Notebook

February 7, 2019

1 Regular Expressions

Regular expressions (or "regex"/"regexes") are one of the most powerful programming tools for lawyers. Essentially, regular expressions are a powerful specialized programming language built into other languages like Python, which allow you to express complicated text searching operations.

The utility of this for lawyers should be obvious: lawyers have to deal with lots and lots and lots of documents, and sometimes need to search through those documents for specific information. If those documents are in electronic form, regular expressions can provide you with a much more powerful way of searching than what is built into ordinary applications.

Here's a simple example. Suppose you've got a bunch of files. And suppose you're trying to find any document that contains a reference to specific dollar amounts—invoices, e-mails with discussions of financial matters, etc. One of the things you might do is search for anything in the usual format for representing dollars, i.e., a number with two digits after the decimal point, and, say, 1-8 digits before the decimal, maybe with commas in there. (Let's suppose that you don't expect any bigger amounts of money to be at issue.)

First, let's create some pretend data. In a real task, we'd probably have these files on disk somewhere, or in a database, and we'd open every file, search it, and save the document if it matched. Also, for this technique to be useful in a real task, we'd usually have hundreds or thousands of dollars. For the purposes of demonstration, however, we can put a handful of fake texts in memory.

```
In [1]: no_money_letter = """
        Hey Boss:

        Just so you know, I did all the crimes.  Like 100 of them. So many crimes.

        We should probably delete this e-mail before we get sued.

        xoxoxo,

        Lackey.

        """

        invoice = """
```

```
        Crime, INC.

        For services rendered:

        Tax Fraud            100.00

        Price-fixing           9.99

        Insider trading       20.00

        TOTAL                129.99

        """

        money_letter = """

        Dear Boss:

        Remember all those crimes I've committed?  They made the firm $1,280,012.05.  Maybe a ra

        xoxoxo,

        Lackey

        """

        docs = [money_letter, no_money_letter, invoice]
```

Now let's search this! I'll show you some code, and then, per usual, we'll walk through it.

```
In [2]: import re

        pattern = r'\d*,?\d*,?\d{0,3}\.\d\d'

        matches = []

        for doc in docs:
            if re.search(pattern, doc):
                matches.append(doc)
```

```
In [3]: for match in matches:
            print(match)
```

```
Dear Boss:

Remember all those crimes I've committed?  They made the firm $1,280,012.05.  Maybe a raise?
```

```
xoxoxo,

Lackey



Crime, INC.

For services rendered:

Tax Fraud            100.00

Price-fixing           9.99

Insider trading       20.00

TOTAL                129.99
```

You see what we did there? We used one search to match both the dollar amount `$1,280,012.05` and dollar amounts like `100.00` and `9.99` while *not* matching `100` in the first example letter, which doesn't look like a dollar amount. Try doing that with Google or Control-F (I think old versions of MS Word used to have something resembling regex capacity, but I can't find that in any online documentation anymore. Sorry law review editors, but you might try saving into plain text formats in order to try tasks like searching for citations...)

Let's look at the code.

```
import re

pattern = r'\d*,?\d*,?\d{0,3}\.\d\d'

matches = []

for doc in docs:
    if re.search(pattern, doc):
        matches.append(doc)
```

Most of that code is pretty self-explanatory, but the one part that's new is the line where we define the pattern and has this wild `r'\d*,?\d*,?\d{0,3}\.\d\d'` thing that sort of looks like Klingon. Unsurprisingly, that's a regular expression pattern.

One very useful way to build regular expressions is to use an application that lets you enter in sample text and try out different regular expressions to see what they match. My favorite is the free webapp Regex101. The neat thing about Regex101 is that it lets you save your examples and share them (using the little menu hamburger icon thingey on the left). So I've created a saved version of this search with our sample text that shows exactly how it works. Check it out here!

But before we get to our complicated pattern, let's start simpler. Regular expressions are basically just amped up search strings, and we know how to search, right? We can use ordinary strings as regexes as well.

First, let's abstract out our search into a function, then let's look at some simple examples.

```
In [4]: def search_docs(pattern, doclist):
            matches = []
            for doc in doclist:
                if re.search(pattern, doc):
                    matches.append(doc)
            return matches

In [5]: wise_statements = ["Dogs are ok, I guess",
                           "Cats are better than dogs",
                           "Professor Gowder has the best cat.",
                          "If you have a cat, you are cool."]

        search_docs(r"cat", wise_statements)

Out[5]: ['Professor Gowder has the best cat.', 'If you have a cat, you are cool.']
```

Ok, the first thing you should notice is that the regex pattern is a normal string, but with a lowercase r prepended to it. The reason for the lowercase r is that it tells Python to ignore special characters in the string—instead of processing those characters in Python-ey ways, it'll process them in regex-ey ways. You can just treat this as a rule: put a lowercase r before the string when you're making a regex pattern, and if you want the nitty-gritty technical details of why, look here.

The second thing you should notice is that this successfully searched for the wise statements with "cat" in lowercase, but not capitalized: regex is case sensitive. How to make it case-insensitive?

```
In [6]: search_docs(r'(?i)cat', wise_statements)

Out[6]: ['Cats are better than dogs',
         'Professor Gowder has the best cat.',
         'If you have a cat, you are cool.']
```

What we've done there is insert a special character sequence before the c, that said what we have to do afterward. Specifically, we've inserted a special character that means "match in either case." There are other ways we could have done this. For example:

```
In [7]: search_docs(r'[cC]at', wise_statements)

Out[7]: ['Cats are better than dogs',
         'Professor Gowder has the best cat.',
         'If you have a cat, you are cool.']
```

By enclosing characters in brackets, we tell regex "match any of these characters" so that regex means "match either `C` or `c`, followed by `at`" It achieves exactly the same thing as the previous one.

But there's a problem...

```
In [8]: wise_statements.append("When looking at data, it's always a good idea to start with a sc

        search_docs(r'[cC]at', wise_statements)

Out[8]: ['Cats are better than dogs',
         'Professor Gowder has the best cat.',
         'If you have a cat, you are cool.',
         "When looking at data, it's always a good idea to start with a scatterplot"]
```

What happened?? Well, the word "scatterplot" has the three letter sequence "cat" in it, so it matched. Ooops. How to fix it?

```
In [9]: search_docs(r'\b[cC]at\b', wise_statements)

Out[9]: ['Professor Gowder has the best cat.', 'If you have a cat, you are cool.']
```

The symbol says "there has to be a word boundary here" Basically, a word boundary means "not a letter or number," and the end of the string counts. So this pattern now says "Match the word cat, capitalized or not, and not adjacent to any other letters or numbers." This rules out "scatterplot."

Unfortunately, it also rules out "Cats"! Because that has an s after it! That's not what we wanted. Time to make it a bit more complicated...

```
In [10]: search_docs(r'\b[cC]ats?\b', wise_statements)

Out[10]: ['Cats are better than dogs',
          'Professor Gowder has the best cat.',
          'If you have a cat, you are cool.']
```

The question mark means "either zero or one of this character." So now our pattern says "match the word cat or cats, not embedded in any other words, and capitalized or not."

Now that you have some idea of how regular expressions work—by mixing up special characters and normal characters to describe a pattern for a string—we can understand the one we started off with.

```
r'\d*,?\d*,?\d{0,3}\.\d\d'
```

Here's a piece-by-piece translation of that.

- `\d` any digit character (0, 1, 2...9)
- `*` apply the preceding character any number of times, including zero so `\d*` means "any number of digits
- `,?` maybe a comma
- then we have any number of digits and maybe a comma again. This is a little imprecise, our regex would match weird stuff like `1,000000,000.00` but I don't really expect to see such a thing in a search of something like litigation documents.

- `\d{0,3}` This means "match any digit between 0 and 3 times." Again, it is a little imprecise. Our regex would match weird things like `.00` on its own. But again, I wouldn't expect to see such a thing in most realistic cases.
- `\.` This just means "match a period." The problem is that a period is a special character in regex language (it means "any character"), so we need to *escape* it with a backslash to tell the regex engine that we don't mean the special character, we mean the literal period.
- `\d\d` two digits.

Put together, our regex says "match any number of digits, maybe followed by a comma, followed by any number of digits, maybe followed by a comma, followed by 0-3 digits, followed by a period and two digits." Which is a pretty good stab at something that looks like a dollar amount.

now here's an exercise for you: You see those places up above where I said this regex is imprecise? Try and find a regex that gets rid of the imprecision by not accepting funky numbers like `1,000000,000.00` and `.00` on its own.

To help you out, I've put the imprecise regex and our bad examples into a regex101 sample. Try playing with it until you get it working.

...
...
...
...
...
...
...
...
...
...
...

If you want to see my solution, which also gets rid of the arbitrary limitation of only working with small dollar amounts, take a look at this regex101... but **try on your own first.** You don't get good at this stuff without practicing, and hunting down the information you don't know in order to solve problems. (Incidentally, I'm deliberately making you hunt down information you don't know to fill in gaps to solve the problem sets too. This is how problem sets work, actually—they're more student-search-driven instructional tools than testing tools.)

Regular expressions are a special language, built into but not part of the Python language—just about every other programming language also supports them, usually with identical or near-identical syntax (although some languages also give you special super-powerful syntax—the Perl programming language is particularly famous for its over-the-top regular expression capacity). And they're a great superpower!

If you want to learn more about them, I highly recommend the Regexone tutorial. One of the questions in our first problem set will probably require regular expressions (you could do it without, but it would be much harder), and going through that tutorial may help you. I also recommend the official Python regex howto

For more comprehensive but less readable documentation, regular-expressions.info has a ton of stuff. And always try out your regular expressions in something like regex101!