<oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum (djbpitt@gmail.com)
Last modified: 2018-01-31T00:45:33+0000

Regex assignment #1

What to submit

We describe below how to use regex to transform plain text into XML, which is the task for this assignment. We don't need to see your XML, but we do need to see a step-by-step explanation of how you used regex to create it. *That explanation should be a plain-text document (not a word-processor document);* the reason is that a word processor might convert your straight quotation marks to curly ones or make other typographic changes that are desirable when you're writing a term paper, but that will corrupt your regex (curly quotation marks are not legal replacements for straight ones in regex).

You can create your plain text explanation of your regex conversion process in <oXygen/>. Create a new document in <oXygen/>, selecting "Text" as the document type. Write up your process there, hitting the "Enter" or "Return" key at the end of each line to keep the line length manageable (<oXygen/> does not wrap long lines automatically, and pretty-printing doesn't work with plain text documents). Save your document with a ".txt" filename extention, which is the conventional extension for plain text documents.

The input text

The text we'll be using as input for the first regex homework assignment is a plain-text version of Shakespeare's sonnets, which you should download from http://www.gutenberg.org/cache/epub/1041/pg1041.txt and open in <oXygen/>. (Note that this site sometimes shows you a pop-up welcome screen and a list of different versions of the file, instead of taking you to the plain text one directly. If that happens, click OK on the Welcome pop-up and then select the version labeled Plain Text UTF-8.) You should manually delete any of the Project Gutenberg information from the beginning and end of this file, so that what you're left with is just the sonnets in order, with roman numerals before each one.

If you find it easier to work with a small amount of text, you can make yourself a document that contains just a few sonnets and use that during development. Once you think you're getting the results you want, you can then try applying the same strategy to the entire file. The structure of this document is very regular, so whatever works for a handful of sonnets should work for all of them.

The task

Your goal is to produce an XML version of this file by using the search-and-replace techniques we discussed in class. Your output should look something like http://dh.obdurodon.org/shakespeare-sonnets.xml. That is, each sonnet should be its own element, each line should be tagged separately, and the roman numerals should be encoded in a useful way (we've used attributes, but you could also put them in a child element).

How to proceed

There are several ways to get to the target output, but here is how we might approach the task:

Reserved characters

The plain text file could, at least in principle, contain characters that have special meaning in XML: the ampersand and the angle brackets. You need to search for those and replace them with their corresponding XML entities; if you don't remember the entity strings, you can look them up in the "Entities and numerical character references" section of http://dh.obdurodon.org/what-is-xml.xhtml. Note that you need to process them in the correct order. What is that order, and why is it important?

Title and author

The title and author at the top are going to have to be tagged manually. You can either remove them now and then paste them back in later, after you've tagged the sonnets, or you can leave them in place and fix them up at the end. You'll use global find-and-replace to tag the sonnets, and if you leave the title and author in place while you do that, you'll wind up tagging them incorrectly. That isn't a problem as long as you remember to fix them manually at the end.

To perform regex searching, you need to check the box labeled "Regular expression" at the bottom of the <oXygen/> find-and-replace dialog box, which you open with Control-f (Windows) or Command-f (Mac). If you don't check this box, <oXygen/> will just search for what you type literally, and it won't recognize that some characters in regex have special meaning. You don't have to check anything else yet. Be sure that "Dot matches all" is unchecked, though; we'll explain why below.

Leading space characters

The non-blank lines all begin with space characters: there are two spaces before most lines (the Roman numerals and the first twelve lines of each sonnet) and four spaces before the last two lines of every sonnet. Those spaces are presentational formatting, and not part of the content of the text, and since we don't need them in order to tag the text, we'll start by deleting them. The regex to match a space character is just a space character, and you can match one or more space characters by using the plus sign repetition indicator. To match one or more instances of the letter "X", you would use a regex like `X+`. To match one or more instances of a space character, just replace the "X" with a space.

You don't want to remove all space characters, though; you just want to remove the ones at the beginning of a line. You can do that by using the caret metacharacter, which anchors a match so that it succeeds only at the beginning of a line. For example, if the regex `X+` matches one or more instances of "X", the regex `^X+` matches one or more instances of "X" *only at the beginning of line*. You can use this information to match one or more space characters at the beginning of a line and replace them with nothing, that is, delete them.

We aren't going to use the blank lines in this approach, so you can delete those if you'd like, or you can leave them in place to enhance the legibility. To delete them, you need to match a blank line, and the easiest way to do that is to match two new line characters in a row and replace them with a single new line character. The regex for a new line character is `\n`. Try it.

Inside out or outside in

We can create our markup either from the outside in (document, then sonnet, then divide the sonnet into Roman numeral and lines) or from the inside out (lines and Roman numeral, then wrap those in a sonnet, then wrap all of the sonnets in a document). Either strategy can be made to work, but we generally find it easier to work from the inside out because when we work from outside in, it's easy to wind up incorrectly wrapping `<line>` tags around the `<sonnet>` start and end tags, etc.

Lines

We'll start by tagging every line as a `<line>`. This will erroneously tag the Roman numerals as if they were lines of poetry, which they aren't, but it's easier to let the first find-and-replace overgeneralize and then go back and retag the Roman numerals than to try to write a more constrained regex that won't overgeneralize. We don't want to tag blank lines (if we left them in), though, so we need a regex that matches only lines that have characters in them. Remember where we told you above to make sure that "Dot matches all" was unchecked? Normally the dot (`.`) matches any character except a new line, which means that we can use the plus sign repetition indicator to match one or more instances of any character except a new line (that is, `.+`). By default regex selects the longest possible match, so even though just two characters on a line will match the pattern, when we run it it will always match the entire line. Since the dot matches any character except a new line, the regex will match each line individually, that is, it won't run over a new line and continue the same match. Try it and examine the results. Now check "Dot matches all", run Find all, and look at those results. Notice that the match no longer stops at the end of the line, and since you want to tag each line individually, you need to uncheck that box to revert to the normal, default behavior.

A human might think of our task as "wrap every line in `<line>` tags", but regex has a find-and-replace view of the world, so a regex way to think about it would be "match every line, delete it, and replace it with itself wrapped in `<line>` tags". That is, regex doesn't think about leaving the line in place and inserting something before and after it; it thinks about matching the line, deleting it, and then putting it back, but with the addition of the desired tags. The regex selects and matches each full line, but how do we write what we selected into the replacement string? The answer is that the sequence `\0` in the replacement pattern means "the entire regex match", and you can use that to write the matched line back into the replacement, but wrapped in `<line>` tags. Try it.

Roman numerals

The Roman numerals are now erroneously tagged as if they were lines of poetry, and in our sample output at http://dh.obdurodon.org/shakespeare-sonnets.xml we want them to be attribute values. To start that process we need to think about how to distinguish a Roman numeral line from a real line of poetry. Since there are 154 sonnets, a Roman numeral line is a line that contains one or more instances of "I", "V", "X", "L", and "C" in any order and nothing else, and no real line of poetry matches that pattern. That means that we can match that pattern by using a regex *character class*, which you can read about at http://www.regular-expressions.info/charclass.html. This approach will match sequences that aren't Roman numerals, like "XVX", but those don't occur, so we don't have to worry about them. This illustrates a useful strategy: a simple regex that overgeneralizes vacuously may be more useful than a complex one that avoids matching things that won't occur anyway. You can use the character class (wrapped in square brackets) followed by a plus sign (meaning one or more) to complete your regex so that it matches only `<line>` elements that contain a Roman numeral and nothing but a Roman numeral. Try it.

In this case you want to write the Roman numeral into the replacement string, but you want to get rid of the spurious `<line>` tags and replace them with other markup. `\0` will write the entire match into the replacement, but that would include the original `<line>` tags that you want to remove. To capture just part of a regex match for reuse in the replacement, you wrap it in parentheses; this doesn't match parenthesis characters, but it does make the part of the regex that's between the parentheses available for reuse in the replacement string. For example, `a(b)c` would match the sequence "abc" and capture the "b" in the middle, so that it could be written into the replacement. Capturing a single literal character value isn't very useful because you could have just written the "b" into the replacement literally, but you can also capture wildcard matches. For example, `a(.)c` matches a sequence of a literal "a" character followed by any single character except a new line followed by a literal "c" character. You can use that type of approach to capture everything between the `<line>` tags in the matched string: write a regex that matches the entire line with the Roman numeral, including the `<line>` tags, but put parentheses around the stuff between the `<line>` tags.

Okay, you've captured the Roman numeral, but how do you write it into the replacement? To write a captured pattern into the replacement, use a backslash followed by a digit, where `\1` means the first captured group, `\2` means the second, etc. Since in this case we're capturing only one group (we have only one set of parentheses), wherever we write `\1` in our replacement string, we'll insert the Roman numeral that we captured. For this task we'd build a replacement string that starts with a `</sonnet>` end tag (since the Roman numeral appears after the end of the preceding sonnet), then a new line, and then a `<sonnet>` start tag, and inside that start tag we'd include the `number` attribute and use the captured string (that is, `\1`) as its value, etc. Try it.

Clean up

You may have to clean up the beginning and end of the document manually, including the title and author, and you'll also need to add a root element.

Checking your results

Although you've added XML markup to the document, <oXygen/> remembers that you opened it as plain text, which means that you can't check it for well-formedness. To fix that, save it as XML with File → Save as and give it the extension ".xml". Even that doesn't tell <oXygen/> that you've changed the file type, though; you have to close the file and reopen it. When you do that, <oXygen/> now knows that it's XML, so you can verify that it's well formed in the usual way: Control+Shift+W on Windows, Command+Shift+W on Mac, or click on the arrow next to the red check mark in the icon bar at the top and choose "Check well-formedness".

General

As we mention above, there are several ways to get to the target output, and whatever works is legitimate, as long as you make meaningful use of computational tools, including regular expressions (where appropriate), and don't just tag everything manually. As you saw in class, there are ways to build your own regular expressions to match whatever patterns you need to identify, and the regex languages is complex and often difficult to read. The way we would approach this task is by figuring out what we need to match and then looking up how to match it. In addition to the mini-tutorial above, there is a more comprehensive description in the regex section of Michael Kay's book and more detailed tutorial information at http://www.regular-expressions.info/tutorialcnt.html. If you decide to look around for alternative reference sites and find something that seems especially useful, please post the URL on the discussion boards, so that your classmates can also consult it.

What to submit (reminder)

We don’t need to see the XML that you produce as the output of your transformation because we’re going to recreate it ourselves anyway, but you do need to upload a step-by-step description of what you did (see the explanation at the top of the page of why and how to create a plain text write-up). Your write-up can be brief and concise, but it should provide enough information to enable us to duplicate the procedure you followed.

If you don’t get all the way to a solution, just upload the description of what you did, what the output looked like, and why you were not able to proceed any further. As always, you’re encouraged to post any questions on the discussion boards, in this case in the regex forum.