Package 'regexPipes'

July 23, 2025

Type Package

Title Wrappers Around 'base::grep()' for Use with Pipes

Version 0.0.1

Author Billy Buchanan

Maintainer Billy Buchanan <Billy.Buchanan@fayette.kyschools.us>

Description Provides wrappers around base::grep() where the first argument is standardized to take the data object. This makes it less of a pain to use regular expressions with 'magrittr' or other pipe operators.

License GPL (>= 2)

LazyData TRUE

Suggests magrittr, pipeR

RoxygenNote 5.0.1

NeedsCompilation no

Repository CRAN

Date/Publication 2016-09-14 21:50:50

Contents

`grep` *Overriding grep from base package*

Description

Overrides the grep function to be amenable to using pipe operators. The documentation below comes from the help file for base::grep

grep, grepl, regexpr, gregexpr and regexec search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

sub and gsub perform replacement of the first and all matches respectively.

Usage

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

grepl(x, pattern, ignore.case = FALSE, perl = FALSE, fixed = FALSE,
  useBytes = FALSE)

sub(x, pattern, replacement, ignore.case = FALSE, perl = FALSE,
  fixed = FALSE, useBytes = FALSE)

gsub(x, pattern, replacement, ignore.case = FALSE, perl = FALSE,
  fixed = FALSE, useBytes = FALSE)

regexpr(text, pattern, ignore.case = FALSE, perl = FALSE, fixed = FALSE,
  useBytes = FALSE)

gregexpr(text, pattern, ignore.case = FALSE, perl = FALSE, fixed = FALSE,
  useBytes = FALSE)

regexec(text, pattern, ignore.case = FALSE, fixed = FALSE,
  useBytes = FALSE)
```

Arguments

x	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
pattern	character string containing a regular expression (or character string for fixed = TRUE) to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed except for regexpr and gregexpr.
ignore.case	if FALSE, the pattern matching is case sensitive and if TRUE, case is ignored during matching.
perl	logical. Should Perl-compatible regexps be used?
value	if FALSE, a vector containing the (integer) indices of the matches determined by grep is returned, and if TRUE, a vector containing the matching elements themselves is returned.
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character
invert	logical. If TRUE return indices or values for elements that do not match.
replacement	a replacement for matched pattern in sub and gsub. Coerced to character if possible. For fixed = FALSE this can include backreferences "\1" to "\9" to parenthesized subexpressions of pattern. For perl = TRUE only, it can also contain "\U" or "\L" to convert the rest of the replacement to upper or lower case

and "\E" to end case conversion. If a character vector of length 2 or more is supplied, the first element is used with a warning. If NA, all elements in the result corresponding to matches will be set to NA.

`text` a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.

Details

Arguments which should be character strings or character vectors are coerced to character if possible.

Each of these functions (apart from regexec, which currently does not support Perl-style regular expressions) operates in one of three modes:

1. fixed = TRUE: use exact matching.

2. perl = TRUE: use Perl-style regular expressions.

3. fixed = FALSE, perl = FALSE: use POSIX 1003.2 extended regular expressions.

See the help pages on regular expression for details of the different types of regular expressions.

The two *sub functions differ only in that sub replaces only the first occurrence of a pattern whereas gsub replaces all occurrences. If replacement contains backreferences which are not defined in pattern the result is undefined (but most often the backreference is taken to be "").

For regexpr, gregexpr and regexec it is an error for pattern to be NA, otherwise NA is permitted and gives an NA match.

The main effect of useBytes is to avoid errors/warnings about invalid inputs and spurious matches in multibyte locales, but for regexpr it changes the interpretation of the output. It inhibits the conversion of inputs with marked encodings, and is forced if any input is found which is marked as "bytes" see Encoding).

Caseless matching does not make much sense for bytes in a multibyte locale, and you should expect it only to work for ASCII characters if useBytes = TRUE.

regexpr and gregexpr with perl = TRUE allow Python-style named captures, but not for long vector inputs.

Invalid inputs in the current locale are warned about up to 5 times.

Caseless matching with PERL = TRUE for non-ASCII characters depends on the PCRE library being compiled with 'Unicode property support': an external library might not be.

Value

grep(value = FALSE) returns a vector of the indices of the elements of x that yielded a match (or not, for invert = TRUE. This will be an integer vector unless the input is a long vector, when it will be a double vector.

grep(value = TRUE) returns a character vector containing the selected elements of x (after coercion, preserving names but no other attributes).

grepl returns a logical vector (match or not for each element of x).

For sub and gsub return a character vector of the same length and with the same attributes as x (after possible coercion to character). Elements of character vectors x which are not substituted will be returned unchanged (including any declared encoding). If useBytes = FALSE a non-ASCII

substituted result will often be in UTF-8 with a marked encoding (e.g., if there is a UTF-8 input, and in a multibyte locale unless fixed = TRUE). Such strings can be re-encoded by enc2native.

regexpr returns an integer vector of the same length as text giving the starting position of the first match or -1 if there is none, with attribute "match.length", an integer vector giving the length of the matched text (or -1 for no match). The match positions and lengths are in characters unless useBytes = TRUE is used, when they are in bytes. If named capture is used there are further attributes "capture.start", "capture.length" and "capture.names".

gregexpr returns a list of the same length as text each element of which is of the same form as the return value for regexpr, except that the starting positions of every (disjoint) match are given.

regexec returns a list of the same length as text each element of which is either -1 if there is no match, or a sequence of integers with the starting positions of the match and all substrings corresponding to parenthesized subexpressions of pattern, with attribute "match.length" a vector giving the lengths of the matches (or -1 for no match).

Warning

POSIX 1003.2 mode of gsub and gregexpr does not work correctly with repeated word-boundaries (e.g., pattern = "\b"). Use perl = TRUE for such matches (but that may not work as expected with non-ASCII inputs, as the meaning of 'word' is system-dependent).

Performance considerations

If you are doing a lot of regular expression matching, including on very long strings, you will want to consider the options used. Generally PCRE will be faster than the default regular expression engine, and fixed = TRUE faster still (especially when each pattern is matched only a few times).

If you are working in a single-byte locale and have marked UTF-8 strings that are representable in that locale, convert them first as just one UTF-8 string will force all the matching to be done in Unicode, which attracts a penalty of around 3x for the default POSIX 1003.2 mode.

If you can make use of useBytes = TRUE, the strings will not be checked before matching, and the actual matching will be faster. Often byte-based matching suffices in a UTF-8 locale since byte patterns of one character never match part of another.

Source

The C code for POSIX-style regular expression matching has changed over the years. As from R 2.10.0 the TRE library of Ville Laurikari (http://laurikari.net/tre/) is used. The POSIX standard does give some room for interpretation, especially in the handling of invalid regular expressions and the collation of character ranges, so the results will have changed slightly over the years.

For Perl-style matching PCRE (http://www.pcre.org) is used.

Examples

```
library(magrittr)
letters %>% regexPipes::grep("[a-z]")

txt <- c("arm","foot","lefroo", "bafoobar")
if(length(i <- txt %>% regexPipes::grep("foo")))
   cat("'foo' appears at least once in\n\t", txt, "\n")
```

```
i # 2 and 4
txt[i]

## Double all 'a' or 'b's;  "\" must be escaped, i.e., 'doubled'
gstext <- "abc and ABC"
gstext %>% regexPipes::gsub("([ab])", "\\1_\\1_")

txt <- c("The", "licenses", "for", "most", "software", "are",
  "designed", "to", "take", "away", "your", "freedom",
  "to", "share", "and", "change", "it.",
   "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
   "is", "intended", "to", "guarantee", "your", "freedom", "to",
   "share", "and", "change", "free", "software", "--",
   "to", "make", "sure", "the", "software", "is",
   "free", "for", "all", "its", "users")
( i <- txt %>% regexPipes::grep("[gu]") ) # indices
stopifnot( txt[i] == txt %>% regexPipes::grep("[gu]", value = TRUE) )

## Note that in locales such as en_US this includes B as the
## collation order is aAbBcCdEe ...
(ot <- txt %>% regexPipes::sub("[b-e]", "."))
txt[ot != txt %>% regexPipes::gsub("[b-e]", ".")]#- gsub does "global" substitution

txt[txt %>% regexPipes::gsub("g", "#") !=
    txt %>% regexPipes::gsub("g", "#", ignore.case = TRUE)] # the "G" words

txt %>% regexPipes::regexpr("en")

txt %>% regexPipes::gregexpr("e")

## Using grepl() for filtering
## Find functions with argument names matching "warn":
findArgs <- function(env, pattern) {
  nms <- ls(envir = as.environment(env))
  nms <- nms[is.na(match(nms, c("F","T")))] # <-- work around "checking hack"
  aa <- sapply(nms, function(.) { o <- get(.)
               if(is.function(o)) names(formals(o)) })
  iw <- sapply(aa, function(a) any(a %>% regexPipes::grepl(pattern, ignore.case=TRUE)))
  aa[iw]
}
findArgs("package:base", "warn")

## trim trailing white space
str <- "Now is the time      "
str %>% regexPipes::sub(" +$", "")  ## spaces only
## what is considered 'white space' depends on the locale.
str %>% regexPipes::sub("[[:space:]]+$", "") ## white space, POSIX-style
## what PCRE considered white space changed in version 8.34: see ?regex
str %>% regexPipes::sub("\\s+$", "", perl = TRUE) ## PCRE-style white space

## capitalizing
txt <- "a test of capitalizing"
txt %>% regexPipes::gsub("(\\w)(\\w*)", "\\U\\1\\L\\2",  perl=TRUE)
```

```
txt %>% regexPipes::gsub("\\b(\\w)", "\\U\\1",  perl=TRUE)

txt2 <- "useRs may fly into JFK or laGuardia"
txt2 %>% regexPipes::gsub("(\\w)(\\w*)(\\w)", "\\U\\1\\E\\2\\U\\3", perl=TRUE)
txt2 %>%  regexPipes::sub("(\\w)(\\w*)(\\w)", "\\U\\1\\E\\2\\U\\3", perl=TRUE)

## named capture
notables <- c("  Ben Franklin and Jefferson Davis",
              "\tMillard Fillmore")
# name groups 'first' and 'last'
name.rex <- "(?<first>[[:upper:]][[:lower:]]+) (?<last>[[:upper:]][[:lower:]]+)"
(parsed <- notables %>% regexPipes::regexpr(name.rex, perl = TRUE))

# Example below fails, but unclear what the cause is
# notables %>% regexPipes::gregexpr(name.rex, perl = TRUE)[[2]]
# Same example with the object passed w/o magrittr does work
regexPipes::gregexpr(notables, name.rex, perl = TRUE)[[2]]
parse.one <- function(res, result) {
  m <- do.call(rbind, lapply(seq_along(res), function(i) {
    if(result[i] == -1) return("")
    st <- attr(result, "capture.start")[i, ]
    substring(res[i], st, st + attr(result, "capture.length")[i, ] - 1)
  }))
  colnames(m) <- attr(result, "capture.names")
  m
}
parse.one(notables, parsed)

## Decompose a URL into its components.
## Example by LT (http://www.cs.uiowa.edu/~luke/R/regexp.html).
x <- "http://stat.umn.edu:80/xyz"
m <- x %>% regexPipes::regexec("^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)")
m
regmatches(x, m)
## Element 3 is the protocol, 4 is the host, 6 is the port, and 7
## is the path.  We can use this to make a function for extracting the
## parts of a URL:
URL_parts <- function(x) {
    m <- x %>% regexPipes::regexec("^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)")
    parts <- do.call(rbind,
                     lapply(regmatches(x, m), `[`, c(3L, 4L, 6L, 7L)))
    colnames(parts) <- c("protocol","host","port","path")
    parts
}
URL_parts(x)

## There is no gregexec() yet, but one can emulate it by running
## regexec() on the regmatches obtained via gregexpr().  E.g.:
pattern <- "([[:alpha:]]+)([[:digit:]]+)"
s <- "Test: A1 BC23 DEF456"
lapply(regmatches(s, s %>% regexPipes::gregexpr(pattern)),
       function(e) regmatches(e, e %>% regexPipes::regexec(pattern)))
```

Index