them to the console or entering them in an R script.

```
name <- '"Tiny Rick" Sanchez
Morty Smith
D\'wayne'
name
```

```
## [1] "\"Tiny Rick\" Sanchez\nMorty Smith\nD'wayne"
```

Beyond letters and numbers

Some guidelines:

- Use single quotes around text that include double quotes (and vice versa)
- Or "escape" quotes with a backslash if being entered within quotes of the same type
- You can enter line breaks, but they appear as "`\n`" when printed to console
- see `?Quotes` for much more information

Viewing character with `cat`

We can use the `cat` function to view character data in a "normal" way where things like line breaks are not printed.

```
cat(name)
```

```
## "Tiny Rick" Sanchez
## Morty Smith
## D'wayne
```

We can also do `writeClipboard(name)` to copy the text to the clipboard and paste into a text document to review.

Character versus factor

- Sometimes data that appear to be character are actually stored as a *factor*
- factors are character data that are stored as integers but have character labels
- factors are good for using character data in statistical modeling (eg, ANOVA, regression, etc)
- If your character data is stored as a factor, R automatically handles conversion to dummy matrices necessary for statistical modeling routines
- factors do not have quotes when printed to console

Factor data in R

```
(y <- factor(c("a","b","c","c")))
```

```
## [1] a b c c
## Levels: a b c
```

```
is.character(y)
```

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

Why should you care about factors?

- Base `R` functions such as `read.csv` and `data.frame` will automatically convert character data to factor
- Set argument `stringsAsFactors = FALSE` to turn off
- Many R functions intended for character data will work on factors by *quietly* coercing the data to character! Example:

```
y
```

```
## [1] a b c c
## Levels: a b c
```

```
# find "a", substitute "z"
(y <- sub("a","z",y))
```

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

Final word on factors (in this workshop)

- Be aware of the structure of your data. Use `str` for this.
- If you plan to clean or manipulate character data, make sure it's character, not factor.
- Change factor to character with `as.character` function

```
(y <- factor(c("a","b","c","c")))
```

```
## [1] a b c c
## Levels: a b c
```

```
(y <- as.character(y))
```

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

Basic character manipulation functions

These are functions available "out-of-the-box" with base `R`.

- `toupper`, `tolower`: convert all characters in a vector to UPPERCASE or lowercase
- `trimws`: trim whitespace around character data
- `substr`: extract a substring of character data based on position
- `paste`, `paste0`: combine two or more vectors of character data
- `sub`, `gsub`: search-and-replace text
- `strsplit`: split a string into parts
- `nchar`: count number of characters in a string

Examples - toupper and tolower

```
state <- c("Virgina", "Maryland", "Delaware")
tolower(state)
```

```
## [1] "virgina"  "maryland" "delaware"
```

```
toupper(state)
```

```
## [1] "VIRGINA"  "MARYLAND" "DELAWARE"
```

Examples - trimws

```
# \n is a newline
(response <- c(" Yes  ", "  No ", "   No way \n"))
```

```
## [1] " Yes  "       "  No "        "   No way \n"
```

```
trimws(response)
```

```
## [1] "Yes"    "No"     "No way"
```

Examples - substr

Extract a substring based on starting and stopping position.

```
id <- c("VA0001", "VA0002", "VA0003")
substr(id, start = 3, stop = 6)
```

```
## [1] "0001" "0002" "0003"
```

Can also identify starting and stopping position of the substring to replace.

```
substr(id, start = 1, stop = 2) <- "ID"
id
```

```
## [1] "ID0001" "ID0002" "ID0003"
```

Examples - paste

paste concatenates items in vectors.

```
first <- c("Geddy","Alex","Neil")
last <- c("Lee","Lifeson","Peart")
paste(first, last)
```

```
## [1] "Geddy Lee"    "Alex Lifeson" "Neil Peart"
```

It uses a space as a separator by default. We can specify a separator with the sep argument.

```
paste(last, first, sep = ", ")
```

```
## [1] "Lee, Geddy"    "Lifeson, Alex" "Peart, Neil"
```

Examples - paste and paste0

Use the collapse argument to collapse multiple elements into one element.

```
(time <- c("10", "12", "58"))
```

```
## [1] "10" "12" "58"
```

```
paste(time, collapse = ":")
```

```
## [1] "10:12:58"
```

paste0 is a convenience function for pasting without a separator.

```
st <- c("VA","TX","CA"); id <- 1:3
paste0(st, id)
```

```
## [1] "VA1" "TX2" "CA3"
```

Examples - sub and gsub

Find a string and substitute another string. sub only does the first match while gsub (global) does all matches.

```
trt <- c("trt_1_a", "trt_1_b", "trt_2_a")
sub(pattern = "_", replacement = "", trt)
```

```
## [1] "trt1_a" "trt1_b" "trt2_a"
```

```
gsub("_", "", trt)
```

```
## [1] "trt1a" "trt1b" "trt2a"
```

Note: sub and gsub assume the pattern is a *regular expression*. More on that later.

Examples - strsplit

Split a character string at a character of your choice.

```
talk <- c("Hello.\nHi.\nHow are you?\nOK.",
          "We should go.\nGood idea.")
strsplit(talk, split = "\n")
```

```
## [[1]]
## [1] "Hello."       "Hi."          "How are you?" "OK."
##
## [[2]]
## [1] "We should go." "Good idea."
```

Notice it returns a list object. This can make `strsplit` results challenging to work with.

Examples - `strsplit` with `unlist`

The `unlist` function "unwraps" a list into a single vector. This can come in handy.

```
unlist(strsplit(talk, split = "\n"))
```

```
## [1] "Hello."         "Hi."              "How are you?"   "OK.
## [5] "We should go." "Good idea."
```

Examples - nchar

Count the number of characters in a string. Count includes spaces and punctuation. Notice NA is not counted.

```
k <- c("W. Main St", "Water St.", NA, "5th St SW")
nchar(k)
```

```
## [1] 10  9 NA  9
```

This can be useful for error checking or subsetting data. For example, check that all state abbreviations are length 2.

```
all(nchar(state) == 2)
```

Working with character patterns

Sometimes we need to identify or extract character data that matches a certain pattern. Examples:

- email addresses
- two-character sequences of capital letters (AL, AK, etc)
- ALLCAP words ending in a : (CLINTON:, SANDERS:, etc.)
- text in between HTML tags or in parentheses
- word variations (cry, crying, cried, cries)

We use *regular expressions* to define these patterns.

Regular Expressions

- Regular expressions are a language for describing text patterns
- A regular expression is usually formed with some combination of *literal characters*, *character classes* and *modifiers*
  - literal character example: `state` (looking for `"state"`)
  - character class example: `[0-9]` (any number 0 - 9)
  - modifier example: `+` (1 or more of whatever it follows)
- Regular expression example: `state[0-9]+` finds patterns such as `state1`, `state12`, `state99` but not `state`

More on Regular Expressions

- Regular expressions are powerful and can be quite complex
- Many programming languages have their own implementation of regular expressions
- We will cover just the basics today as they work in R

Character classes

- `[0-9]`, `[a-z]`, `[A-Z]`
- Define your own: `[0-3a-g]`, `[AEIOUaeiou]`
- Predefined character classes
  - `[:alpha:]` all letters
  - `[:digit:]` numbers 0 - 9
  - `[:alnum:]` Alphanumeric characters (alpha and digit)
  - `[:blank:]` Blank characters: space and tab
  - `[:lower:]` lowercase letters
  - `[:upper:]` UPPERCASE letters
  - `[:punct:]` Punctuation characters
  - `[:print:]` Printable characters: [:alnum:], [:punct:] and space
  - `[:space:]` Space characters: tab, newline, vertical tab, form feed, carriage return, space

Modifiers

- `^` start of string
- `$` end of string
- `.` any character except new line
- `*` 0 or more
- `+` 1 or more
- `?` 0 or 1
- `|` or (alternative patterns)
- `{}` quantifier brackets: exactly `{n}`; at least `{n,}`; between `{n,m}`
- `()` group patterns together
- `\` escape character (needs to be escaped itself in R! `\\`)
- `[]` character class brackets

Note: precede these with a double backslash if you want to treat them as literal characters.

Shorthand character classes

- `\d` is for "digit"; short for `[0-9]`
- `\w` is for "word character"; short for `[A-Za-z0-9_]`
- `\s` is for "whitespace character"; short for `[\t\r\n\f]`

Negated versions:

- `\D` is short for `[^\d]`
- `\W` is short for `[^\w]`
- `\S` is short for `[^\s]`

Recall: The backslash in `R` has to be escaped itself. Hence all these need `\\`

Word boundaries

The metacharacter `\b` matches word boundaries. It allows us to search for whole words or numbers.

The regex `"red"` matches `"red"`, `"redder"`, and `"Fred"`.

The regex `"\\bred"` matches `"red"` and `"redder"` but not `"Fred"`.

The regex `"\\bred\\b"` matches `"red"` but not `"redder"` or `"Fred"`.

Lookahead and Lookbehind

When we want to match words or patterns that come (or don't come) before or after certain words, we can use lookahead and lookbehind. For example, match "done" if it does not follow "almost".

- Lookahead `(?=foo)` What follows is `foo`
- Lookbehind `(?<=foo)` What precedes is `foo`
- Negative Lookahead `(?!foo)` What follows is not `foo`
- Negative Lookbehind `(?<!foo)` What precedes is not `foo`

Note: We can also match regular expressions in lookahead but not lookbehind.

Learning more about regex

- There is MUCH more to regular expressions!
- Google is your friend when learning regex or creating a regular expression
- See `http://www.regular-expressions.info/` for a free and well-done tutorial
- See `https://regex101.com/` where you can build and test regexes

Let's look at a few examples and introduce some new functions.

grep and grepl

grep and grepl search for patterns in strings. grep returns indices of matches while grepl returns a logical vector. Example: find strings containing ".x"

```
text <- c("1", "1.x", "2", "2.x", "3.1")
grep(pattern = "\\.x", x = text)
```

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

```
grepl(pattern = "\\.x", x = text)
```

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

Notice we have to "escape" the period with two backslashes: \\.

grep and grepl assume regex patterns

By default grep and grepl (and sub and gsub) assume the pattern you give it is a regular expression. Set fixed = TRUE to make them interpret the pattern literally.

```
text <- c("1", "1.x", "2", "2.x", "3.1")
grep(pattern = ".x", x = text, fixed = TRUE)
```

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

```
grepl(pattern = ".x", x = text, fixed = TRUE)
```

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

More on grep

Setting `value=TRUE` will return the string *containing* the matching pattern.

```
grep(pattern = "\\.x", x = text, value = TRUE)
```

```
## [1] "1.x" "2.x"
```

Setting `invert=TRUE` and `value=TRUE` will return the string *not containing* the pattern.

```
grep(pattern = "\\.x", x = text, value = TRUE,
     invert = TRUE)
```

```
## [1] "1"   "2"   "3.1"
```

regex with sub and gsub

We can also use regular expressions with sub and gsub. Below we find the period and any letter or number following it, and replace with nothing.

```
id <- c("1.x","1.2","1.01","2.1","2.x")
sub(pattern = "\\.[[:alnum:]]+", "", id)
```

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

Here we get rid of all punctuation.

```
text <- "No? You sure?! OK. Good luck!"
gsub(pattern = "[[:punct:]]", "", text)
```

```
## [1] "No You sure OK Good luck"
```

A few more regex examples

Find file names ending in ".csv"

```
files <- c("pre.csv.R","arrests.csv","lcsv2.jpg")
grep(pattern = "\\.csv$", files, value = TRUE)
```

```
## [1] "arrests.csv"
```

Find all instances of “cry” such as “crying”, “cried”, “cries”

```
resp <- c("crying", "he cried", "encrypted",
          "I wanted to cry", "increase")
grep(pattern = "\\bcr(y|ying|ied|ies)\\b", resp)
```

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

A few more regex examples

If what follows is " x" (lookahead), replace with "---"

```
x <- c("129 x", "128", "130 x", "x 131")
sub(pattern = "[0-9]{3}(?= x)", "---",x , perl = TRUE)
```

```
## [1] "--- x" "128"   "--- x" "x 131"
```

If what precedes is "x " (lookbehind), replace with "999"

```
sub(pattern = "(?<=x )[0-9]{3}", "999", x , perl = TRUE)
```

```
## [1] "129 x" "128"   "130 x" "x 999"
```

Notice we had to set `perl=TRUE` to use Perl-compatible regular expressions!

R packages for character manipulation

Some character manipulation tasks are so common that others have developed R packages that provide functions to easily carry out the tasks.

Three such packages:

- `stringr` - Simple, Consistent Wrappers for Common String Operations
- `qdapRegex` - Regular Expression Removal, Extraction, and Replacement Tools
- `lubridate` - Make Dealing with Dates a Little Easier

The `stringr` package

- "a clean, modern interface to common string operations"
- Comes with a thorough but easy-to-follow vignette
- The main stringr functions all begin with `str_`
- `stringr` functions are actually wrappers for functions from the `stringi` package!

stringr function: str_extract and str_extract_all

str_extract and str_extract_all extract matching patterns from a string. The former extracts the first match while the latter extracts all matches. Example: extract year.

```
library(stringr)
date <- c("4-5-1973", "6 Sept 1987", "Dec 12, 2012")
str_extract(date, pattern = "[0-9]{4}$")
```

```
## [1] "1973" "1987" "2012"
```

str_extract vs. grep

You might think using grep with value = TRUE would be the same as str_extract. It's not.

```
date <- c("4-5-1973", "6 Sept 1987", "Dec 12, 2012")
grep(pattern = "[0-9]{4}$", date, value = TRUE)
```

```
## [1] "4-5-1973"     "6 Sept 1987"  "Dec 12, 2012"
```

str_extract *extracts* the match; grep with value=TRUE returns the string that *contains* the match.

stringr function: str_pad

str_pad pads a string with the character you specify. Example: pad id with leading 0s so all ids have three digits.

```
id <- c("8","19","101","144")
str_pad(id, width = 3, side = "left", pad = "0")
```

```
## [1] "008" "019" "101" "144"
```

stringr function: str_sub

str_sub extracts and replaces substrings from a character vector. It's equivalent to substr but also accepts negative positions, which are calculated from the left of the last character. Example: extract the last 4 characters of a string.

```
date <- c("4-5-1973", "6 Sept 1987", "October 12, 2013")
str_sub(date, start = -4)
```

```
## [1] "1973" "1987" "2013"
```

There are several other functions in the stringr package. Check out the vignette!

The qdapRegex package

- A collection of regular expression tools associated with the qdap package
- Works fine as a standalone package
- Functions for the removal/extraction/replacement of abbreviations, dates, dollar amounts, email addresses, hash tags, numbers, percentages, citations, person tags, phone numbers, times, and zip codes
- It has no vignette but the documentation has many good examples

qdapRegex function: ex_between

ex_between extracts strings between two markers. It returns a list, so we often use it in conjunction with unlist

```
library(qdapRegex)
rev <- c("(100)", "(215)", "(-400)")
unlist(ex_between(rev, left = "(", right = ")"))
```

```
## [1] "100"  "215"  "-400"
```

qdapRegex function: ex_mail

ex_email extracts email addresses.

```
text <- c("mailto:jcf2d@virginia.edu",
          "Doe, Jon (jd3z) jd3z@virginia.edu")
unlist(ex_email(text))
```

```
## [1] "jcf2d@virginia.edu" "jd3z@virginia.edu"
```

qdapRegex has many functions like this.

Working with dates

- Dates and times are often read in as character data. We usually want them formatted as a `Date` class so we can calculate things like elapsed time.
- The `lubridate` package makes it easy to convert character dates to `Date` class
- Use any permutation of `m`, `d`, `y` as a function to indicate order of month, day and year.

```
library(lubridate)
Dates <- c("12-12-2001","1/7/2004","Oct 7, 2008")
(Dates <- mdy(Dates))
```

```
## [1] "2001-12-12" "2004-01-07" "2008-10-07"
```

- The dates are now stored as number of days since Jan 1, 1970.

More on lubridate

- You can also read in hours, minutes and seconds using `h`, `m` and `s` either by themselves or with `mdy` (following an underscore) .

```
Times <- c("2017-02-06 02:23:12", "2017-02-07 09:54:18")
(Times <- ymd_hms(Times))
```

```
## [1] "2017-02-06 02:23:12 UTC" "2017-02-07 09:54:18 UTC"
```

```
is.character(Times)
```

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

- These dates are stored as number of seconds since Jan 1, 1970.
- See the `lubridate` vignette for a great intro to the package.

Let's go to R!

For the remainder of the workshop we'll work in RStudio, demonstrating what we covered in the slides. We'll also introduce more functions and strategies for manipulating character data.

References and further reading

- Sanchez, G. (2013). *Handling and Processing Strings in R*. Trowchez Editions. `http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf`
- Spector, P. (2008). *Data Manipulation in R*. Springer.
- Teetor, P. (2011). *R Cookbook*. O'Reilly.
- Li, G. and Bryan, J. (2014). *Regular Expressions in R*. `http://stat545.com/block022_regular-expression.html`
- Regular-Expressions.info: `http://www.regular-expressions.info/`
- Regex101: `https://regex101.com/`
- Google "regex cheat sheet"

Thanks for coming today!

For help and advice with your statistical analysis:
`statlab@virginia.edu`

Sign up for more workshops or see past workshops:
`http://data.library.virginia.edu/training/`

Register for the Research Data Services newsletter to stay up-to-date on RDS events and resources:
`http://data.library.virginia.edu/newsletters/`