Regular expressions: the basic idea
Paul Gowder
Tutorial for R.A.s
8-24-15

Regular expressions ("regex" for short) are essentially a powerful way of doing search in a text. When you hit control-f in a word processor, and then you type something in, the word processor typically searches for the exact text you enter (maybe case sensitive, maybe not). But with regular expressions, you can search for patterns in a chunk of text, and match everything in that pattern. As the name "regular expressions" suggests, using them is a way of capitalizing on the regularity in the way that different kinds of meaning are expressed in text. Lots of time we're trying to search text for information, where we know the form in which that information is expressed, but we don't know the particular substance of the information.

Classic example: we know there's a phone number buried somewhere in this 100,000 word document, but we don't know *what* the phone number is (that's why we're searching for it). Wouldn't it be nice to be able to search for "three digits, then a dash, then three more digits, then a dash, then four digits?" With regular expressions, we can. (And with more complex regular expressions, we can also make the same search catch phone numbers delimited with parentheses and spaces, or just run together in a block, or whatever.)

You can see already how this might be helpful for legal practice. Think of being able to go through the text of discovery documents and find anything that looks like a phone number or an e-mail address, or through a really long case (or collection of cases) and find anything that looks like a statute or regulation.

So we're going to learn them, and then we're going to use them to extract statute/rule references from the case dataset we're creating.

First, let me give you a real-world, if complicated, example of how regexes work. The code here: https://github.com/paultopia/stray-cites is a tool that I wrote for myself, in order to fix the citations in a book that I have coming out soon. It had hundreds of citations, and I wanted to make sure that the things in the footnotes matched the things in the references list, e.g., that there wasn't anything missing from either, or years wrong, or the like (this is the sort of stuff that is really hard to do by hand, because humans don't concentrate nearly so well as computers, and don't have anything remotely like the same memory capacity, so you get errors). It relied heavily on regular expressions to match citation forms.

Here's (a slightly simplified version of) one of the regular expressions I used:

```
(^[A-Z][A-Za-z]*-?[A-Za-z]*),.*( \(?\d\d\d\d[a-z]?[.)])
```

That looks horrible, right? But it's actually really simple. Forget the parentheses for now, I'll explain them in a moment.

The point of this expression is to leverage the fact that all citations have a name, then a bunch of stuff, then a year, and that the year tends to be either surrounded by parentheses or ends in a period.

The first character, ^, means "look at the start of a line." Because references tend to be separated by line breaks, this is a good place to start.

The next block, [A-Z], means "any capital letter." The brackets are "pick any of these," and then the stuff in the middle gives the range, from capital A to capital Z. This is meant to capture the first letter of a name.

The next block, [A-Za-z]*-?[A-Za-z]*, is a bit more complicated but is meant to capture the rest of a last name. So the first set of brackets means "capture any upper case or lower-case letter" (A-Z or a-z). Then the asterisk means "keep hoovering this in until you run out," that is, capture any number of items in that set, from 0 to infinity. Some last names are hyphenated, so the hyphen in the middle looks for that, and then we have another block of as many upper or lower case letters as it finds in a row.

So far, our regular expression will just capture all names, hyphenated or no. But that doesn't narrow things down much. The next block, ,.*, says two things. First, the comma just means "capture a comma." It helps narrow things down, because citations (in the format I was using) have a comma after the last name. Which is exactly what this captures. The period and the asterisk are special. The period means "capture *anything*." And the asterisk, again, means capture anything as many times as it appears.

That does seem rather to make it worse, doesn't it? Now I'm not just capturing a name, I'm capturing any string of a name, plus a comma, plus *anything else of unlimited length* (except, and this is important, line breaks), including, for example, a million random digits. So this would capture the string "Gowder,ith3tih987gh309g7fhdkg8g8fho348fh493fh348fg3q847gf4ghpfhqffuhoui3g4huog49@#@@#^45788;;;uhq", which doesn't look like what I want. Fortunately, the next bit narrows things down even more:

The next block leads with \(?. Here's what this does. The backslash is called an "escape character." Escape characters, in programming, are characters that you put in front of special characters to mean "don't treat the character that follows like a special character, instead, treat it like what it is." So remember when I told you to ignore the parentheses? That's because they have a special function in regular expressions, which I'll explain later. But because *this* parenthesis is preceded by a backslash, it's not functioning as a special character, instead, it just means "match an

open parenthesis." And the question mark means "match it if it's there, but it's ok if it's not there too," or, in other words, "match exactly 0 or 1 of these."

Then we have an easy block: \d\d\d\d[a-z]?. The \d is a special signal that means "match any number" (I could have also used [0-9] to mean the same thing). Four of them means "match any number four times in a row." And the [a-z] at the end is included to capture the fact that sometimes citations have a letter appended at the end. (For example, if Smith published two things in 1990, and one is citing them both, one cites them as Smith 1990a and Smith 1990b.) The question mark at the end once again signifies that the letter is optional.

Finally, the [.)] block says "match either a period or a close parenthesis. You'll notice that there's no backslash in front of a parenthesis here. That's because you don't need to escape special characters in regular expressions if they appear in a bracketed list of options. (Otherwise I'd have had to escape the period too.) That isn't necessarily true in all implementations of regular expressions (every language does it a little differently), but it's true in the python implementation I was using.

So to sum up, what does this do? It matches the following pattern:
A new line
Followed by a word that starts with a capital letter, and might be hyphenated
Followed by a comma
Followed by any arbitrary text, so long as that before we get to the end of the line (before we get a line break) that arbitrary text ends with:
Four numerical digits, possibly preceded with an open and followed by a close parenthesis, and if not followed by a close parenthesis then followed with a period.

This is obviously a match to any citation format that leads with a name and comma (except those rare cases of uncapitalized names, like bell hooks and e.e. cummings), has a year in there somewhere, and is surrounded by line breaks. Which is exactly what I want, because that describes an ordinary list of references.

Ok, what about the parentheses that I mentioned? Well, so far, the regular expression will just search for the whole string of text that matches it, and it will give you back that string to work with. But often times, you only want part of the text. For a trivial legal example, suppose you're searching for the string "I find the defendant [guilty/not guilty]." If you write a regex to search for that, you'll get the whole sentence back, when really what you want is just the information "guilty" or "not guilty." If you use parenthesis, however, you can create "match groups" and then ask the underlying programming language to just give you back the stuff in those groups.

Go take a look at this example in action now, at this link:

https://regex101.com/r/yD8uQ8/1

You'll see that it captures citations (except for a couple of imperfections) and doesn't capture text. And it pulls out "match groups" for the author and the year---exactly the information I want. With a little bit more coding, this allows me to take any arbitrary document or collection of documents with the citations in this format, feed it to a python script, and get out a full author-date list of all the references that appear in it, while discarding everything else in the middle (like book titles, first names, page numbers, etc.).

Incidentally, parentheses can also be used to apply modifiers like * and ? to entire groups. So if I were actually trying to write a regular expression to match "I find the defendant guilty" and I find the defendant not guilty," I'd use:

I find the defendant (not)?\s?(guilty)\.

That expression finds the sentence in question, whether or not "not" appears. It will then give the code that uses it back one result ("guilty") where the verdict is guilty, and two results ("not" and "guilty") where the verdict is not guilty (the \s? just handles the extra space you get when there's a "not" -- \s means "match whitespace"). Pretty useful. (Unfortunately, the implementation of regex in python we'll be using hits a wall when you try to do really fancy stuff with nested parentheses and brackets and such, but we won't need to do that here. There are more powerful new implementations of regex for python, and also languages like perl that are just better at this stuff, if you ever find yourself truly in need of some serious pattern-matching.)

Fortunately, when we use regex in this project, it'll be a lot more like the simple "find guilty and not guilty verdicts" and a lot less like the complicated "find everything that's shaped like a citation."

Now that you understand the basics of how regular expressions work, I'd like you to go through the exercises on http://regexone.com/ -- when you finish those, you should be well equipped to carry out the matching tasks that I have for you. And you should also be well equipped to apply these skills to scare your colleagues when you're out in the practice of law and you need to do things like quickly find bits of text in documents…