Data Management With R: Working with Strings

Matthias Haber

30 October 2017

Final project

Datasets

Prerequisites

Packages

```
library(tidyverse)
library(stringr) #install.packages("stringr")
```

Data

```
url <- paste0("https://raw.githubusercontent.com/mhaber/",
              "HertieDataScience/",
              "master/slides/week7/data/")
films <- read_csv(paste0(url, "films.csv"))
people <- read_csv(paste0(url, "people.csv"),
                   col_types = "iccc")
reviews <- read_csv(paste0(url, "reviews.csv"))
roles <- read_csv(paste0(url, "roles.csv"))
#flights
```

Strings

Creating strings

Strings are wrapped in ' or " quotes:

```
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string
```

You can use \ to "escape" single or double quotes inside a string:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

Special characters

- `\n` newline
- `\r` carriage return
- `\t` tab
- `\b` backspace
- `\a` alert (bell)
- `\f` form feed
- `\v` vertical tab
- `\\` backslash \

Functions for strings

```r
# Length
str_length("Data Management with R")
```

```
## [1] 22
```

```r
# Combining strings
str_c("Data Management","with R", sep = " ")
```

```
## [1] "Data Management with R"
```

Functions for strings

```
#Subsetting strings
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
## [1] "App" "Ban" "Pea"
```

```
str_sub(x, -3, -1)
```

```
## [1] "ple" "ana" "ear"
```

```
# Changing case
str_to_upper(c("a", "b"))
```

```
## [1] "A" "B"
```

```
str_to_lower(c("A", "B"))
```

Regular expressions

Regular expressions

Regular Expressions (regex) are a language or syntax to search in texts. Regex are used by most search engines in one form or another and are part of almost any programming language. In R, many string functions in `base` R as well as in `stringr` package use regular expressions, even Rstudio's search and replace allows regular expression.

You could use regex to e.g.:

- Count the occurence of certain persons/organization etc. in text
- Calculate the sums of fund discussed in legislation
- Chose your texts based on regexes

In textpreparation, regex are used to remove certain unwanted parts of text.

Regular expression syntax

Regular expressions typically specify characters to seek out, possibly with information about repeats and location within the string. This is accomplished with the help of metacharacters that have specific meaning:

- `$ * + . ? [ ] ^ { } | ( ) \`.

String functions related to regular expression

- `grep(..., value = FALSE)`, `grepl()`, `stringr::str_detect()` to identify match to a pattern
- `grep(..., value = TRUE)`, `stringr::str_extract()`, `stringr::str_extract_all()` to extract match to a pattern
- `regexpr()`, `gregexpr()`, `stringr::str_locate()`, `string::str_locate_all()` to locate pattern within a string
- `sub()`, `gsub()`, `stringr::str_replace()`, `stringr::str_replace_all()` to replace a pattern
- `strsplit()`, `stringr::str_split()` to split a string using a pattern

Quantifiers

Quantifiers specify the number of repetitions of the pattern.

- `*`: matches at least 0 times.
- `+`: matches at least 1 times.
- `?`: matches at most 1 times.
- `{n}`: matches exactly n times.
- `{n,}`: matches at least n times.
- `{n,m}`: matches between n and m times.

Quantifiers (II)

```
strings <- c("a", "ab", "acb", "accb", "acccb", "accccb")
grep("ac*b", strings, value = TRUE)
```

```
## [1] "ab"     "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac+b", strings, value = TRUE)
```

```
## [1] "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac?b", strings, value = TRUE)
```

```
## [1] "ab"  "acb"
```

```
grep("ac{2}b", strings, value = TRUE)
```

```
## [1] "accb"
```

Position of pattern within the string

- `^`: matches the start of the string.
- `$`: matches the end of the string.
- `\b`: matches the empty string at either edge of a *word*. Don't confuse it with `^ $` which marks the edge of a *string*.
- `\B`: matches the empty string provided it is not at an edge of a word.

```
(strings <- c("abcd", "cdab", "cabd", "c abd"))
```

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

```
grep("ab", strings, value = TRUE)
```

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

```
grep("^ab", strings, value = TRUE)
```

Operators

- `.`: matches any single character
- `[...]`: a character list, matches any one of the characters inside the square brackets. We can also use `-` inside the brackets to specify a range of characters.
- `[^...]`: an inverted character list, similar to `[...]`, but matches any characters **except** those inside the square brackets.
- `\`: suppress the special meaning of metacharacters in regular expression, i.e. `$ * + . ? [ ] ^ { } | ( ) \`, similar to its usage in escape sequences. Since `\` itself needs to be escaped in R, we need to escape these metacharacters with double backslash like `\\$`.
- `|`: an "or" operator, matches patterns on either side of the `|`.
- `(...)`: grouping in regular expressions which allows to retrieve the bits that matched various parts of your regular expression.

Operators (II)

```
strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12")
grep("ab.", strings, value = TRUE)
```

```
## [1] "abc"   "abd"   "abe"   "ab 12"
```

```
grep("ab[c-e]", strings, value = TRUE)
```

```
## [1] "abc" "abd" "abe"
```

```
grep("ab[^c]", strings, value = TRUE)
```

```
## [1] "abd"   "abe"   "ab 12"
```

```
grep("^ab", strings, value = TRUE)
```

```
## [1] "ab"    "abc"   "abd"   "abe"   "ab 12"
```

Character classes

Character classes allow to specify entire classes of characters, such as numbers, letters, etc. There are two flavors of character classes, one uses `[:` and `:]` around a predefined name inside square brackets and the other uses `\` and a special character. They are sometimes interchangeable.

- `[:digit:]` or `\d`: digits, 0 1 2 3 4 5 6 7 8 9, equivalent to `[0-9]`.
- `\D`: non-digits, equivalent to `[^0-9]`.
- `[:lower:]`: lower-case letters, equivalent to `[a-z]`.
- `[:upper:]`: upper-case letters, equivalent to `[A-Z]`.
- `[:alpha:]`: alphabetic characters, equivalent to `[[:lower:][:upper:]]` or `[A-z]`.
- `[:alnum:]`: alphanumeric characters, equivalent to `[[:alpha:][:digit:]]` or `[A-z0-9]`.

Character classes (II)

- `\w`: word characters, equivalent to `[[:alnum:]_]` or `[A-z0-9_]`.
- `\W`: not word, equivalent to `[^A-z0-9_]`.
- `[:xdigit:]`: hexadecimal digits (base 16), 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f, equivalent to `[0-9A-Fa-f]`.
- `[:blank:]`: blank characters, i.e. space and tab.
- `[:space:]`: space characters: tab, newline, vertical tab, form feed, carriage return, space.
- `\s`: space, ‘ ‘.
- `\S`: not space.
- `[:punct:]`: punctuation characters, ! " # $ % & ’ ( ) - + , - . / : ; < = > ? @ [ ] ^ _ ‘ { | } ~.

Character classes (III)

- `[:graph:]`: graphical (human readable) characters: equivalent to `[[:alnum:][:punct:]]`.
- `[:print:]`: printable characters, equivalent to `[[:alnum:][:punct:]\\s]`.
- `[:cntrl:]`: control characters, like `\n` or `\r`, `[\x00-\x1F\x7F]`.

Note:

- `[:...:]` has to be used inside square brackets, e.g. `[[:digit:]]`.
- `\` itself is a special character that needs escape, e.g. `\\d`. Do not confuse these regular expressions with R escape sequences such as `\t`.

General modes for patterns

There are different syntax standards for regular expressions, and R offers two:

- POSIX extended regular expressions (default)
- Perl-like regular expressions.

You can easily switch between by specifying `perl = FALSE/TRUE` in `base` R functions, such as `grep()` and `sub()`. For functions in the `stringr` package, wrap the pattern with `perl()`.

General modes for patterns (II)

There's one last type of regular expression – "fixed", meaning that the pattern should be taken literally. Specify this via `fixed = TRUE` (base R functions) or wrapping with `fixed()` (`stringr` functions). For example, `"A.b"` as a regular expression will match a string with "A" followed by any single character followed by "b", but as a fixed pattern, it will only match a literal "A.b".

```
strings <- c("Axbc", "A.bc")
pattern <- "A.b"
grep(pattern, strings, value = TRUE)
```

```
## [1] "Axbc" "A.bc"
```

```
grep(pattern, strings, value = TRUE, fixed = TRUE)
```

```
## [1] "A.bc"
```

General modes for patterns (III)

By default, pattern matching is case sensitive in R, but you can turn it off with `ignore.case = TRUE` (base R functions) or wrapping with `ignore.case()` (`stringr` functions). Alternatively, you can use `tolower()` and `toupper()` functions to convert everything to lower or upper case. Take the same example above:

```
pattern <- "a.b"
grep(pattern, strings, value = TRUE)
```

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

```
grep(pattern, strings, value = TRUE, ignore.case = TRUE)
```

```
## [1] "Axbc" "A.bc"
```

Regular expression vs shell globbing

The term globbing refers to pattern matching based on wildcard characters. A wildcard character can be used to substitute for any other character or characters in a string. Globbing is commonly used for matching file names or paths, and has a much simpler syntax. Below is a list of globbing syntax and their comparisons to regular expression:

- `*`: matches any number of unknown characters, same as `.*` in regular expression.
- `?`: matches one unknown character, same as `.` in regular expression.
- `\`: same as regular expression.
- `[...]`: same as regular expression.
- `[!...]`: same as `[^...]` in regular expression.

Resources

- Regular expression in R official document.
- Perl-like regular expression: regular expression in perl manual.
- `qdapRegex` package: a collection of handy regular expression tools, including handling abbreviations, dates, email addresses, hash tags, phone numbers, times, emoticons, and URL etc.
- On these websites, you can simply paste your test data and write regular expression, and matches will be highlighted.
  - regexpal
  - RegExr

Homework Exercises

Homework Exercises

For this week's homework exersises go to Moodle and answer the Quiz posted in the Working with Strings section.

Deadline: Sunday, November 5 before midnight.

That's it for today. Questions?