Regular Expressions: Details and Tips

Michael Assmus

12/2/2017

Introduction:

Regular expressions, commonly referred to by the abbreviation “regex”, are strings of characters that are read in a specific way in order to match desired patterns within text. While we did make some use of them towards the end of Stat 133, we never went too deep into what they were, how they worked, and what their uses may be. I wanted to do some more exploration on that front, and I hope to provide some useful information about the details of regex and some of the various functions that make use of it.

Note that no particular packages are required for the code used in this post. All functions used here are available in baseline R. Additionally, no code has been hidden; all examples should be fully reproducible using the visibly associated code.

History:

For the sake of information, I decided I might as well include a bit of the background of regex here. The term “regular expressions” was first coined in 1956 by the mathematician Stephen Kleene, who was using a notation called “regular sets” to describe a type of formal language called (unsurprisingly) “regular languages”. Thirteen years later, in 1968, regular expressions entered common use with Unix creator Ken Thompson’s article “Regular Expression Search Algorithm”, with one of the primary uses being pattern matching in text. Becoming yet more mainstream with its use in the Perl programming language in the 1980’s, Regex has since been expanded and refined, and is now included as standard in many programming languages, including Python, Perl, and (to at least some extent) R.

Internal Functionality:

While regular expressions can be used in various ways, in this post I’ll be focusing on the use of regular expressions in text matching, which was the way we used it in class. Text matching with regex operates by taking in a regex (which we’ll call the “search term”) and a string. The idea is to search through the string for something that matches the search term. The way the program does this is by splitting up the search term into “tokens” and then attempting to match these tokens to characters in the string in succession. It will start by attempting to match the first token to the first character in the string. If that matches, then it will try to match the second token to the second character, and so on. If at any point it is unable to match a token to a character, it will abort that matching attempt, and will go back to try to match the search term to the string starting from the *second* character. This principle holds for the rest of the string, with the program continuing to attempt new starting points until it either finds a match or runs out of characters.

We’ll look at a few examples here. For now we’ll just be using the basic `grep` function, which looks for a given regex in each string in a vector of strings.

```
#We'll look for the simple string "cat" within several different strings
grep("cat", c("cat"), perl = TRUE, value = TRUE)
```

```
## [1] "cat"
```

```
grep("cat", c("catfish"), perl = TRUE, value = TRUE)
```

```
## [1] "catfish"
```

```
grep("cat", c("dogfish"), perl = TRUE, value = TRUE)
```

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

These examples are pretty self-explanatory. The function runs through the given string looking for the regex, and if it finds it it returns that string. What’s less obvious, but which can be important under certain circumstances and with certain functions, is that the regex engine will always return the first match it encounters, which, due to the way it goes through the string, will be the leftmost match. This can be seen using the `regexpr` function, which displays the position of the found match in the given string. It also displays a “match length” vector showing the length of the matched string, along with a “useBytes” logical result, but you can ignore those for the moment.

```
#This time we'll put the input strings together in a single vector for simplicity
regexpr("cat", c("cat", "a cat catastrophe", "a dog cataclysm"), perl = TRUE)
```

```
## [1] 1 3 7
## attr(,"match.length")
## [1] 3 3 3
## attr(,"useBytes")
## [1] TRUE
```

As you can see in that first vector, the regex engine found the word “cat” at character 3 in the second string and stopped at that point, not moving on to find the second possible match at the beginning of “catastrophe”. In the third string, however, that first “cat” wasn’t present, so it went all the way through to the beginning of “cataclysm”.

While this may not seem particularly useful at first glance, it’s much more important when used in conjunction with more advanced expressions. As you probably already know from your work in Stat 133 or similar classes, the real power of regex comes from metacharacters, specific regex tokens that have special meanings when interpreted by the regex engine. A number of these tokens are capable of matching with more than one particular character, which means that if there are multiple words (or other groups of characters) in the input string that can be matched with the regex, the regex engine will always take whichever one of those words it first comes across in the string.

Here's an example, using the function "regmatches", which takes a vector of strings and the output of the regexpr function and returns the matched words. Recall that '.' is a metacharacter that can match with any single character.

```
#regmatches takes a vector of strings and a set of positions and lengths, so
#we'll make those separately
strings <- c("cat", "a car campaign", "a bad campaign")
places <- regexpr("ca.", strings, perl = TRUE)

#and now we'll plug our inputs into regmatches
regmatches(strings, places)
```

```
## [1] "cat" "car" "cam"
```

As you can see, in the second string, even though there were two different substrings that could have matched with "ca." (specifically "car" and "cam"), "car" was the first match the function came across, so it was the one that was returned. In the third string, there was no substring ahead of "campaign" that could match with the given regex, so the "cam" at the beginning of "campaign" was the substring that was first encountered and thus returned. This search order should be kept in mind while using regex to return particular substrings from strings that could have multiple possible matches.

Syntax and Pitfalls

While the search order is useful to know, it's not the only thing to watch out for. One of the more frustrating obstacles I ran into while working with regex for the first time was the way long strings are formatted when they're coming from certain sources. Specifically, some text formats are organized into lines and paragraphs, and they do this using certain syntax. Rather than having a new line be a separate string, such text is frequently a single long string, with the token '\n' used to denote a new line. An example:

```
#you might expect
#this text to be split up
#like this
grep('...', c('you might expect', 'this text to be split up', 'like this'),
     perl = TRUE, value = TRUE)
```

```
## [1] "you might expect"         "this text to be split up"
## [3] "like this"
```

```
#but it will often
#be more
#like this
grep('...', c('but it will often \n be more \n like this'),
     perl = TRUE, value = TRUE)
```

```
## [1] "but it will often \n be more \n like this"
```

Once you know about it, it isn't necessarily much of an issue, but if you *don't* know about it, it can be extremely frustrating. One such case is if you're trying to find a substring at the end of a line; if you think the lines are separate strings, you might try to use the '$' token (which denotes the end of a string) to denote that you're looking at the end of a line, but if the lines are actually split up by newline metacharacters ('\n'), then you're unlikely to match with anything before the final line.

Similar issues hold if you're trying to match things at the beginning of lines using the metacharacter for the start of a string ('^'), or if you're trying to match a string that extends across multiple lines and you aren't expecting the newline metacharacters to be interspersed with the text.

Long story short, make sure you understand the syntax of whatever strings you're trying to match using regex. It can save you a lot of time and frustration.

Conclusion

As with most code and programming languages, regex is very useful, but if you don't know certain details about how it works, you can end up running into horrible frustration and delay. As I've shown, misunderstandings about regex or your inputs can end up causing your programs to do anything from producing the wrong outputs to completely skipping most of the input text. While I can't hope to be anything close to comprehensive with this single post, I do hope that the tips and suggestions I've given will prove useful to one of you someday. Keep an eye on those inputs, and good coding!

Sources:

- Regular-Expressions.info: Explains details of inner workings of Regex. https://www.regular-expressions.info/engine.html
- Regular-Expressions.info: Talks about Regex functions in R, not counting packages. https://www.regular-expressions.info/rlanguage.html
- Introduction to Stringr: Talks about functions in Stringr package. https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html
- Wikipedia: Regular Expressions: Gives bits of the history behind Regex, along with other miscellaneous details. https://en.wikipedia.org/wiki/Regular_expression
- Wikipedia: Regular Languages: Gives some details about regular languages. https://en.wikipedia.org/wiki/Regular_language
- Basic Regex Expressions in R: Cheat Sheet: Gives summary of regex functions, syntax, and other useful information. https://github.com/ucb-stat133/stat133-fall-2017/blob/master/cheat-sheets/regular-expressions-cheatsheet.pdf
- Regular Expressions: A Brief History: Gives information about the history of regex. https://blog.staffannoteberg.com/2013/01/30/regular-expressions-a-brief-history/