Further Readings for Week 3

This week, we learned how to use regular expressions to match patterns of interest. We'll see a practical scenario to apply regex on a problem that I came across when working on a CSV file.

To summarize the task, I had to extract values from certain columns of a CSV file. Fairly simple and straightforward! This is a mock-up version of the CSV file I had -

```
$ cat data.csv
column1,column2,column3,column4,column5,column6,column7,column8
"12,875",45.2%,"32,941,053",100.0%,9.0,12,"80,828",43
```

I wanted values in columns 1 and 3. I mindlessly used `cut` to extract these values.

```
$ cut -f1,3 -d, data.csv
column1,column3
"12,45.2%
```

Only when the output did not look right, I realized that the numbers are also separated by commas for readability. The biggest challenge in retrieving the values is to differentiate the field separator commas from the digit group separators.

(I agree its much easier to handle this with a CSV-parsing function in other programming languages, but I wanted to experiment with regex and Linux commands to learn something new!).

There are two basic approaches we can use to manipulate the CSV file to extract values from columns of interest:

1. Remove all the commas inside the double quotes
2. Identify the field separator commas and replace them with another field separator.

Let's explore both these approaches.

Remove commas inside double quotes -

First, let's make a list of patterns that identify such commas-

- **Commas that are preceded and succeeded by a number** - This is too generic as it matches `9.0,12`(columns 5 and 6).
- **Commas that come before 3 digits** - This seems reasonable. We can write a pattern using positive lookahead(a lookaround is required here as we are trying to match the comma relative to its context without consuming it).

```
,(?=[0-9]{3})
```

This matches the comma that comes before 3 numbers. So to remove such commas, we can use substitution. An intuitive guess is to use `sed` but unfortunately, it does not support PCRE unlike `grep`. To

get around this, we can use `perl`.

Perl is another general-purpose programming language used for text file manipulation. Fun fact, there is no official full form of Perl, but some expansions commonly stated are "Practical Extraction and Reporting Language", "Pathologically Eclectic Rubbish Lister" and "Practically Everything Really Likable"!

We'll only look at how we can use `perl` in our case, without getting too deep into it.

```
$ perl -pe 's/,(?=[0-9]{3})//g' data.csv
column1,column2,column3,column4,column5,column6,column7,column8
"12875",45.2%,"32941053"100.0%,9.0,12,"80828",43
```

`-p` option tells perl to go over each line of the input file.

`-e` option enables executing one-liners like the above.

The result looks almost correct except that it fails at `"32941053"100.0%` (columns 3 and 4).

Can we extend this pattern and make it work? Prof. Collins says yes and suggested a solution using `sed` (and without lookarounds, but a similar concept of capturing context) just using what we have learned so far in class:

We can add more context to the pattern to avoid the above failure (`"32941053"100.0%`) by writing a pattern to match numbers on both sides of the comma -

```
([0-9]{1,3}),([0-9]{3})
------------ ----------
      1          2
```

The first part matches 1-3 numbers, followed by a comma and second part matches the 3 fixed numbers that come after comma(for example, 3,916 or 12,578 or 846,292). We can use this in `sed` command along with match groups to extract only the first and the second part and exclude the digit grouping comma:

```
$ sed -r -e 's/([0-9]{1,3}),([0-9]{3})/\1\2/g' data.csv
column1,column2,column3,column4,column5,column6,column7,column8
"12875",45.2%,"32941,053",100.0%,9.0,12,"80828",43
```

`\1` matches first group and `\2` matches second group. We can see that this has already removed some digit grouping commas like in column 1 and 7. So how do we extend this idea to remove the lingering comma in column 3? Use another round of substitution using the same pattern:

```
$ sed -r -e 's/([0-9]{1,3}),([0-9]{3})/\1\2/g' -e 's/([0-9]{1,3}),([0-9]
{3})/\1\2/g' data.csv
column1,column2,column3,column4,column5,column6,column7,column8
"12875",45.2%,"32941053",100.0%,9.0,12,"80828",43
```

There we go, all the digit grouping commas are removed! You may wonder how many rounds of such patterns we need to keep adding if we have a HUGE number. You don't have to add any more, just two works to capture any number. Lets illustrate this:

```
$ echo "28,374,532,567,139,235,134,083" | sed -r -e 's/([0-9]{1,3}),([0-9]
{3})/\1\2/g' -e 's/([0-9]{1,3}),([0-9]{3})/\1\2/g'
2837453256713923513408 3
```

We can see how this works if we break it down -

```
$ echo "28,374,532,567,139,235,134,083" | sed -r -e 's/([0-9]{1,3}),([0-9]
{3})/\1\2/g'
28374,532567,139235,134083
$ echo "28,374,532,567,139,235,134,083" | sed -r -e 's/([0-9]{1,3}),([0-9]
{3})/\1\2/g' -e 's/([0-9]{1,3}),([0-9]{3})/\1\2/g'
2837453256713923513408 3
```

- **Commas inside the double quotes** - We need to use lookarounds here as well to consume the opening quote, ending quote and numbers but not the commas. We can write a pattern like below -

```
(?<="[0-9]+),(?=[0-9]{3}")
------------- --------------
      1             2
```

The first part is a positive lookbehind which matches the opening quote and some numbers, followed by a comma that we want to consume and the second part is a positive lookahead which matches 3 numbers followed by a closing quote. There are two problems here:

1. There can be any number of commas depending on how big the number is (for example, "23,732" or "204,194,284"). The fact that we have double quotes in the positive lookbehind and positive lookahead kind of hardcodes and limits the numbers it matches.
2. The bigger problem is the pattern does not work! This is because the lookbehind requires a fixed width pattern and adding quantifiers like `*` or `+` make it a variable width pattern.

```
$ perl -pe 's/(?<="[0-9]+),(?=[0-9]{3}")//g' data.csv
Lookbehind longer than 255 not implemented in regex m/(?<="[0-9]+),(?=[0-9]
{3}")/ at -e line 1.
```

Something to keep in mind with this approach in general is that it may not always be appropriate to remove the digit grouping commas thereby changing the data we have. Here it might not matter much, but think if it was an address field or some text where removing commas could alter its meaning. In general, it is a good practice to leave the as data as is and only manipulate the non-data part which is what we'll see in the next approach.

Identify the field separator commas and replace them with another field separator

In this approach, we will replace the field separator comma with something else to extract all the elements that are not field separators and then join those elements together again with a different field separator. First, let us frame a pattern to do this. This is simple, we need to match everything within quotes(like column1, "12,875") and also everything that is not within quotes(like column2, 45.2%).

```
([^,]+)|("[^"]+")
------- ------------
   1         2
```

The first part `([^,]+)` matches everything except comma, `|` specifies OR condition and last part `("[^"]+")` matches everything within quotes including quotes.

Let us use this in `grep` and see the output it gives:

```
$ grep -Eo '([^,]+)|("[^"]+")' data.csv
column1
column2
column3
column4
column5
column6
column7
column8
"12,875"
45.2%
"32,941,053"
100.0%
9.0
12
"80,828"
43
```

We can now pipe this to `paste` to get tab-separated lines and then use `cut` as usual:

```
$ grep -Eo '([^,]+)|("[^"]+")' data.csv | paste - - - - - - - - | cut -f1,3
column1 column3
"12,875"        "32,941,053"
```

There we have it!

However, if we have hundreds of columns, it is cumbersome to type in `-`(STDIN) as input to `paste`. We can workaround this using loops(which we haven't covered yet) or a handy trick that Prof. Collins suggested: first replace the newline character with any character that is not a part of the file content, tab-delimit the `grep` output and finally replace the chosen character with a newline character. Let's go through an example to it understand better:

```
$ cat data.csv
column1,column2,column3,column4,column5,column6,column7,column8
```

```
"12,875",45.2%,"32,941,053",100.0%,9.0,12,"80,828",43

$ tr "\n" "@" <data.csv #Replace new line characters with @(for example)
column1,column2,column3,column4,column5,column6,column7,column8@"12,875",45.
2%,"32,941,053",100.0%,9.0,12,"80,828",43@

$ tr "\n" "@" <data.csv | grep -Eo '[^@,]+|"[^"]+"|@' #use grep as before
but also include @ in your pattern
column1
column2
column3
column4
column5
column6
column7
column8
@
"12,875"
45.2%
"32,941,053"
100.0%
9.0
12
"80,828"
43
@

$ tr "\n" "@" <data.csv | grep -Eo '[^@,]+|"[^"]+"|@' | tr "\n" "\t"
#convert newlines into tabs
column1 column2 column3 column4 column5 column6 column7 column8 @
"12,875"        45.2%   "32,941,053"    100.0%  9.0     12      "80,828"
43      @

#lets verify the tabs using cat -T
$ tr "\n" "@" <data.csv | grep -Eo '[^@,]+|"[^"]+"|@' | tr "\n" "\t" | cat -
T
column1^Icolumn2^Icolumn3^Icolumn4^Icolumn5^Icolumn6^Icolumn7^Icolumn8^I@^I"
12,875"^I45.2%^I"32,941,053"^I100.0%^I9.0^I12^I"80,828"^I43^I@^I

$ tr "\n" "@" <data.csv | grep -Eo '[^@,]+|"[^"]+"|@' | tr "\n" "\t" | sed
's/\t@\t/\n/g' #finally replace the @ and it's surrounding tabs with newline
character
column1 column2 column3 column4 column5 column6 column7 column8
"12,875"        45.2%   "32,941,053"    100.0%  9.0     12      "80,828"
43
```

You can come back next week and see if you can use loops to do this!

Last thing, an alternative to using `grep` and chaining it with other commands like we did above is to use `awk` like so:

```
$ awk -v FPAT='([^,]+)|("[^"]+")' '{print $1,$3}' data.csv
column1 column3
"12,875" "32,941,053"
```

`-v` sets the special variable `FPAT`, short for Field PATtern which can take any regular expression to match the fields rather than field separator. `awk` has features to handle CSV files, check out this section of awk's manual to read more on this.

It is also interesting to note that newer versions of `awk` have built-in capabilities to handle CSV files without using FPAT(similar to CSV parsing functions available in other programming languages). This Stackoverflow answer expands on this feature.

If you want to some practice, try writing regex to match anything that has a fixed pattern like:

- Zipcode
- Phone number
- Email address
- Postal address

You can verify your pattern using any regex tools like https://regexr.com/ mentioned in the class or my personal favorite, https://regex101.com/.