Regular Expressions with The R Language

Contents

1 Introduction

The R Project for Statistical Computing provides seven regular expression functions in its base package. The R documentation claims that the default flavor implements POSIX extended regular expressions.

That is not correct. In R 2.10.0 and later, the default regex engine is a modified version of Ville Laurikari's TRE engine. It mimics POSIX but deviates from the standard in many subtle and not-so-subtle ways. What this website says about POSIX ERE does not (necessarily) apply to R.

The best way to use regular expressions with R is to pass the perl=TRUE parameter. This tells R to use the PCRE regular expressions library. When this website talks about R, it assumes you're using the perl=TRUE parameter.

All the functions use case sensitive matching by default. You can pass `ignore.case=TRUE` to make them case insensitive. R's functions do not have any parameters to set any other matching modes. When using `perl=TRUE`, as you should, you can add mode modifiers to the start of the regex.

2 Regex Flavours

R supports two regular expression flavours: POSIX 1003.2 and Perl. Regular expression functions in R contain two arguments: extended, which defaults to TRUE, and perl, which defaults to FALSE. By default R uses POSIX extended regular expressions, though if extended is set to FALSE, it will use basic POSIX regular expressions. If perl is set to TRUE, R will use the Perl 5 flavor of regular expressions as implemented in the PCRE library.

3 Metacharacters

Regular expressions are represented as strings. Metacharacters often need to be escaped. For example, the metacharacter $\backslash w$ must be entered as $\backslash\backslash w$ to prevent *R* from interpreting the leading backslash before sending the string to the regular expression parser.

4 Search Patterns

Regular expressions allow three ways of making a search pattern more general than a single, fixed expression:

Alternatives: You can search for instances of one pattern or another, indicated by the || symbol. For example `beach||beech` matches both beach and beech.

Grouping: You group patterns together using parentheses ( ). For example you write `be(a||e)ch` to find both beach and beech.

Quantifiers: You specify whether an element in the pattern must be repeated or not by adding $*$ (occurs zero or many times) or + (occurs one or many times). For example, to find either bach or beech (zero or more of a and e but not both), you use `b(e*||a*)ch`.

5 The `grep` function

The `grep` function takes your regex as the first argument, and the input vector as the second argument. If you pass `value=FALSE` or omit the value parameter then `grep` returns a new vector with the indexes of the elements in the input vector that could be (partially) matched by the regular expression.

If you pass value=TRUE, then `grep` returns a vector with copies of the actual elements in the input vector that could be (partially) matched.

```
> grep("a+", c("abc", "def", "cba a", "aa"), perl=TRUE, value=FALSE)
[1] 1     3      4
> grep("a+", c("abc", "def", "cba a", "aa"), perl=TRUE, value=TRUE)
[1] "abc" "cba a" "aa"
```

Example:

```
grep("apple", c("crab apple", "Apple jack", "apple sauce"))
```

returns the vector $(1, 3)$ because the first and third elements of the array contain "apple." Note that grep is case-sensitive by default and so "apple" does not match "Apple." To perform a case-insensitive match, add `ignore.case = TRUE` to the function call.

6 The `grepl` function

The `grepl` function takes the same arguments as the `grep` function, except for the value argument, which is not supported. `grepl` returns a logical vector with the same length as the input vector. Each element in the returned vector indicates whether the regex could find a match in the corresponding string element in the input vector.

```
> grepl("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] TRUE  FALSE TRUE  TRUE
```

7 The `regexpr` and `gregexpr` function

The regexpr function takes the same arguments as `grepl`. regexpr returns an integer vector with the same length as the input vector. Each element in the returned vector indicates the character position in each corresponding string element in the input vector at which the (first) regex match was found. A match at the start of the string is indicated with character position 1.

If the regex could not find a match in a certain string, its corresponding element in the result vector is -1. The returned vector also has a match.length attribute. This is another integer vector with the number of characters in the (first) regex match in each string, or -1 for strings that didn't match.

gregexpr is the same as regexpr, except that it finds all matches in each string. It returns a vector with the same length as the input vector. Each element is another vector, with one element for each match found in the string indicating the character position at which that match was found. Each vector element in the returned vector also has a match.length attribute with the lengths of all matches. If no matches could be found in a particular string, the element in the returned vector is still a vector, but with just one element -1.

```
> regexpr("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1]  1 -1  3  1
attr(,"match.length")
[1]  1 -1  1  2
> gregexpr("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
[[1]]  [1] 1    attr(,"match.length")  [1] 1
[[2]]  [1] -1   attr(,"match.length")  [1] -1
[[3]]  [1] 3 5  attr(,"match.length")  [1] 1 1
[[4]]  [1] 1    attr(,"match.length")  [1] 2
```

Use regmatches to get the actual substrings matched by the regular expression. As the first argument, pass the same input that you passed to regexpr or gregexpr . As the second argument, pass the vector returned by regexpr or gregexpr. If you pass the vector from regexpr then regmatches returns a character vector with all the strings that were matched. This vector may be shorter than the input vector if no match was found in some of the elements. If you pass the vector from regexpr then regmatches returns a vector with the same number of elements as the input vector. Each element is a character vector with all the matches of the corresponding element in the input vector, or NULL if an element had no matches.

```
>x <- c("abc", "def", "cba a", "aa")
> m <- regexpr("a+", x, perl=TRUE)
> regmatches(x, m)
[1]  "a"  "a"  "aa"
> m <- gregexpr("a+", x, perl=TRUE)
> regmatches(x, m)
[[1]]  [1] "a"
[[2]]  character(0)
```

```
[[3]]  [1] "a"   "a"
[[4]]  [1] "aa"
```

8 Replacing Regex Matches in String Vectors (sub and gsub)

The `sub` function has three required parameters: a string with the regular expression, a string with the replacement text, and the input vector.

sub returns a new vector with the same length as the input vector. If a regex match could be found in a string element, it is replaced with the replacement text. Only the first match in each string element is replaced. If no matches could be found in some strings, those are copied into the result vector unchanged.

Use gsub instead of sub to replace all regex matches in all the string elements in your vector. Other than replacing all matches, gsub works in exactly the same way, and takes exactly the same arguments.

9 `strsplit`

The function strsplit also uses regular expressions, splitting its input according to a specified regular expression.

10 Back references

You can use the backreferences $/1$ through $/9$ in the replacement text to reinsert text matched by a capturing group. You cannot use backreferences to groups 10 and beyond. If your regex has named groups, you can use numbered backreferences to the first 9 groups. There is no replacement text token for the overall match. Place the entire regex in a capturing group and then use $/1$ to insert the whole regex match.

```
> sub("(a+)", "z\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zazbc"  "def"  "cbzaz a"   "zaaz"
> gsub("(a+)", "z\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zazbc"  "def"  "cbzaz zaz" "zaaz"
```

You can use $/U$ and $/L$ to change the text inserted by all following backreferences to uppercase or lowercase. You can use $/E$ to insert the following backreferences without any change of case. These escapes do not affect literal text.

```
> sub("(a+)", "z\\U\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zAzbc"  "def"  "cbzAz a"   "zAAz"
> gsub("(a+)", "z\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zAzbc"  "def"  "cbzAz zAz" "zAAz"
```

A very powerful way of making replacements is to assign a new vector to the regmatches function when you call it on the result of gregexpr. The vector you assign should have as many elements as the original input vector.

Each element should be a character vector with as many strings as there are matches in that element. The original input vector is then modified to have all the regex matches replaced with the text from the new vector.

```
> x <- c("abc", "def", "cba a", "aa")
> m <- gregexpr("a+", x, perl=TRUE)
> regmatches(x, m) <- list(c("one"), character(0), c("two", "three"), c("four"))
> x
[1]  "onebc"       "def"         "cbtwo three" "four"
```