Informatics

String Manipulation with R

Claudio Sartori
Department of Computer Science and Engineering
claudiosartori@uniboit
https://wwwuniboit/sitoweb/claudiosartori/

What is a *string*

- A vector of characters
  - a character is an ASCII value
- Has a number of dedicated functions, different from those of standard vectors
- Strings can be the basis for any repeating objects
  - Vector of strings
  - Matrix of strings
  - Data frame column of strings
  - List of strings

What to do with strings?

- remove a given character in the names of your variables
- replace a given character in your data
- convert labels to upper case (or lower case)
- struggling with xml (or html) files
- modifying text files in excel changing labels, categories, one cell at a time, or doing one thousand copy-paste operations
- split unformatted text into paragraphs, sentences, words
- get rid of punctuation and special characters
- ...

A toy example

```
> head(USArrests)
           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
```

A toy example

```
> head(rownames(USArrests))
[1] "Alabama"    "Alaska"     "Arizona"
"Arkansas"   "California" "Colorado"
> stateNames <- rownames(USArrests)
> statesChars <- nchar(states)
which(statesChars==max(statesChars))
```

Getting text into R

function	***description***
`read.table()`	read a file with regular structure: rows made by sequences of fields of equal length; if lengths are different: strange results
`read.csv()`	specialisation of read.table for reading data frames
`scan(file, what)`	read a text file and generate a vector – what: character(), double()
`readLines()`	read entire lines into a vector of strings

Example (1)

```
> abc <- "http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv"
> radio <- read.table(abc, header = TRUE, sep = ",")
> dim(radio)
[1] 52 18
> typeof(radio)
[1] "list"
> head(radio)
  State                        Website.URL                   Station
1   QLD    http://www.abc.net.au/brisbane/        ABC Radio Brisbane
2   QLD http://www.abc.net.au/capricornia/           ABC Capricornia
3   QLD    http://www.abc.net.au/farnorth/             ABC Far North
4   QLD   http://www.abc.net.au/goldcoast/       91.7 ABC Gold Coast
5   QLD    http://www.abc.net.au/northqld/      ABC North Queensland
6   QLD   http://www.abc.net.au/northwest/ ABC North West Queensland
...
```

Example (2)

```
> abc <- "http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv"
> radio <- read.csv(abc)
> dim(radio)
[1] 52 18
> typeof(radio)
[1] "list"
> head(radio)
  State                        Website.URL                   Station
1   QLD    http://www.abc.net.au/brisbane/        ABC Radio Brisbane
2   QLD http://www.abc.net.au/capricornia/           ABC Capricornia
3   QLD    http://www.abc.net.au/farnorth/             ABC Far North
4   QLD   http://www.abc.net.au/goldcoast/       91.7 ABC Gold Coast
5   QLD    http://www.abc.net.au/northqld/      ABC North Queensland
6   QLD   http://www.abc.net.au/northwest/ ABC North West Queensland
...
```

Strings ad factors?

by default read.csv converts strings to factors
if the number of unique values is very high this is useless

```
> typeof(radio$Town)
[1] "integer"
> length(radio$Town)
[1] 52
> length(unique(radio$Town))
[1] 52
> radio <- read.csv(abc, stringsAsFactors = FALSE)
> typeof(radio$Town)
[1] "character"
```

Inspect the structure

```
> str(radio, vec.len = 1)
'data.frame':   52 obs. of  18 variables:
 $ State            : chr  "QLD" ...
 $ Website.URL      : chr
"http://www.abc.net.au/brisbane/" ...
 $ Station          : chr  "ABC Radio Brisbane"
...
 $ Town             : chr  " Brisbane " ...
 $ Latitude         : num  -27.5 ...
 $ Longitude        : num  153 ...
 $ Talkback.number : chr  "1300 222 612" ...
 $ Enquiries.number: chr  "07 3377 5222" ...
```

```
 $ Fax.number       : chr  "07 3377 5612" ...
 $ Sms.number       : chr  "0467 922 612" ...
 $ Street.number    : chr  "114 Grey Street" ...
 $ Street.suburb    : chr  "South Brisbane" ...
 $ Street.postcode : int  4101 4700 ...
 $ PO.box           : chr  "GPO Box 9994" ...
 $ PO.suburb        : chr  "Brisbane" ...
 $ PO.postcode      : int  4001 4700 ...
 $ Twitter          : chr  " abcbrisbane" ...
 $ Facebook         : chr  "
https://www.facebook.com/abcinbrisbane" ...
```

String functions

- **`print()`** – generate output depending on the class of the variable
- **`nchar()`** - Find the length of a string
- **`paste()`** - Assemble a string from parts
- **`substr()`** - Extract a substring
- **`grep()`** - Search for a substring
- **`strsplit()`** - Split a string into substrings.
    - like the Linux command of the same name
- **`sprintf()`** - Assemble a string from parts
- **`sub()`, `gsub()`** – Substitute patterns

nchar()

takes a character vector as an argument and returns a vector whose elements contain the sizes of the corresponding elements of x

```
> v <- c("abc", "defg")
> nchar(v)
[1] 3 4
```

```
paste(..., sep = " ", collapse = NULL)
paste0(..., collapse = NULL)
```

Concatenate vectors after converting to character

```
> v <- paste("x=", 1, "y=", 2)
> print(v)
[1] "x= 1 y= 2"
> v0 <- paste0("x=", 1, "y=", 2)
> print(v0)
[1] "x=1y=2"
```

More on `paste`

- sep
  - the character for term separation
- collapse
  - the character for the separation of the results,
  - if each term in the argument is a vector, paste generates a vector of results, they will be separated by the "collapse" parameter

```
cat(... , file = "", sep = " ", append = FALSE)
```

- possible file output to text file
- sep = separation character
- if a new line is required at the end it is given by **"\n"**

```
format(x, digits = NULL, nsmall = 0L,
       justify = c("left", "right", "centre", "none"),
       width = NULL, scientific = NA, ...)
```

- **`width`** minimum width of the string
  - if necessary spaces are added to the left
- **`trim`** if set to TRUE no padding with spaces
  - if set to FALSE (the default), when formatting vectors spaces are added to the left to make all the outputs of equal length
- **`justify`** "left", "right", "centre", "none"
- **`digits`** for numbers, number of digits in the output
- **`nsmall`** for number of digits to the right of the decimal place
- **`scientific`** TRUE for scientific notation

Examples

```
> format(13.7)
[1] "13.7"
> format(3.1416)
[1] "3.1416"
> format(13.7, digits = 2)
[1] "14"
> format(13.7, nsmall = 2)
[1] "13.70"
> format(13.7, scientific = TRUE)
[1] "1.37e+01"
```

`strsplit(x, split, ...)`

- Split the elements of a character vector x into substrings according to the matches to substring split within them
- This can be used if for some reason, for instance for cleaning purpose, we need to start considering entire lines (read them with **`readLines()`**) and then we split them
- As an alternative, **`scan()`** alone generates word splitting

Regular expressions
(regex or regexp for short)

- a special text string for describing a search pattern
  - you can think of regular expressions as wildcards on steroids
  - you are probably familiar with wildcard notations such as *.txt to find all text files in a file manager.
  - the regex equivalent is ˆ.*\.txt$. Reference for regular expressions
- an extremely powerful tool for pattern matching and text manipulation

A general algorithm for regex

- for each string in a variable (e.g. a vector)
    - match the pattern with the string
        - if match is true then
            - do something according to the specific function
                - e.g.:
                    - output the index of the string
                    - output the string
                    - substitute the match with something else

What is a regex?

- a combination of alphanumeric characters and special characters
- regex can be combined by means of operators, as happens for expressions
  - concatenation
  - logical OR
  - replication
  - grouping

Special characters (metacharacters)

<table>
<tr><th>Metacharacter</th><th>Meaning</th><th>Escape in R</th></tr>
<tr><td>.</td><td>matches any character</td><td>\\.</td></tr>
<tr><td>$</td><td>end of string</td><td>\\$</td></tr>
<tr><td>*</td><td>quantifier zero or more</td><td>*</td></tr>
<tr><td>+</td><td>quantifier one or more</td><td>\\+</td></tr>
<tr><td>{}</td><td>delimit quantifier multiplier</td><td>\\{ \\}</td></tr>
<tr><td>[]</td><td>delimit an or pattern</td><td>\\[\\]</td></tr>
<tr><td>|</td><td>or operator</td><td>\\|</td></tr>
<tr><td>^</td><td>negation operator | beginning of string</td><td>\\^</td></tr>
<tr><td>()</td><td>grouping operator</td><td>\\(\\)</td></tr>
<tr><td>\</td><td>escape</td><td>\\\\</td></tr>
</table>

The *escape* allows to use metacharacters as standard characters

Concatenation

- Sequence of pattern to obtain an extended condition

```
# concatenate "1" and "0"
# with value = FALSE (the default) generate
# the indexes of the rows matching the pattern
grep("10", top100, value = TRUE)
 [1] "On Jan. 1, 1992, the \"Modern Rock\" station KITS San Francisco (\"Live-105\")"
 [2] "broadcast its list of the \"Top 105.3 of 1991.\"  Here is the countdown"
 [3] "10. NORTHSIDE                   TAKE FIVE"
 [4] "100. MEAT PUPPETS               SAM"
 [5] "101. SMASHING PUMPKINS          SIVA"
 [6] "102. ELVIS COSTELLO             OTHER SIDE OF ..."
 [7] "103. SEERS                      PSYCHE OUT"
 [8] "104. THRILL KILL CULT           SEX ON WHEELZ"
 [9] "105. MATTHEW SWEET              I'VE BEEN WAITING"
[10] "105.3  LATOUR                   PEOPLE ARE STILL HAVING SEX"
```

Logical OR

- Patterns to match alternatively

```
# OR combination
grep("10|20", top100, value = TRUE)
 [2] "broadcast its list of the \"Top 105.3 of 1991.\"  Here is the countdown"
 [3] "10. NORTHSIDE                   TAKE FIVE"
 [4] "20. R.E.M.                      SHINY HAPPY PEOPLE"
 [5] "100. MEAT PUPPETS               SAM"
 [6] "101. SMASHING PUMPKINS          SIVA"
```

Logical OR with *character classes*

- Expression matching *one* character from a *class*

Anchor	Description
[aeiou]	any lowercase vowel
[0-9]	any digit
[a-z]	any lowercase letter
[a-zA-Z0-9]	any letter or digit
[^aeiou]	anything but lowercase vowels

Repetition

- Pattern that matches also in case of repetition

```
> a <- c("abcabcde", "ab", "accccb", "bc")
> grep(pattern = "a*", x = a) # any number of ‘a’ (including 0)
[1] 1 2 3 4
> grep(pattern = "a+", x = a) # at least one ‘a’
[1] 1 2 3
> grep(pattern = "c{2}", x = a) # two ‘c’
[1] 3
```

Grouping

- Include a pattern in round parentheses to apply globally an operator
- uses *quantifiers, negation, …*
- *Groups can be referred in the substitution with reference to the position*
  - *\\1 refers to the first group*

```
> a <- c("abcabcde", "ab", "accccb", "bc")
➢grep("^(ab|ac)", x = a) # ab or ac at the beginning
[1] 1 2 3
```

Main regex functions

Function	Purpose	Characteristic
grep()	finding regex matches	what elements are matched (index or value)
grepl()	finding regex matches	what elements are matched (TRUE OR FALSE)
regexpr()	finding regex matches	what elements are matched
gregexpr()	finding regex matches	Variant
regexec()	finding regex matches	Variant
sub()	replacing regex matches	only first match is replaced
gsub()	replacing regex matches	all matches are replaced
strsplit()	splitting regex matches	split vector according to matches

Focus: **grep()**

```
grep(pattern, x, value = FALSE
      , ignore.case = FALSE, invert = FALSE)
```

it is the basic tool for pattern matching in strings

with value = FALSE returns the indexes of the matching strings in the x vector

with value = TRUE returns the matching strings in the x vector

invert = TRUE inverts the selection logic

Focus: **regexpr()**

```
regexpr(pattern, x, value = FALSE
      , ignore.case = FALSE, invert = FALSE)
```

To find exactly where the pattern is found in a given string

Returns more detailed information than grep():

- what elements of the text vector actually contain the regex pattern
- identifies the position of the substring that is matched by the regular expression pattern
- useBytes allows match byte by byte rather than character by character
  - relevant for multi-byte characters (character encoding)

Focus: **`regexpr()`** (continued)

```
regexpr(pattern, x, useBytes = FALSE
      , ignore.case = FALSE)
```

returns three outputs

- an integer vector of the same length of x with
  - -1 if the pattern was not found
  - the starting position of the first match of the pattern, if it was found
- **`match.length`** - an integer vector of the same length of x with:
  - -1 if the pattern was not found
  - the length of the match, if it was found

Focus: **`sub()`**, **`gsub()`**

`sub(pattern, replacement, x, ignore.case = FALSE...)`

- in all the components matching pattern substitutes the replacement
- if **`pattern`** contains **`(...)`** this is a *matching group*
- in **`replacement`** the matching groups are referred as **`\\1, \\2, ...`** as they appear in the pattern from left to right, then **`sub`** can use the matching groups in substitutions
- **`sub`** operates only on the first match in the **`x`** variable, **`gsub`** operates on all the matches

Example: pattern substitution

```
pattern = "(^[0-9]+)\\. ", replacement = "\\1;"
```

the pattern is any sequence of one or more digits at the beginning of a line

the matching of the group in yellow to the left is replaced in the placeholder in yellow to the right

```
x = "2. abc"  replaced = "2;abc"

x = "123. xyz"    replaced = "123;xyz"
```