Regular Expressions (Regex) Activity

You've used GREL expressions in previous activities, so you know that GREL is a powerful tool for cleaning/editing your data. You can make GREL even more powerful by learning how to use regular expressions (aka regex). A regular expression is a sequence of characters that define a search pattern – it is used to search for matches within text. In OpenRefine, you can use it in your GREL expressions to create sophisticated patterns describing what type of information you want to find within your dataset, then do something with the matching text (edit it, delete it, put it in a new column, etc.).

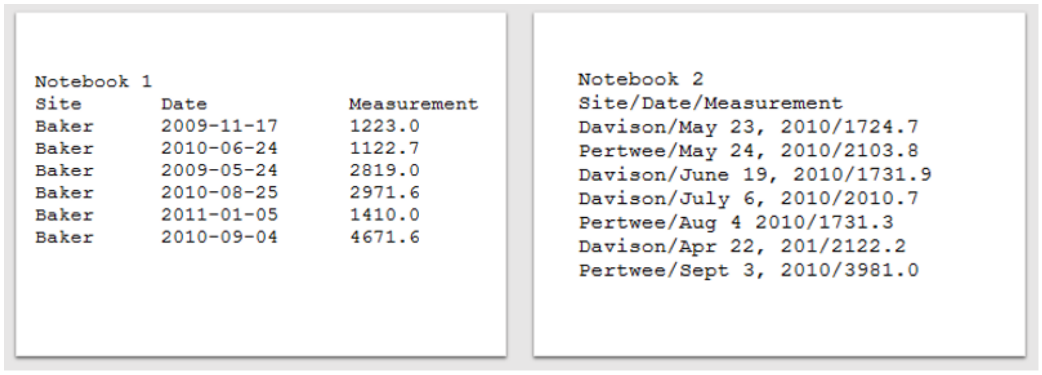
This activity assumes you have already completed the Survey of Household Spending and Citizen Science activities, have a familiarity with OpenRefine and know how to create simple GREL expressions.

In this activity, you are going to:

A. Open regex101.com and load some sample data
B. Practice some regex basics
C. Use regex in OpenRefine

A. Open regex101.com and load some sample data

1. **Browse to the website *regex101.com***. The *REGULAR EXPRESSION* box at the top of the screen is where you will type your regular expressions. The *TEST STRING* box below that needs to have some text that you want to match. Let's grab some sample data to use for matching.
2. Imagine you're helping a researcher who has collected some data, but it is a bit messy. The researcher had some grad students go out and collect data that consists of a site name, a date, and an instrument reading. However, they each formatted their file differently. For example, in Notebook 1, we have tabs separating columns of data, and the dates are written in international standard date format. Mostly this data file looks like it would be fairly decent to deal with. Notebook 2 on the other hand, has slashes as column separators, and the dates are in a non-standard format.



```
Notebook 1
Site        Date              Measurement
Baker       2009-11-17        1223.0
Baker       2010-06-24        1122.7
Baker       2009-05-24        2819.0
Baker       2010-08-25        2971.6
Baker       2011-01-05        1410.0
Baker       2010-09-04        4671.6
```

```
Notebook 2
Site/Date/Measurement
Davison/May 23, 2010/1724.7
Pertwee/May 24, 2010/2103.8
Davison/June 19, 2010/1731.9
Davison/July 6, 2010/2010.7
Pertwee/Aug 4 2010/1731.3
Davison/Apr 22, 201/2122.2
Pertwee/Sept 3, 2010/3981.0
```

3. **Open the file *sample_data.txt* in a text editing program such as Notepad**. This file contains the contents of Notebook 1 and Notebook 2. **Highlight the text and copy+paste it into the test string window in Regex101.** We are now ready to start writing regular expressions!

B. Practice some regex basics

4. Look at the regular expression box in Regex101. Notice that the regular expression starts with a slash and ends with a slash. You type the regular expression between the slashes. The simplest possible regular expression is a single character**. Type the letter *b* in the regular expression box.**
5. You can use the boxes at the right-hand side of the screen to gain information about what has happened. The *EXPLANATION* box will tell you what the regular expression you typed actually does, and the *MATCH INFORMATION* box will show you a list of all your matches.
   a. **If it looks like nothing happened when you typed the letter b:** check the explanation box – does it say the match was case sensitive? If yes, you need to **click on the *flag icon* at the far right end of the regular expression box and choose *insensitive* instead.** Now you should see matches.
   b. **If it looks like the letter b only matched one time, even though there are several instances of the letter b in this text: click on the flag icon at the far right end of the regular expression box, and choose *global*.** Now it should match all cases of the letter b in the text.

   The global and insensitive options are referred to as "mode modifiers". They go after the second slash in the regex syntax.
6. Let's start making some more interesting matches. In order to build regular expressions we use combinations of normal strings of text, along with certain characters that have special meaning, known as **metacharacters**. A list of all the metacharacters can be found here: http://www.rexegg.com/regex-quickstart.html. Some of the common ones that we will be using include backslash, pipe (vertical bar), parentheses, period, asterisk, and question mark.
7. Let's start with some matches on the first research assistant's data, which is easier to work with. (We'll need a different set of regular expressions to match the second assistant's data). First, find data from the month of May or the month of June. Use the **pipe** (vertical bar) metacharacter to represent OR. **In the regular expression box, type *`05|06`***. Have a look at the results in the Explanation and Match Information boxes.
8. Notice that we also made a match for January 5th which is not a month, so that was not what was intended here. You can take advantage of context to make a more precise match. We know the month has a hyphen on either side of it, whereas the day only has a hyphen at the beginning of it. **So to match months only, type *`-05-|-06-`***
9. It would be more efficient to write this in parentheses, i.e., **type *`-(05|06)-`* instead**. This way you don't have to repeat the same character more than once. The parentheses are also very handy because they create "groups", which are then displayed in the Match information box. **Have a look there right now – look at the difference between the *Full match* and the *Group 1* matches**. Groups allow you to pull out discrete pieces of your expression so that you can do things with them individually.

10. Now let's try to match entire dates. We can do this by taking advantage of context – we know the format of the dates is 4 digits, a hyphen, 2 digits, another hyphen, and then 2 more digits. We can use the **period** metacharacter as a stand-in for any single character in any particular position. **Type `....-..-..` to try this out**. Even more useful would be to add groups, so **change the expression to *`(....)-(..)-(..)`*** Now you will have 3 match groups, one for years, one for months, and one for days. We could use these to separate out the individual parts of the date in a tool like OpenRefine, if we wished to.
11. Now that we have successfully matched the easy dates, let's move on to research assistant 2's data. They're non-standard so this will be a bit harder. As a starting point, we know that the dates are surrounded by a slash on either side. **Remove the expression in the box right now and replace it with a slash character /.** You should see the error message "pattern error". This is because the slash is used in the syntax for writing expressions (i.e., it appears at the beginning and the end of every expression). If we want to match slashes, we need to "escape" them using the backslash metacharacter – this provides an instruction to treat the slash as a regular character rather than as a special character. **Type a backslash followed by a slash: \/** The same rule would apply to any metacharacter you want to actually match – you must first escape it.
12. Now we want to match all the characters between the two slashes. We don't know exactly how many characters there are, because the months and days have a variable number of characters in them. We can use the **asterisk *** metacharacter in combination with our period metacharacter, to match any character zero or more times. **Type `\/.*\/` and look at the results.** Let's also use grouping to get a match that doesn't actually include those slashes. **Type `\/(.*)\/` instead.**
13. Now let's break out each part of the date into a separate group, the way we did with the standardized dates. We need to take advantage of the context again, noting the space between the month and the day, and the comma+space between the day and the year. **Type `\/(.*) (.*), (.*)\/` and look at the results**.
14. This is mostly working, but not entirely. Notice there are some typos in the data – for example, one row is missing the comma. If we want to account for this, we need to match dates when they have a comma and when they don't. We can do this using the **question mark** metacharacter, which is used to make the preceding character optional. **Type `\/(.*) (.*),? (.*)\/` and look at the results.**
15. Now we've caught the date that is missing the comma, but we've introduced a new problem – group 2 now includes the commas when they exist. Making the comma optional means it is no longer a viable separator differentiating group 2 from group 3, and only the space is accomplishing this. We need another strategy. Using the period metacharacter is always a bit error prone, because it matches any character at all and so is not very precise. We could try making our match more specific, say, only allowing numbers. That will force the commas out. To do this, we can use the **square brackets** metacharacter, which can enclose a range indicating the specific kinds of characters we want to match. This is also known as "character sets". **Type `\/(.*) ([0-9]*),? (.*)\/` and look at the results**. Woohoo! You did it!

C. Use regex in OpenRefine

16. Now let's put this into practice to do something useful in OpenRefine. If you don't already have OpenRefine running on your computer, open it now. **Create a new project, and choose the file *sample_data.txt*. Click *Next*.**
17. In the preview view, notice that OpenRefine isn't able to parse the data into columns because there is no consistent column separator (due to the two notebooks being pasted into one file even though their formatting is different). We will have to use regex to divide it into columns ourselves. Notice also that line-based text files don't give you the option to treat the first row as column headers, nor would it make sense to do so right now anyway. **Check *Ignore first* and type *1 line(s)*** so that we don't get the column names mixed in with our data. **Give the project a name and then click *Create Project*.**
18. Let's use the regex we worked on in the last section to pull out the date information and put it into a new Date column. **From the *Column 1* pull down menu, select *Edit column -> Add column based on this column*.**
19. A GREL expression-builder window opens. We're going to use a GREL function called *partition* to break out the data into 3 partitions: 1. Everything before the dates (i.e., the site name), 2. The dates, and 3. Everything after the dates (i.e., the numeric measurements). **To start writing a partition statement, type *`value.partition()`***
20. Inside the parentheses is where GREL expects to find a regular expression. Let's start by putting the slashes to indicate the beginning and the end of a regex: ***`value.partition(//)`*** You can see that OpenRefine recognizes that this indicates a regex because some stuff appears in the preview column. But, it isn't what we want yet.
21. Next, include the expression we used earlier that matched the international standard formatted dates: ***`value.partition(/....-..-../)`*** Notice that you've got some results that seem like they could be useful now. The partition function is able to partition our data into a) everything before our pattern match, b) the pattern match itself, and c) everything after the pattern match.
22. This output is what is known as an *array.* An array is a collection of elements. It is notated by square brackets, with the different elements inside, separated by commas. Each element in the array can be referred to by a number (aka an index): the first element is 0, the second is 1 and so on. In order to create a column based on the dates, you need to tell OpenRefine to use the second element in the array – i.e., the element at index location 1. Do this by typing: ***`value.partition(/....-..-../)[1]`*** Ta-da!
23. However, scroll down until you can see some data from the second research assistant, with the less structured date formats. The match isn't working there. We need to combine the two regular expressions we developed earlier, one for each date format. **For now, remove the *[1]* so we can see all the elements of the match arrays**.
24. We need to make use of OR (remember this is the pipe or vertical bar symbol |). Start by putting parentheses around the entire existing expression, add the pipe symbol, and then a second set of parentheses with the second expression within them. The result looks pretty complicated, but should be familiar: ***`value.partition(/(....-..-..)|(\/(.*)([0-9]*),? (.*)\/)/)`***

25. Have a look at the results. It mostly works, except the slashes are being included with the date matches for the second expression. This is because the partition GREL function is not able to make use of regex groups the way we were able to in regex101.com. It only takes the entire match for the expression, so because we have included slashes in the expression, they match. (Side note: there's another GREL function, *match*, that can use regex groups, but it is much more difficult to work with for this example). So, we need to find a way to remove the slashes and have the match still work. There's a trick we can use here that we didn't talk about earlier; it's a special shortcut, *w*, that matches only "word" characters. If we replace the period in our first and last parentheses with a \w, then instead of matching any character at all, it will only match letters and number and will exclude slashes. Then we will no longer need to explicitly include the slashes in the expression. Type:
    ```
    value.partition(/(....-..-..)|((\w*) ([0-9]*),? (\w*))/)
    ```
26. The final touch is to add the [1] back to the end:
    ```
    value.partition(/(....-..-..)|((\w*) ([0-9]*),? (\w*))/)[1]
    ```
27. **Give the column the name *Date* and click *OK***.
28. The dates are now in their own column. They aren't all formatted the same way, and also note that OpenRefine is not recognizing them as dates yet (you can tell because they aren't shown in green). To transform them to dates, **from the Date column pull down menu, choose *Edit cells -> Common transformations -> To date***.
29. For a further challenge, try to perform these steps again to create a column for the site name (array position [0]) and the numeric measurements (array position [2]). In each case, you'll need to do some fiddling with the expression to keep the slashes out of the array element you're trying to parse at that time.

That's it for our Regular Expressions activity! You're now able to perform matching operations on your data using GREL and regular expressions.