Regular Expressions for Humanists

Tutorial author: Professor Martin Paul Eve
Birkbeck, University of London
14 June 2017

blackbox.llc.ed.ac.uk

1. What are regular expressions, and why learn them?

Regular expressions (also known as regex or regexp) are a widely-used standard for extracting, matching, or replacing text. They are sometimes called PCRE (Perl-Compatible Regular Expressions), regexp or regex, with the plural jokingly being regexen, and they have their origins in various language theories.

Regular expressions are a way of quickly performing advanced search-and-replace operations across a single text or range of texts. With a regular expression you can, for example, match different verb forms, extract words with either British or American spellings, find addresses, or parse dates. This can be very useful when you need to wrangle your data into other formats, for instance converting unstructured plain text data into a csv file.

Regular expressions are not an independent language in themselves but rather a set of scripting tools that are usually used inside a web environment, within another programming language, or in a text editor. Most modern programming languages have the capacity to run regular expressions. In general, regular expressions work like a search tool, and they run from left to right (unless you are working with right-to-left text; in this tutorial we are dealing only with left-to-right text).

Any literal text within a regular expression will literally match the search. The word `test`, for instance, is a valid regular expression that will match the word "test" within a string. At the same time, `^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$` is also a valid regular expression, which raises an important point. When you know what you are doing, regular expressions are easy to write, but they are usually *difficult to read*. You should, therefore, take care in your documentation of any regular expressions that you write so that, when you revisit them at a later stage, you can understand what you meant.

2. How can I use regular expressions?

To use a regular expression, you need:

- An expression itself
- A body of text on which to operate
- An application with which to implement the regular expression

Many word processing tools, from Microsoft Word through to command-line utilities such as sed and awk, can use regular expressions to search and find text. Languages such as Python, Perl, C#, Java, C++ and many others have the capability of using regular expressions to parse text; you

will need to consult the language or tool of your choice for specific implementations.

There are other online environments that yield different backend environments. For instance, if you intend to work primarily with the .NET Framework, you may wish to find an online testing tool that is specifically written in .NET to avoid any quirks.

At the time of writing, there are a range of online tools which are useful for learning regular expressions:

- https://regex101.com/ : a powerful online tool that explains the operation of regular expressions in human-readable terms while providing a real-time editing environment for testing. It allows the user to specify the "flavour" of regular expression implementation that they wish to use (eg. the "flavour" of regex used within Python).
- http://regexr.com : another good online tool. The advantage of this site is the quick online "cheatsheet" in the left-hand menu.
- http://www.regexpal.com : another good tool. There isn't much difference between this and other tools.

In order to do the exercises in the remainder of this tutorial, please open one of these sites. Put the text you are searching into the box labelled "text" or "test string", and the regular expression into the box labelled "expression" or "regular expression". Note that any regular expressions or other text that you may need to type will be given in `Courier font`.

3. What does a regular expression look like?

A regular expression consists of a series of syntactic elements which match text elements within the body of text on which it is operating.

A basic regular expression might look like this: `\d{1,4}`

Here, the element `\d` means "match a **d**igit", while the phrase `{1,4}` is a quantifier meaning "match between 1 and 4 of the elements immediately before this phrase". So, taken together, `\d{1,4}` means "find a string of numbers that is one, two three or four digits long".

To see how this works, copy and paste the following text into the test string box:

```
In the Year of our Lord 992 the Justified Ancients of Mu Mu set
sail in their longboats on a voyage to rediscover the lost
continent.
```

Now, type the regular expression `\d{1,4}` into the regular expression box. The resulting match should be 992. This is because 9, 9, and 2 are all digits and there are three of them (between 1 and 4) in a row.

In regular expressions, text that is not part of a special syntax simply matches itself. You can see this by typing the phrase `the Justified Ancients of Mu Mu` into the regular expression box, and observing that it matches "the Justified Ancients of Mu Mu". This phrase is also, then, a valid regular expression. (Note that regular expressions are usually case sensitive unless a flag is passed to the engine to disable this).

4. What are the basic elements of regular expression syntax?

The following are special syntactic aspects of regular expressions, ie. characters that match something other than their literal meaning:

`.`	The dot matches any single character
`\n`	Matches a newline character
`\t`	Matches a tab
`\d`	Matches a digit
`\D`	Matches a non-digit
`\w`	Matches an alphanumberic character
`\W`	Matches a non-alphanumberic character
`\s`	Matches a whitespace character
`\S`	Matches a non-whitespace character
`\`	The backslash escapes special characters. So, if you want to match an actual, literal dot (".") you would use `\.`
`^`	Matches the start of a string
`$`	Matches the end of a string

These can be used in combination with a range of quantifiers:

`*`	Matches the preceding element 0 or more times
`+`	Matches the preceding element 1 or more times
`?`	Matches the preceding element 0 or 1 times
`{x}`	Matches the preceding element x times
`{x,y}`	Matches the preceding element between x and y times
`{x,}`	Matches the preceding element at least x times
`{,y}`	Matches the preceding element between 0 and y times

To see how this works, type the following into the "regular expressions" box: `\w+\.$`
It should match the last word of a string that ends in a full stop, ie. `continent.`
Note that the full stop after the word is also included, as it is part of the regular expression.

To explain:

- The `\w` element specifies that we are looking for alphanumeric characters.
- The `+` quantifier specifies that we want the `\w` character to be repeated 1 or more times, but we don't want to include any whitespace (which is not an alphanumeric character).
- The `\.` element specifies that we are looking for a full stop after the repetition of the alphanumeric characters. The backslash before the `.` is there so the `.` doesn't follow its

special syntactic function of matching any character.
- The `$` element specifies that we want to look for this sequence only at the end of an input string.

Important "gotcha": without a specific flag passed to the engine, the `^`, `$` and other operators apply *within a line*. That is, in a multi-line string, `$` would match the end of the line, not the end of the string. This may be specific to the engine that you are using, but is worth checking if this example does not work as you expect.

5. Greediness

Regular expressions are "greedy" by default. That is, they try to match as much as possible. This can yield unexpected results for those new to regular expressions.

Copy and paste the following into the test string box:

```
<a href="https://www.martineve.com/">Some text</a> <p>Some more text</p>
```

(Note that you shouldn't really match HTML using regular expressions but should instead use a markup parsing engine. This is just used as an example.)

Let us say that you wanted to match each of the HTML tags, plus the text within the tags, in the above block of text. Those who are new to regular expressions might attempt to match this with the following:

```
<.+\/.+>
```

To explain the logic (which doesn't work):

`<` means match `<` literally
`.+` means match any character one or more times
`\/` means match `/` literally, with the `\` telling the engine to ignore the special syntactic function of `/`
`.+` means match any character one or more times
`>` means match `>` literally

What will actually happen in this case is that the expression will match the *entire string*. That is, the match will be

Some text Some more text

because the expression matches everything between the opening "a" tag and the closing "p" tag. In order to solve this problem, we have to make our quantifiers *lazy*. That is, we need to instruct the regex engine to *stop looking* after the first occurrence of the matched elements. We do this by appending a `?` after the above quantifiers.

A solution that *would* work in this case would be `<.+?>.*?<\/.+?>`

To explain:

`<` means match `<` literally

`.+?` means lazily match any character one or more times until finding the next literal `>`
`>` means match `>` literally
`.*?` means lazily match any text zero or more times, up until finding the next literal `<`
`<` means match `<` literally
`/` means match `/` literally
`.+?` means lazily match any character one or more times until finding the next literal `>`
`>` means match `>` literally

This regular expression therefore returns two matches, separately:

Some text

and

Some more text

6. Character groups

One of the most useful things that you can do with a regular expression is to specify different *character groups* or *classes*. The premise behind this is that you may often want to match on a range of confined inputs within a limited space.

Copy and paste the following into the test string box:

```
Immanentize the eschaton, or immanentise the eschaton.
```

Let us say that, in this contrived example, we want to match for verb endings spelled in both the British English and American English ways. In other words, we want to extract words that end in -ize or -ise. One way of achieving this would be to use the regex `\w+i[sz]e` (so try putting this into the appropriate box).

To explain:

- the `\w` element specifies that we are looking for alphanumeric characters
- the `+` quantifier specifies that we are looking for the `\w` character to be repeated 1 or more times, but not to include any whitespace (which would not be an alphanumeric character)
- the literal `i` specifies that we are looking for the character "i"
- the *character group* block `[sz]` specifies that we are looking for *either* an "s" or a "z"
- the literal `e` specifies that we are looking for the character "e"

Let's look more closely at the idea of a character group block. The syntax of character groups uses square bracket notation:

`[`	Begins a character group
`]`	Closes a character group
Any character except `^-]\`	When inside a character group, will literally match the character
`\` followed by `^-]\`	Literally matches the character after the backslash
`^`	When immediately following the `[` character, `^` specifies a *negation* of what is inside the character group. Ie. the regex

	will match if the character in the string to be searched is *not* inside this character group
`-` between two characters	Functions as a character *range*. For instance, `a-z` means every character between a and z in the alphabet, in lowercase

Some of the syntactic elements we saw in step 5 are shortcuts for character groups. For instance, the digit matching element `\d` is a shortcut for `[0-9]`, ie. any digit between zero and nine.

To familiarise yourself with character groups, copy and paste the following placeholder text into the test string box:

```
Lorem ipsum \dolor\ sit amet, ^consectetur adipiscing elit?
Maecenas tempor eros sit amet [ligula] volutpat, a semper eros
dig-nissim.
```

Now try searching for the following regular expressions to see what they match:

```
[?]
[a-g]
[^a-g]
[\^]
[\\]
[\]]
[[\]]
```

7. Capturing matches

Many regular expression engines have the ability to "capture" different matches, so that you can extract the matched text so as to do things with it afterwards. It is also possible to "name" these groups. Groups are created using round brackets surrounding the group to capture. A name is applied by using this syntax:

```
(?P<group_name>.+)
```

In this example, the group called "group_name" will match any character one or more times (with `.` signifying any character other than whitespace, and the quantifier `+` signalling that there can be one or more repetitions of this).

Return to the HTML example we used in point 5, and paste the following in as a test string:

```
<a href="https://www.martineve.com/">Some text</a> <p>Some more text</p>
```

Now type in this regular expression:

```
<.+?>(?P<contents>.*?)<\/.+?>
```

In this case, the expression that results is the same as the one in point 5 except that, within the regex engine, there is also a captured group called "contents" that contains "Some text" in the first match and "Some more text" in the second. (Some sites, eg. regex101.com, will highlight the captured groups in one colour, and will highlight in another colour the other parts of the regular

expression that are being recognised as a regular expression, but not being captured.)

8. Backreferences

Consider a situation in which we want to match text to an earlier part of a string. Type the following into the test string box:

```
Is this equal to that? Is this equal to this?
```

We want to match the second phrase (Is this equal to this?) but not the first (Is this equal to that?).

The way that we can achieve this is by using a regular expression which employs backreferences:

```
Is (\w+) equal to \1\?
```

Every group that is captured within a regular expression can be referenced later within the expression. To explain:

- `Is` will match "Is " literally.
- `(\w+)` means match an alphanumeric character one or more times and capture it in a group, which as it's the first group captured is designated group 1
- `equal to` will match " equal to " literally.
- `\1` means match the same text as was captured in *group 1*
- `\?` means match a question mark literally (as it is a special character that requires escaping with a backslash so that it does not carry its special meaning but rather its literal meaning)

9. Lookahead and lookbehind

From time to time you will want to match a set of characters that either precede or succeed another set of characters, without including the preceding or succeeding characters in the match. Consider the following contrived example (type it into the test string box):

```
In the year 210AD there were 394 chickens roaming the plains.
```

One of the three numbers (210) is a date while the other (394) is not. Lookahead and lookbehind are the ways in which this problem of date extraction could be solved.

`(?=zzz)`	Lookahead. True if the next part of the string is "zzz".
`(?<=zzz)`	Lookbehind. True if the preceding part of the string is "zzz".
`(?!zzz)`	Negative lookahead. True if the next part of the string is *not* "zzz".
`(?<!zzz)`	Negative lookbehind. True if the preceding part of the string is *not* "zzz".

So, as a practical example, to extract the date in the above example we could use the regex `\d+(?=AD)`

To explain:

`\d+` will matches a digit one or more times
`(?=AD)` means: only match the preceding item `(\d+)` if it is followed by the literal `AD` (lookahead)

Important: some regex engines restrict the syntax allowed in lookbehind since it can become very processor intensive, and regex is supposed to be fast. Check the engine you are using for any such limitation.

10. More advanced regular expressions

You should now be able to match and extract basic regular expressions. There are several more advanced features of regular expressions, including recursion, subroutines, comments and free-spacing mode, which are beyond the remit of this introductory tutorial. For additional practice with regular expressions, see the "Understanding Regular Expressions" tutorial on the Programming Historian website at http://programminghistorian.org.

To suggest changes, or discover more tutorials,
visit http://blackbox.llc.ed.ac.uk