Lecture 9: Strings

UNIVERSITY OF
SAN FRANCISCO

James D. Wilson

BSDS 100 - Intro to Data Science with `R`

Outline

- Character Functions
- Regular Expressions
- Character Matching
- Substitutions

Reference: Chapters 14 in *R for Data Science* book

Character Functions

- `R` is not known for its prowess in dealing with textual analysis and natural language processing, but it does have some useful features and functions that give it some of the textual functionality of Python and Perl
- We will partially rely on the `stringr` package, which has simplified text analysis quite a bit
- Before we get there, we'll first talk about a few basic character functions

The `length()` function

- For vectors, matrices, arrays, factors, data frames and lists, `length()` returns the **number of elements**. A single string only counts as *one* element:

```
> myStr_01 <- "abcdef"
> length(myStr_01)
[1] 1

> myStr_02 <- "abcdefghijklmnopqrstuvwxyz"
> length(myStr_02)
[1] 1

> myVec_01 <- 1:5
> length(myVec_01)
[1] 5
```

The `nchar()` function

- The vectorized `nchar()` function will return the *number of characters* in a character value

```
> myStr_01 <- "abcdef"
> nchar(myStr_01)
[1] 6

> myStr_02 <- "abcdefghijklmnopqrstuvwxyz"
> nchar(myStr_02)
[1] 26

> myVec_02 <- 12:8
> nchar(myVec_02)
[1] 2 2 2 1 1
```

- Note that `nchar()` coerces numeric values to characters

The `cat()` function

- The `cat()` function will combine character values and print them to the screen or to a file
- `cat()` coerces its arguments to character values, the concatenates and displays them

The `paste()` function

- The `paste()` function will accept an unlimited number of scalars, and join them together, separating each scalar with a space by default
- To use a character string other than a space as a separator, the `sep=` argument can be used

```
> x <- 99

> paste('My age is: ', 1 + x, "...boy am I old", sep="***")
[1] "My age is: ***100***...boy am I old"
```

The `paste()` function

- If you pass a character **vector** to `paste()`, the `collapse=` argument can be used to specify a character string to place between each element of the vector

```
> paste(c("abc", "abcdef", "ZzZzZZzZzZ"))
[1] "abc"        "abcdef"     "ZzZzZZzZzZ"

> paste(c("abc", "abcdef", "ZzZzZZzZzZ"), sep = "")
[1] "abc"        "abcdef"     "ZzZzZZzZzZ"

> paste(c("abc", "abcdef", "ZzZzZZzZzZ"), collapse = "")
[1] "abcabcdefZzZzZZzZzZ"

> paste(c("abc", "abcdef", "ZzZzZZzZzZ"), collapse = " ")
[1] "abc abcdef ZzZzZZzZzZ"
```

The `paste()` function

- When multiple arguments are passed to `paste()`, it will vectorize the operations, recycling shorter elements when necessary

```
> paste("x", 1:5, sep = "_")
[1] "x_1" "x_2" "x_3" "x_4" "x_5"
```

- Including `collapse=` collapses individual elements into a single string

```
> paste("x", 1:5, sep = "_", collapse = "")
[1] "x_1x_2x_3x_4x_5"

> paste("x", 1:5, sep = "_", collapse = " ")
[1] "x_1 x_2 x_3 x_4 x_5"
```

The `substring()` function

- The `substring()` function can be used either to extract parts of character strings, or to change the values of part of character strings
- `substring()` accepts the arguments `first=` and `last=`, identifying the location of the first and last character (with an integer), respectively, in the the string
- The `first=` is required, but `last=` may be omitted
- `substring()` coerces inputs to characters

The `substring()` function

```
> (myStr_03 <- paste(LETTERS[1:8], letters[1:8], sep = "",
collapse = ""))
[1] "AaBbCcDdEeFfGgHh"

> substring(myStr_03, 6, 12)
[1] "cDdEeFf"

> myNum <- 123456789

> substring(myNum, 3)
[1] "3456789"
```

`substring()` continued...

- `substring()` is vectorized
  - for `first=` and `last=` arguments

```
> (myStr_04 <- c("paul", "john", "sally"))
[1] "paul"  "john"  "sally"

> substring(myStr_04, 3, 4)
[1] "ul" "hn" "ll"
```

  - for character vectors passed to `substring()`

```
> (myStr_05 <- 'my tiny bed')
[1] "my tiny bed"

> substring(myStr_05, first = c(1, 4, 9), last = c(2, 7, 11))
[1] "my"   "tiny" "bed"
```

`substring()` continued...

- `substring()` may also be used for assignment

```
> myStr_06 <- "my big dog"

> substring(myStr_06, 8, 10) <- "cat"

> myStr_06
[1] "my big cat"

> substring(myStr_06, 4, 6) <- "gigantic"

> myStr_06
[1] "my gig cat"
```

- There is also a function `substr()` which operates similarly to `substring()`, but the latter is more robust

Regular Expressions in R

- Regular expressions are a method of expressing patterns in character values which can then be used to extract parts of strings or to modify those strings in some way
- The most common functions that support working with regular expressions in R are `strsplit()`, `grep()`, `grepl()`, `sub()`, `gsub()`, `regexpr()` and `gregexpr()`
- The backslash character (\) is used in regular expressions to signal that certain characters with special meaning in regular expressions should be treated as normal characters

Regular Expressions in R

- In R, this means that two backslash characters need to be entered into an input string anywhere that special characters need to be escaped
- Although the double backslash will display when the string is printed, the use of `nchar()` or `cat()` will confirm that only a single backslash is actually included in the string

Regular Expressions

- Regular expressions are composed of three components
  - *literal characters*: matched by a single character
  - *character classes*: can be matched by any number of characters
  - *modifiers*: operate on literal characters or character classes
- As many punctuation marks are regular expression modifiers, the following characters must always be preceded by a backslash to retain their literal meaning

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

Forming Character Classes

- To form a character class, use square brackets ([ ]) surrounding the characters to be matched

E.g. to create a character class that will be matched either by `x`, `y`, `z` or the number `1`, use `[xyz1]`

Forming Character Classes

- Dashes are used inside of character classes to represent a range of values, e.g., `[a-z]` or `[2-9]`
- If a dash is to be literally included in a character class, it should be either the first character in the class or it should be preceded by a backslash
- Other special characters, save square brackets, do not need to be preceded by a backslash when used in a character class

Modifiers for Regular Expressions

Modifier	Meaning
^	anchors expression to the beginning of target
$	anchors expression to end of target
.	matches any single character except newline
|	separates alternative patterns
( )	groups patterns together
*	matches 0 or more occurrences of preceding entity
?	matches 0 or 1 occurrence of preceding entity

Modifiers for Regular Expressions

Modifier	Meaning
$+$	matches 1 or more occurrences of preceding entity
$\{n\}$	matches exactly n occurrences of preceding entity
$\{n,\}$	matches n or more occurrences of preceding entity
$\{n,m\}$	matches between n and m occurrences of preceding entity

Example

- Modifiers operate on whatever entity then follow, using parentheses for grouping if necessary

E.g. To identify a string with two digits, followed by one or more letters, the matching regular expression would be

‘`[0-9][0-9][a-zA-Z]+`’

E.g. For two consecutive appearances of the string ‘`photo`’ the matching regular expression could be

‘`(photo){2}`’

Example

E.g. For `jpg` filename consisting exclusively of letters, the matching regular expression could be

‘`^[a-zA-Z]+\\.jpg$`’

- Observe how this regular expression is constructed
  - `^` explicitly states that the file must **begin** with whatever proceeds it, in this case, `[a-zA-Z]`
  - `+` allows for any number of letters, so long as there is at least one
  - The second backslash, i.e., the backslash on the right, is required so that `.` can retain its literal meaning (recall `.` is a regular expression modifier
  - The first backslash, i.e., the backslash on the left, is a required `R` quirk
  - `$` ensures that the **final** four characters in the file name are `.jpg`

The `strsplit()` Function

- The `strsplit()` function can use a character string or regular expression to divide up a character string into smaller pieces
- `strsplit()` returns its results in a list, regardless of input
- Given `strsplit()` accepts regular expressions to determine where to split a string, the function is very versatile

Example

To break up a sentence into its constituent words

```
> myString_07 <- "I enjoy reading books"

> strsplit(myString_07, "")
[[1]]
 [1] "I" " " "e" "n" "j" "o" "y" " " "r" "e" "a" "d" "i" "n" "g"
[16] " " "b" "o" "o" "k" "s"

> strsplit(myString_07, " ")
[[1]]
[1] "I"       "enjoy"   "reading" "books"
```

Example

E.g. It commonly occurs that when customers input free-form text on surveys, they may accidentally include more than once space between words, thereby requiring a regular expression to appropriately handle the variable inputs

```
> myString_08 <- "I     enjoy reading   books"

> strsplit(myString_08, " ")
[[1]]
 [1] "I"       ""        ""        ""        ""        "enjoy"
 [7] "reading" ""        ""        "books"

> strsplit(myString_08, " +")
[[1]]
[1] "I"       "enjoy"   "reading" "books"
```

Using Regular Expressions in R

- Sufficiently versed in the construction of regular expressions, those expressions can now be implemented for use with specific functions
- `grep()` and `grepl()` are used to test for the presence of a regular expression
- `regexpr()` and `gregexpr()` can pinpoint and potentially extract those parts of a string that were matched by a regular expression

The `grep()` Function

- `grep()` accepts a regular expression and a character string or vector of character strings, and returns the **indices** of those elements of the string which are matched by the regular expression
- If the `value = TRUE` argument is passed to `grep()`, it will return the actual strings which matched the expression instead of the indices
- To match literal strings (instead of regular expressions), the `fixed = TRUE` argument should be passed to `grep()`

Example

```
> firstNames <-read.csv("~/Desktop/firstNames.csv",
stringsAsFactors = F)

> grep('^Ai', firstNames$firstname)
[1] 54 55 56 57 58 59 60 61

> grep('^Ai', firstNames$firstname, value = T)
[1] "Ai"     "Aida"   "Aide"   "Aiko"   "Aileen" "Ailene"
[7] "Aimee"  "Aisha"
```

The `grep()` Function

- To find regular expressions regardless of case, use

  `ignore.case = TRUE`

- To search for a regular expression that is a single word, e.g., `shoes`, which is not preceded nor proceeded by any other characters except for punctuation, whitespace or the beginning or ending of a line, wrap the string with escaped angled brackets (`\\<` and `\\>`)

```
> myStr_09 <- c("run to the store", "running", "run...quickly!",
 "run!")


> grep('\\<run\\>', myStr_09, value = T)
[1] "run to the store" "run...quickly!"   "run!"
```

The `grep()` Function

- If the regular expression passed to `grep()` is not matched, `grep()` returns an empty numeric vector, which can be easily evaluated to be `TRUE` or `FALSE` using the `any()` function
- The `grepl()`] function will return a logical vector the length of the input vector, identifying the elements which matched the regular expression

The `regexpr()` Function

- `regexpr()` pinpoints and can extract those parts of a string that are matched by a regular expression
- `regexpr()` outputs a vector of *starting positions* of the regular expressions found; if none are found, `-1` is returned
- `match.length` provides information about the length of each match
- `regexpr()` will only provide information on the **first** match in a given input string

Example

```
> myStr_10 <- c("94112 94117", "H8P 2S5", " 90210", "47907-1233")

> regexpr('[0-9]{5}', myStr_10)
[1]  1 -1  2  1
attr(,"match.length")
[1]  5 -1  5  5
attr(,"useBytes")
[1] TRUE
```

The `gregexpr()` Function

- `gregexpr()` operates similarly to `regexpr()`, but returns information on **all** matches found (not just the first)
- `gregexpr()` always returns its result in the form of a list

Example

```
> myStr_10 <- c("94112 94117", "H8P 2S5", " 90210", "47907-1233")

> gregexpr('[0-9]{5}', myStr_10)
[[1]]
[1] 1 7
attr(,"match.length")
[1] 5 5
attr(,"useBytes")
[1] TRUE

[[2]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"useBytes")
[1] TRUE

[[3]]
[1] 2
attr(,"match.length")
[1] 5
attr(,"useBytes")
[1] TRUE

[[4]]
[1] 1
attr(,"match.length")
[1] 5
attr(,"useBytes")
[1] TRUE
```

Substitutions

- To substitute text based on regular expressions, `R` provides two functions, `sub()` and `gsub()`
- Both functions accept
  1. a regular expression
  2. a string containing what will be substituted for the regular expression
  3. a string (or strings) to operate on
- Like `regexpr()` and `gregexpr()`, `sub` only changes **only** the first occurrence of the regular expression, whereas `gsub` changes all occurrences

The `gsub()` Function

A common application of `gsub()` is the scrubbing of financial data

```
> financialData_01 <- c("$11,345.65", "$99,125.22", "$13,321.99")

> str(financialData_01)
 chr [1:3] "$11,345.65" "$99,125.22" "$13,321.99"

> as.numeric(financialData_01)
[1] NA NA NA
Warning message:
NAs introduced by coercion

> sub('[$,]', '', financialData_01)
[1] "11,345.65" "99,125.22" "13,321.99"

> gsub('[$,]', '', financialData_01)
[1] "11345.65" "99125.22" "13321.99"
```