# Introduction to Text Computations in R

*@chris-ukds*

*July 11, 2016*

It can be useful to be able to flag illegal strings with the assistance of computer software during the data ingest process, as manual inspection can be both time-consuming and tedious. Many libraries are widely used for processing text, often implemented in scripting languages such as `Perl` or `Python`. In this tutorial, we focus on the `R` language instead, since the end goal is to build a simple web application using the `Shiny` package. `R` has a number of facilities for text processing that we can take advantage of.

Computer systems have different mechanisms for representing text. In `R`, objects of type `character` fill this role; `character` objects correspond to textual information in a way that can be processed by machines and interpreted by human readers. The essential concept for representing text from a human language into a set of machine-readable numeric codes is known as `character encoding`. Throughout the history of computing, various attempts at accommodating arbitrary languages have been made - and after much heartache, there is now (fortunately) a near-universal acceptance of two standards: `UTF-8` and `Unicode`. In `R`, it is possible to specify a particular text encoding (or `locale`) by a call to the function `Sys.setlocale()`. It is also possible to convert between encodings if necessary, with a call to `iconv()`. A comprehensive discussion of encoding can be found in the official documentation (`help(locales)`).

Broadly speaking, there are five main classes of functions for text computations: matching, finding, replacing, subsetting, and splitting based on textual patterns.

## Matching patterns

The function `match(x, table, ...)` (wholly) matches character data `x` to the values found in `table` and returns the index of the first match.

```
match("Gena", c("Joe", "Sam", "Gena"))
```

```
## [1] 3
```

```
match("Coffee", c("Coffee", "Tea", "Lemonade", "Coffee"))
```

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

It is also possible to find *partial* matches in text, using the `pmatch(x, table, ...)` and `charmatch(x, table, ...)` functions.

```
##  Set up the input text and table.
x = c("", "Gh", "Gh")
tab = c("Garrison", "Ghost", "Nightmare")
```

```
##  Observe that there are no full matches.
match(x, tab)
```

```
## [1] NA NA NA
```

```
##  By default, values of "table" that are matched once are excluded
##  from the search for subsequent matches. This behaviour can be
##  inhibited by setting "dup = TRUE".
pmatch(x, tab)
```

```
## [1] NA  2 NA
```

```
pmatch(x, tab, dup = TRUE)
```

```
## [1] NA  2  2
```

```
##  "charmatch" differentiates between a non-match and ambiguous
##  partial match. It also matches empty strings and permits
##  duplicates by default.
charmatch(x, tab)
```

```
## [1] 0 2 2
```

### Finding patterns

#### grep and grepl

The function `grep(pattern, x, ...)` matches the character data `x` to the values in `pattern`. It returns a `numeric` vector consisting of the location of the matched elements. `grepl(pattern, x, ...)` is equivalent to `grep()`, except that it returns a `logical` vector (hence the name grep**l**).

```
##  Set up the input text and pattern.
x = c("Rafael", "Roger", "John", "James")
pat = "roger"
```

```
grep(pat, x)
```

```
## integer(0)
```

```
grep(pat, x, ignore.case = TRUE)
```

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

```
grep(pat, x, ignore.case = TRUE, val = TRUE)
```

```
## [1] "Roger"
```

```
grepl(pat, ignore.case = TRUE, x)
```

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

#### regexpr, gregexpr, and regexec

The function `regexpr(pattern, x, ...)` takes the same arguments as `grep()` but returns an `integer` vector of the same length as `x`. This means that it identifies which element of `x` contains the `pattern` as well as the position of the subexpression matched by `pattern`. Each element in the vector returned by `regexpr()` contains the position at which the match was found. Each element also has a `match.length` attribute which contains the *lengths* of all the matches. Any non-matches are indicated by the value `-1`. The value of the `useBytes` attribute indicates whether or not the pattern matching is done on a byte-by-byte basis. Further details can be found in the documentation (`help(gregexpr)`).

The function `regexec(pattern, x, ...)` is similar to `regexpr()`, except it returns a `list` containing the starting position of the match. Another close relative is `gregexpr(pattern, x, ...)`, which finds *all* matches in each string and returns a `list` of the same length as `x`.

```
##  Set up the input text and pattern.
x = c("abc", "ab", "abcab", "abba", "abcba bcbab abba")
pat = "ba"
```

```
grep(pat, x)
```

```
## [1] 4 5
```

```
regexpr(pat, x)
```

```
## [1] -1 -1 -1  3  4
## attr(,"match.length")
## [1] -1 -1 -1  2  2
## attr(,"useBytes")
## [1] TRUE
```

```r
regexec(pat, x)
```

```
## [[1]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] 3
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
##
## [[5]]
## [1] 4
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
```

```r
gregexpr(pat, x)
```

```
## [[1]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] 3
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
##
## [[5]]
## [1]  4  9 15
## attr(,"match.length")
## [1] 2 2 2
## attr(,"useBytes")
## [1] TRUE
```

### Replacing patterns

We may, on occasion, wish to replace one textual pattern with another. The functions `sub(pattern, replacement, x, ...)` and `gsub(pattern, replacement, x, ...)` can be used for this purpose. `sub()` replaces the *first* instance of the pattern, whereas `gsub()` replaces *all* instances. Note that if either `pattern` or `replacement` has `length > 1`, only the first element will be used.

```
##  Set up input text, pattern, and replacement.
x = c("hezrtzche", "ghzstly", "nzturzl")
patt = "z"
repl = "a"
```

```
sub(patt, repl, x)
```

```
## [1] "heartzche" "ghastly"   "naturzl"
```

```
gsub(patt, repl, x)
```

```
## [1] "heartache" "ghastly"   "natural"
```

### Subsetting patterns

It can be useful to extract substrings in a `character` vector to obtain a *subset*. The functions `substr(x, start, stop)` and `substring(x, first, last)` offer facilities for string extraction. The only difference is that for `substring`, the second argument to `substring` is optional - hence it is slightly more flexible than `substr`.

```
##  Set up input text, start/first, and stop/last.
x = c("replacement", "treason", "programming")
a = 3
b = 7
```

```
substr(x, a, b)

## [1] "place" "eason" "ogram"

substring(x, a, b)

## [1] "place" "eason" "ogram"

substring(x, a)

## [1] "placement" "eason"     "ogramming"

##  Note that the above functions can also be used for string replacement.
x = c("replacing", "strings", "with", "substr", "and", "substring")
a = 3
rep = "!@#"

substring(x, a) = rep
x

## [1] "re!@#cing" "st!@#gs"   "wi!@"      "su!@#r"    "an!"       "su!@#ring"

##  Observe that the recycling rule is applied if length(rep) > 1.
x = c("replacing", "strings", "with", "substr", "and", "substring")
rep = c("!", "@", "#")

substring(x, a) = rep
x

## [1] "re!lacing" "st@ings"   "wi#h"      "su!str"    "an@"       "su#string"
```

### Splitting on patterns

Another common task in text computation is to split a string based on a pattern. The function `strsplit(x, split, ...)` specializes in this task, breaking up `character` strings into fields and returning the `split` substrings in a `list`. The `split` argument may be regarded as being equivalent to the `pattern` argument seen in functions such as `grep()` and `regexpr()`.

```
##  Set up an input vector
x = c("bob@ukdataservice.ac.uk", "john@data-archive.ac.uk",
      "bill@ukda.ac.uk", "jake@ukds.ac.uk")

##  Split on "@" to obtain the user and domain names.
strsplit(x, "@")

## [[1]]
## [1] "bob"                 "ukdataservice.ac.uk"
##
## [[2]]
## [1] "john"               "data-archive.ac.uk"
##
## [[3]]
## [1] "bill"       "ukda.ac.uk"
##
## [[4]]
## [1] "jake"       "ukds.ac.uk"
```

## Regular Expressions

Regular expressions are instructions for encoding pattern-matching instructions into character strings using certain rules, and are widely used on UNIX-like systems. Regular expressions are governed by a set of rules that allow them to be parsed and interpreted; they were designed to give functions instructions on matching particular patterns in text data.

In `R`, regular expressions are passed as `character` strings to functions, and are applied to each element of some text data (say `x`), of type `character`. The regular expression is is used for either *matching* string patterns or *substituting* text. That is, they can be passed on as either the `pattern` or `split` arguments to the functions studied above. Regular expressions are requests to the function to seek a matching substring in a string, where the pattern is read from left to right, and each chunk of the textual pattern (called *subexpressions*) are matched to the text.

### Metacharacters

The simplest possible form of regular expressions are single characters, which match themselves. For instance, `"foo"` matches the word *foo*. There are, however, a number of *special characters*, called *metacharacters*, which have a special meaning that depend on the context in which they are used. The metacharacters in `R` are:

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

Metacharacters are a part of the `R` language grammar and need to be *escaped* with a double backslash (`\\`) if they are to be interpreted literally.

```
##  Demo for metacharacters.
metaChars = c("$", "*", "+", ".", "?", "[", "^", "{", "|", "(", "\\")
```

```
##  "$" matches the empty string at the end of a line.
grep("$", metaChars,  value = TRUE)
```

```
##  [1] "$"  "*"  "+"  "."  "?"  "["  "^"  "{"  "|"  "("  "\\"
```

```
##  Escape for literal interpretation.
grep("\\$",metaChars,  value = TRUE)
```

```
## [1] "$"
```

### Quantifiers

Subexpressions can be *quantified* using *quantifers* which specify how many instances of the subexpression will be matched. Quantifiers in `R` include:

| Quantifier | Match |
|---|---|
| `*` | zero or more instances |
| `+` | one or more instances |
| `?` | optional, and matched only once |
| `{n}` | exactly `n` instances |
| `{n,}` | at least `n` instances |
| `{n,m}` | at least `n` but no more than `m` instances |

```
##  Set up input vector.
x = c("", "ab", "abab", "ababab", "baba ab", "aa", "aaa")
```

```
grep("a*", x, val = TRUE)
```

```
## [1] ""        "ab"      "abab"    "ababab"  "baba ab" "aa"      "aaa"
```

```r
grep("a+", x, val = TRUE)
```

```
## [1] "ab"      "abab"    "ababab"  "baba ab" "aa"      "aaa"
```

```r
grep("a{3}", x, val = TRUE)
```

```
## [1] "aaa"
```

```r
grep("a{2,}", x, val = TRUE)
```

```
## [1] "aa"  "aaa"
```

```r
grep("a{1,2}", x, val = TRUE)
```

```
## [1] "ab"      "abab"    "ababab"  "baba ab" "aa"      "aaa"
```

```r
grep("a*b", x, val = TRUE)
```

```
## [1] "ab"      "abab"    "ababab"  "baba ab"
```

```r
##  It is possible to join multiple regular expressions by the
##  infix operator |. The resulting expression matches *any*
##  string matching either subexpression.
grep("a*b|a{3}", x, val = TRUE)
```

```
## [1] "ab"      "abab"    "ababab"  "baba ab" "aaa"
```

### Sequences

Sequences define a set of characters that can be matched:

| Anchor | Match |
|---|---|
| \\d | Digit |
| \\D | Non-digit |
| \\s | Space |
| \\S | Non-space |
| \\w | word |
| \\W | Non-word |
| \\b | Word boundary |
| \\B | Non-(word boundary) |
| \\h | Horizontal space |
| \\H | Non-horizontal space |
| \\v | Vertical space |
| \\V | Non-vertical space |

```r
##  Set up input vector.
x = c("", "!@#", "a12b", "abba", "foo bar quux", "boo-hoo")
grep("\\d", x, val = TRUE)
```

```
## [1] "a12b"
```

```r
grep("\\s", x, val = TRUE)
```

```
## [1] "foo bar quux"
```

```r
## "\\b" matches the empty string at either end of a word.
grep("\\b", x, val = TRUE)
```

```
## [1] ""             "!@#"          "a12b"         "abba"
## [5] "foo bar quux" "boo-hoo"
```

```
grep("\\w", x, val = TRUE)

## [1] "a12b"         "abba"         "foo bar quux" "boo-hoo"

##  Mask all.
gsub(".", "*", "1q2w3e4")

## [1] "*******"

##  Mask numbers.
gsub("\\d", "*", "a4b3c2d1")

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

### Character Classes

A character class in a regular expression is a list of characters enclosed by square brackets (`[.]`), and are used to match elements character by character. Inserting a caret (ˆ) at the beginning *negates* the character class, searching for characters that are *not* enclosed by square brackets. Some commonly encountered character classes include:

| Anchor | Match |
|---|---|
| [0-9] | Any digit |
| [ˆ0-9] | Non-digit |
| [a-z] | lower case ASCII |
| [A-Z] | upper case ASCII |
| [a-zA-Z] | lower and upper case ASCII |

```
##  Set up input vector.
x = c("a", "ab", "abc", "a12", "abc2", "123", "*")
grep("[0-9]+", x, val = TRUE)

## [1] "a12"  "abc2" "123"

grep("[a-zA-Z0-9]", x, val = TRUE)

## [1] "a"    "ab"   "abc"  "a12"  "abc2" "123"

grep("[^a-zA-Z0-9]", x, val = TRUE)

## [1] "*"
```

However, the classes above depend on the *locale* (see introduction). The current locale of your machine defines what is considered to be alphanumeric. In particular, letters need not be restricted to the usual characters in the English language, i.e. the character range "[a-zA-Z]". To ensure that the software adapts to the locale, it is preferable to use predefined `POSIX` character classes (at the cost of a slight loss in code readability), which have reserved names and are enclosed between `[:` and `:]`. The table below lists the most commonly encountered `POSIX` character classes in `R`:

| Anchor | Match |
|---|---|
| [:lower:] | Lower-case letters |
| [:upper:] | Upper-case letters |
| [:alpha:] | [:lower:] and [:upper:] |
| [:digit:] | Any digit |
| [:alnum:] | [:alpha:] and [:digit:] |
| [:space:] | Space and tab |
| [:punct:] | Punctuation characters |

Note that the classes above must be enclosed in square brackets, e.g. `[[:space:]]`, to allow bracket expressions to contain any combination of the classes and individual characters.

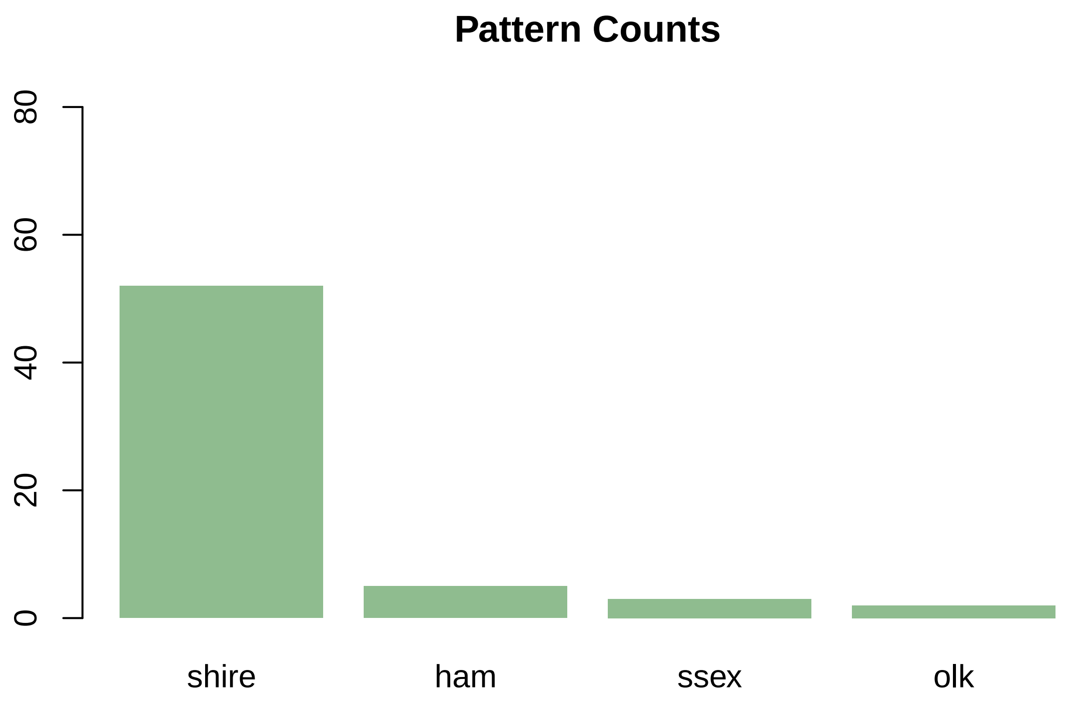
```
##  Set up input vector.
x = c("a", "ab", "abc", "a12", "abc2", "123", "*")
grep("[[:digit:]]+", x, val = TRUE)
## [1] "a12"  "abc2" "123"
grep("[[:alnum:]]", x, val = TRUE)
## [1] "a"    "ab"   "abc"  "a12"  "abc2" "123"
grep("[^[:alnum:]]", x, val = TRUE)
## [1] "*"
```

### Examples

Having covered the bases, we can have a bit of fun.

```
##  It can be useful to be able to trim leading and trailing (as well
##  as consecutive) white spaces from text.
trimSpace =
    function(x) {
        stopifnot(is.character(x))
        gsub("^[[:space:]]|[[:space:]]$", "",
             gsub("[[:space:]]+", "\ ", x))
    }
x = c("Boris Johnson ", "Stabbed in ", "the  back", "in   spectacular", " fashion")
x
## [1] "Boris Johnson "   "Stabbed in "      "the  back"
## [4] "in   spectacular" " fashion"
trimSpace(x)
## [1] "Boris Johnson"  "Stabbed in"     "the back"       "in spectacular"
## [5] "fashion"
##  For whatever reason, it might be useful to count the number of
##  instances a pattern is found and display this in a plot of some
##  form.
url = "http://biostall.com/wp-content/uploads/2012/11/counties.csv"
counties = read.csv(url)[, 2]
pattern = c("ssex", "shire", "ham", "olk")

findPattern =
  function(input, pattern, ignore = TRUE)
    sort(sapply(pattern, function(x)
                sum(sapply(gregexpr(x, input, ignore = ignore), function(y)
                   ifelse(y[1] > 0, length(y), 0)))), dec = TRUE)
##  Plot the counts.
barplot(findPattern(counties, pattern), main = "Pattern Counts",
        border = NA, ylim = c(0, 80), col = "darkseagreen")
```



```
##  Email addresses
email = c("mdiaz@gmail.com", "brexit.or.breset?@gov.uk",
          "alle_z@boo.fr", "asd+a@what.isyourname")

emailReg = paste0("^([[:lower:][:digit:]_\\.-]+)@",
                  "([[:digit:][:lower:]\\.-]+)\\.",
                  "([[:lower:]\\.]{2,6})$")

grepl(emailReg, email)
```

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

```
### Advanced: List all the base operators in R
allObj = objects("package:base", all = TRUE)
rNames = "^[.[:alpha:]][._[[:alnum:]]*$"
repOps =  "^[.[:alpha:]][._[:alnum:]]*<-$"
s3Meth =  "[.][[:alpha:]][._[:alnum:]]*$"
miscOp = "^\\[|^\\(|\\{+|\\}+"
nonOps = paste(rNames, repOps, s3Meth, miscOp, sep = "|")
allObj[-grep(nonOps, allObj)]
```

```
##  [1] "-"    "!"    "!="   "$"    "$<-"  "%%"   "%*%"  "%/%"  "%in%" "%o%"
## [11] "%x%"  "&"    "&&"   "*"    "/"    ":"    "::"   ":::"  "@"    "@<-"
## [21] "^"    "|"    "||"   "~"    "+"    "<"    "<-"   "<<-"  "<="   "="
## [31] "=="   ">"    ">="
```