Regular Expression in R

Gloria Li and Jenny Bryan
October 19, 2014

In this tutorial, we will use the Gapminder data and file names in our class repository (https://github.com/STAT545-UBC/STAT545-UBC.github.io) as examples to demonstrate using regular expression in R. First, let's start off by cloning the class repository, getting the list of file names with `list.files()`, and load the Gapminder dataset into R.

We will also need to use some functions from the stringr (https://github.com/hadley/stringr) package. It provids a clean, modern alternative to common string operations, and is sometimes easier to remember and use than R basic string functions. If you have not done so yet, install the package.

```
install.packages("stringr")
```

```
library(stringr)
files <- list.files()
head(files)
```

```
## [1] "_output.yaml"               "automation00_index.html"
## [3] "automation00_index.md"      "automation01_slides"
## [5] "automation02_windows.html" "automation02_windows.md"
```

```
gDat <- read.delim("gapminderDataFiveYear.txt")
str(gDat)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1
1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997
...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3
3 3 3 3 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

Now we can use some string functions to extract certain filenames, say all documents on `dplyr`. We can use `grep()` function to identify filenames including the string `dplyr`. If we set the argument `value = TRUE`, `grep()` returns the matches, while `value = FALSE` returns their indices. The `invert` argument let's you get everything BUT the pattern you specify. `grepl()` is a similar function but returns a logical vector. See here (http://www.rdocumentation.org/packages/base/functions/grep) for more information.

```
grep("dplyr", files, value = TRUE)
```

```
##  [1] "bit001_dplyr-cheatsheet.html"
##  [2] "bit001_dplyr-cheatsheet.md"
##  [3] "bit001_dplyr-cheatsheet.rmd"
##  [4] "block0_dplyr-fake.rmd"
##  [5] "block000_dplyr-fake.rmd.txt"
##  [6] "block009_dplyr-intro.html"
##  [7] "block009_dplyr-intro.md"
##  [8] "block009_dplyr-intro.rmd"
##  [9] "block010_dplyr-end-single-table.html"
## [10] "block010_dplyr-end-single-table.md"
## [11] "block010_dplyr-end-single-table.rmd"
## [12] "cm007_dplyr-intro.html"
## [13] "cm007_dplyr-intro.md"
## [14] "hw03_dplyr-and-more-ggplot2.html"
## [15] "hw03_dplyr-and-more-ggplot2.md"
## [16] "xblock000_dplyr-fake.rmd"
```

```
grep("dplyr", files, value = FALSE)
```

```
##  [1]  12  13  14  23  24  43  44  45  46  47  48 114 115 186 187 267
```

```
grepl("dplyr", files)
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [12]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [23]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TR
UE
##  [45]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [111] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [177] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TR
UE
## [188] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [199] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [210] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [221] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
```

```
SE
## [232] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [243] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [254] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAL
SE
## [265] FALSE FALSE  TRUE
```

What if we wanted to extract all homework files on `dplyr`? We would need a way to specify matching a string containing `hw` and then something and then `dplyr`. This is where regular expressions come in handy.

String functions related to regular expression

Regular expression is a pattern that describes a specific set of strings with a common structure. It is heavily used for string matching / replacing in all programming languages, although specific syntax may differ a bit. It is truly the heart and soul for string operations. In R, many string functions in `base` R as well as in `stringr` package use regular expressions, even Rstudio's search and replace allows regular expression, we will go into more details about these functions later this week:

- identify match to a pattern: `grep(..., value = FALSE)`, `grepl()`, `stringr::str_detect()`
- extract match to a pattern: `grep(..., value = TRUE)`, `stringr::str_extract()`, `stringr::str_extract_all()`
- locate pattern within a string, i.e. give the start position of matched patterns. `regexpr()`, `gregexpr()`, `stringr::str_locate()`, `string::str_locate_all()`
- replace a pattern: `sub()`, `gsub()`, `stringr::str_replace()`, `stringr::str_replace_all()`
- split a string using a pattern: `strsplit()`, `stringr::str_split()`

Regular expression syntax

Regular expressions typically specify characters (or character classes) to seek out, possibly with information about repeats and location within the string. This is accomplished with the help of metacharacters that have specific meaning: `$ * + . ? [ ] ^ { } | ( ) \`. We will use some small examples to introduce regular expression syntax and what these metacharacters mean.

Escape sequences

There are some special characters in R that cannot be directly coded in a string. For example, let's say you specify your pattern with single quotes and you want to find countries with the single quote `'`. You would have to "escape" the single quote in the pattern, by preceding it with `\`, so it's clear it is not part of the string-specifying machinery:

```
grep('\'', levels(gDat$country), value = TRUE)
```

```
## [1] "Cote d'Ivoire"
```

There are other characters in R that require escaping, and this rule applies to all string functions in R, including regular expressions. See here (https://stat.ethz.ch/R-manual/R-devel/library/base/html/Quotes.html) for a complete list of R esacpe sequences.

- `\'`: single quote. You don't need to escape single quote inside a double-quoted string, so we can also use `"'"` in the previous example.
- `\"`: double quote. Similarly, double quotes can be used inside a single-quoted string, i.e. `'"'`.
- `\n`: newline.
- `\r`: carriage return.
- `\t`: tab character.

> *Note:* `cat()` *and* `print()` *to handle escape sequences differently, if you want to print a string out with these sequences interpreted, use* `cat()`*.*

```
print("a\nb")
```

```
## [1] "a\nb"
```

```
cat("a\nb")
```

```
## a
## b
```

Quantifiers

Quantifiers specify how many repetitions of the pattern.

- `*` : matches at least 0 times.
- `+` : matches at least 1 times.
- `?` : matches at most 1 times.
- `{n}` : matches exactly n times.
- `{n,}` : matches at least n times.
- `{n,m}` : matches between n and m times.

```
(strings <- c("a", "ab", "acb", "accb", "acccb", "accccb"))
```

```
## [1] "a"      "ab"     "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac*b", strings, value = TRUE)
```

```
## [1] "ab"     "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac+b", strings, value = TRUE)
```

```
## [1] "acb"    "accb"   "acccb"  "accccb"
```

```
grep("ac?b", strings, value = TRUE)
```

```
## [1] "ab"  "acb"
```

```
grep("ac{2}b", strings, value = TRUE)
```

```
## [1] "accb"
```

```
grep("ac{2,}b", strings, value = TRUE)
```

```
## [1] "accb"   "acccb"  "accccb"
```

```
grep("ac{2,3}b", strings, value = TRUE)
```

```
## [1] "accb"  "acccb"
```

Exercise

Find all countries with `ee` in Gapminder using quantifiers.

```
## [1] "Greece"
```

Position of pattern within the string

- `^` : matches the start of the string.
- `$` : matches the end of the string.
- `\b` : matches the empty string at either edge of a *word*. Don't confuse it with `^ $` which marks the edge of a *string*.
- `\B` : matches the empty string provided it is not at an edge of a word.

```
(strings <- c("abcd", "cdab", "cabd", "c abd"))
```

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

```
grep("ab", strings, value = TRUE)
```

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

```
grep("^ab", strings, value = TRUE)
```

```
## [1] "abcd"
```

```
grep("ab$", strings, value = TRUE)
```

```
## [1] "cdab"
```

```
grep("\\bab", strings, value = TRUE)
```

```
## [1] "abcd"  "c abd"
```

Exercise

Find all `.txt` files in the repository.

```
## [1] "block000_dplyr-fake.rmd.txt"     "gapminderDataFiveYear_dirty.tx
t"
## [3] "gapminderDataFiveYear.txt"       "note-to-alums.txt"
```

Operators

- `.` : matches any single character, as shown in the first example.
- `[...]` : a character list, matches any one of the characters inside the square brackets. We can also use `-` inside the brackets to specify a range of characters.
- `[^...]` : an inverted character list, similar to `[...]`, but matches any characters **except** those inside the square brackets.
- `\` : suppress the special meaning of metacharacters in regular expression, i.e. `$ * + . ? [ ] ^ { } | ( ) \`, similar to its usage in escape sequences. Since `\` itself needs to be escaped in R, we need to escape

these metacharacters with double backslash like `\\$`.

- `|`: an "or" operator, matches patterns on either side of the `|`.
- `(...)`: grouping in regular expressions. This allows you to retrieve the bits that matched various parts of your regular expression so you can alter them or use them for building up a new string. Each group can than be refer using `\\N`, with N being the No. of `(...)` used. This is called **backreference**.

```
(strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12"))
```

```
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 12"
```

```
grep("ab.", strings, value = TRUE)
```

```
## [1] "abc"   "abd"   "abe"   "ab 12"
```

```
grep("ab[c-e]", strings, value = TRUE)
```

```
## [1] "abc" "abd" "abe"
```

```
grep("ab[^c]", strings, value = TRUE)
```

```
## [1] "abd"   "abe"   "ab 12"
```

```
grep("^ab", strings, value = TRUE)
```

```
## [1] "ab"    "abc"   "abd"   "abe"   "ab 12"
```

```
grep("\\^ab", strings, value = TRUE)
```

```
## [1] "^ab"
```

```
grep("abc|abd", strings, value = TRUE)
```

```
## [1] "abc" "abd"
```

```
gsub("(ab) 12", "\\1 34", strings)
```

```
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 34"
```

Excercise

Find countries in Gapminder with letter `i` or `t`, and ends with `land`, and replace `land` with `LAND` using backreference.

```
## [1] "FinLAND"     "IceLAND"     "IreLAND"     "SwaziLAND"   "SwitzerL
AND"
## [6] "ThaiLAND"
```

Character classes

Character classes allows to – surprise! – specify entire classes of characters, such as numbers, letters, etc. There are two flavors of character classes, one uses `[:` and `:]` around a predefined name inside square brackets and the other uses `\` and a special character. They are sometimes interchangeable.

- `[:digit:]` or `\d`: digits, 0 1 2 3 4 5 6 7 8 9, equivalent to `[0-9]`.
- `\D`: non-digits, equivalent to `[^0-9]`.
- `[:lower:]`: lower-case letters, equivalent to `[a-z]`.
- `[:upper:]`: upper-case letters, equivalent to `[A-Z]`.
- `[:alpha:]`: alphabetic characters, equivalent to `[[:lower:][:upper:]]` or `[A-z]`.
- `[:alnum:]`: alphanumeric characters, equivalent to `[[:alpha:][:digit:]]` or `[A-z0-9]`.
- `\w`: word characters, equivalent to `[[:alnum:]_]` or `[A-z0-9_]`.
- `\W`: not word, equivalent to `[^A-z0-9_]`.
- `[:xdigit:]`: hexadecimal digits (base 16), 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f, equivalent to `[0-9A-Fa-f]`.
- `[:blank:]`: blank characters, i.e. space and tab.
- `[:space:]`: space characters: tab, newline, vertical tab, form feed, carriage return, space.

- `\s` : space, ` `.
- `\S` : not space.
- `[:punct:]` : punctuation characters, ! " # $ % & ’ ( ) * + , - . / : ; < = > ? @ [ ] ^ _ ` { | } ~.
- `[:graph:]` : graphical (human readable) characters: equivalent to `[[:alnum:][:punct:]]`.
- `[:print:]` : printable characters, equivalent to `[[:alnum:][:punct:]\\s]`.
- `[:cntrl:]` : control characters, like `\n` or `\r`, `[\x00-\x1F\x7F]`.

Note:

- `[:...:]` has to be used inside square brackets, e.g. `[[:digit:]]`.
- `\` itself is a special character that needs escape, e.g. `\\d`. Do not confuse these regular expressions with R escape sequences such as `\t`.

General modes for patterns

There are different syntax standards (http://en.wikipedia.org/wiki/Regular_expression#Standards) for regular expressions, and R offers two:

- POSIX extended regular expressions (default)
- Perl-like regular expressions.

You can easily switch between by specifying `perl = FALSE/TRUE` in `base` R functions, such as `grep()` and `sub()`. For functions in the `stringr` package, wrap the pattern with `perl()`. The syntax between these two standards are a bit different sometimes, see an example here (http://www.inside-r.org/packages/cran/stringr/docs/perl). If you had previous experience with Python or Java, you are probably more familiar with the Perl-like mode. But for this tutorial, we will only use R’s default POSIX standard.

There’s one last type of regular expression – “fixed”, meaning that the pattern should be taken literally. Specify this via `fixed = TRUE` (base R functions) or wrapping with `fixed()` (`stringr` functions). For example, `"A.b"` as a regular expression will match a string with "A" followed by any single character followed by "b", but as a fixed pattern, it will only match a literal "A.b".

```
(strings <- c("Axbc", "A.bc"))
```

```
## [1] "Axbc" "A.bc"
```

```
pattern <- "A.b"
grep(pattern, strings, value = TRUE)
```

```
## [1] "Axbc" "A.bc"
```

```
grep(pattern, strings, value = TRUE, fixed = TRUE)
```

```
## [1] "A.bc"
```

By default, pattern matching is case sensitive in R, but you can turn it off with `ignore.case = TRUE` (base R functions) or wrapping with `ignore.case()` (`stringr` functions). Alternatively, you can use `tolower()` and `toupper()` functions to convert everything to lower or upper case. Take the same example above:

```
pattern <- "a.b"
grep(pattern, strings, value = TRUE)
```

```
## character(0)
```

```
grep(pattern, strings, value = TRUE, ignore.case = TRUE)
```

```
## [1] "Axbc" "A.bc"
```

Exercise

Find continents in Gapminder with letter `o` in it.

```
## [1] "Europe"  "Oceania"
```

Examples

As an example, let's try to integrate everything together, and find all course materials on `dplyr` and extract the topics we have covered. These files all follow our naming strategy: `block` followed by 3 digits, then `_`, then topic. As you can see from the topic index (http://stat545-ubc.github.io/topics.html), we had two blocks on `dplyr`: the intro (http://stat545-ubc.github.io/block009_dplyr-intro.html), and verbs for a single dataset (http://stat545-ubc.github.io/block010_dplyr-end-single-table.html). We'll try to extract the `.rmd` filenames for these blocks. To make the task a bit harder, I also put a few fake files inside the repository that don't quite match our naming strategy!

We know that the filename should have `block` and `dplyr` in it, and is a Rmd file, so what if we just put these three parts together?

```
pattern <- "block.*dplyr.*rmd"
grep(pattern, files, value = TRUE)
```

```
## [1] "block0_dplyr-fake.rmd"
## [2] "block000_dplyr-fake.rmd.txt"
## [3] "block009_dplyr-intro.rmd"
## [4] "block010_dplyr-end-single-table.rmd"
## [5] "xblock000_dplyr-fake.rmd"
```

Apart from the two files we wanted, we also got three fake ones: block0_dplyr-fake.rmd, block000_dplyr-fake.rmd.txt, xblock000_dplyr-fake.rmd. Looks like our pattern is not stringent enough. The first fake file does not have 3 digits after `block`, second one does not start with `block`, and last one has `.txt` after `rmd`. So let's try to fix that:

```
pattern <- "^block\\d{3}_.*dplyr.*rmd$"
(dplyr_file <- grep(pattern, files, value = TRUE))
```

```
## [1] "block009_dplyr-intro.rmd"
## [2] "block010_dplyr-end-single-table.rmd"
```

Now we have the two file names stored in `dplyr_file`, let's try to extract the topics out.

One way to do that is to use a substitution function like `sub()`, `gsub()`, or `str_sub()` to replace anything before and after the topic with empty strings:

```
(dplyr_topic <- gsub("^block\\d{3}_.*dplyr-", "", dplyr_file))
```

```
## [1] "intro.rmd"            "end-single-table.rmd"
```

```
(dplyr_topic <- gsub("\\.rmd", "", dplyr_topic))
```

```
## [1] "intro"            "end-single-table"
```

Alternatively, instead of using `grep() + gsub()`, we can use `str_match()`. As mentioned above, this function will give specific matches for patterns enclosed with `()` operator. We just need to reconstruct our regular expression to specify the topic part:

```
pattern <- "^block\\d{3}_.*dplyr-(.*)\\.rmd$"
(na.omit(str_match(files, pattern)))
```

```
##      [,1]                                  [,2]
## [1,] "block009_dplyr-intro.rmd"            "intro"
## [2,] "block010_dplyr-end-single-table.rmd" "end-single-table"
## attr(,"na.action")
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33
34
##  [35]  35  36  37  38  39  40  41  42  43  44  46  47  49  50  51  52
53
##  [52]  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
70
##  [69]  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
87
##  [86]  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103
104
## [103] 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121
## [120] 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137
138
## [137] 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154
155
## [154] 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
172
## [171] 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188
189
## [188] 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205
206
## [205] 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222
223
## [222] 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240
## [239] 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256
257
## [256] 258 259 260 261 262 263 264 265 266 267
## attr(,"class")
## [1] "omit"
```

The second column of the result data frame gives the topic we needed.

Some more advanced string functions

There are some more advanced string functions that are somewhat related to regular expression, like splitting a string, get a subset of a string, pasting strings together etc. These functions are very useful for data cleaning, and we will get into more details about them later this week. Here is a short introduction with above example.

From above example, we got two topics on `dplyr`: . We can use `strsplit()` function to split the second one, , into words. The second argument `split` is a regular expression used for splitting, and the function will return a list. We can use `unlist()` function to convert the list into a character vector. Or an alternative function `str_split_fixed()` will return a data frame.

```
(topic_split <- unlist(strsplit(dplyr_topic[2], "-")))
```

```
## [1] "end"    "single" "table"
```

```
(topic_split <- str_split_fixed(dplyr_topic[2], "-", 3)[1, ])
```

```
## [1] "end"    "single" "table"
```

We can also use `paste()` or `paste0()` functions to put them back together. `paste0()` function is equivalent to `paste()` with `sep = ""`. We can use `collapse = "-"` argument to concatenate a character vector into a string:

```
paste(topic_split, collapse = "-")
```

```
## [1] "end-single-table"
```

Another useful function is `substr()`. It can be used to extract a part of a string with start and end positions. For example, to extract the first three letters in `dplyr_topic`:

```
substr(dplyr_topic, 1, 3)
```

```
## [1] "int" "end"
```

Exercise

Get all markdown documents on peer review and extract the specific topics.

> *Hint: file names should start with `peer-review`.*

```
## marking-rubric,  peer-evaluation-guidelines
```

Regular expression vs shell (git09_shell.html) globbing

The term globbing in shell (git09_shell.html) or Unix-like environment refers to pattern matching based on wildcard characters. A wildcard character can be used to substitute for any other character or characters in a string. Globbing is commonly used for matching file names or paths, and has a much simpler syntax. It is somewhat similar to regular expressions, and that's why people are often confused between them. Here is a list of globbing syntax and their comparisons to regular expression:

- `*`: matches any number of unknown characters, same as `.*` in regular expression.
- `?`: matches one unknown character, same as `.` in regular expression.
- `\`: same as regular expression.
- `[...]`: same as regular expression.
- `[!...]`: same as `[^...]` in regular expression.

Resources

- Regular expression in R official document (https://stat.ethz.ch/R-manual/R-devel/library/base/html/regex.html).
- Perl-like regular expression: regular expression in perl manual (http://perldoc.perl.org/perlre.html#Regular-Expressions).
- `qdapRegex` package (http://trinkerrstuff.wordpress.com/2014/09/27/canned-regular-expressions-qdapregex-0-1-2-on-cran/): a collection of handy regular expression tools, including handling abbreviations, dates, email addresses, hash tags, phone numbers, times, emoticons, and URL etc.
- Recently, there are some attemps to create human readable regular expression packages, Regularity (https://github.com/andrewberls/regularity) in Ruby is a very successful one. Unfortunately, its implementation in R is still quite beta at this stage, not as friendly as Regularity yet. But keep an eye out, better packages may become available in the near future!
- There are some online tools to help learn, build and test regular expressions. On these websites, you can simply paste your test data and write regular expression, and matches will be highlighted.
- regexpal (http://regexpal.com/)
- RegExr (http://www.regexr.com/)
