Unix and Computer Science Skills Tutorial
Workbook 5:
Regular Expressions and Places to Use Them

October 23, 2015

This workbook is a tutorial on how to use regular expressions and the tools on the *nix command that are powered by them. The first section of this workbook is a general introduction to regular expressions and the second section looks a the the two *nix commands `grep` and `find` which are very powerful tools for finding things and are both powered by regular expressions.

As with the other workbooks I recommend that you take the time to work through the examples to get a feeling for how things work. I have also included some questions and prompts, again, to prompt you to take a look at the tools and ideas. Sometimes the answers to the questions will be fairly direct from the workbook and some will require some extra research on your part. The prompts will encourage you to try things for yourself.

This workbooks assumes that you are comfortable doing basic tasks on the *nix command line including the material covered in Workbook 1 and Workbook 2.

1 Regular Expressions

Regular expressions (regexs) are a tool to let us match text. They allow us to test strings to see if they match a pattern we have created. Once a regular expression engine has been given a pattern and an input it steps through the input to see if the pattern matches. Regular expressions are used in a lot of applications where its necessary to validate text or to extract information from more complex text.

One important thing to note is that there are a lot of different implementations of Regular Expressions. Most of the things we'll look at in this workbook are fairly common. POSIX defines both a basic and extended set of regular expression terms which also provides common features that you will see in most places. Before you use a regular expression implementation, you should read over the documentation to see exactly how things will behave.

I put a simple regex tool (built in Java) in the course directory, if you would like to experiment with a simple tool. I will point out some more tools at the end of the section. You can run my tool with the command `java Regex`. Then enter a pattern and as many inputs as you like. (It's very simple so you'll have to close it using `ctrl-C`).

When using regular expressions we want to define a pattern of characters that a string can match. The pattern consists of a series of characters that the characters of the string need to match to match the regular expression. These characters can either be literal (for example the character must be 'f') or described by a class (for example the character must be a 0-9 digit).

For example is we start with the pattern `dog`. Then any string that contains the characters 'd', 'o', and 'g' in that order will match it.

```
Pattern:  dog
Input:  dog (matches)
Input:  I am a dog (matches)
Input:  A dog I am (matches)
Input:  I am a cat (does not match)
```

If we want to be able to check a bunch of strings to see if they contain the word dog, this is great. But often enough we want to do something more complicated. Say for example we wanted to match any time the word grey was in a string even if it had been written by an American (gray).

We can do this by replacing the vowel 'e' or 'a' with a class. We do that by including the characters we would like the class to include in square brackets. The class that we need in this case is `[ae]`. We can use it to replace the character where we're happy with either 'a' or 'e'.

```
Pattern:  gr[ae]y
Input:  gray (matches)
Input:  grey (matches)
Input:  griy (does not match)
Input:  dog (does not match)
```

Classes are great, but using `[abcdefghijklmnopqrstuvwxyz]` every time you want to match any alphabet character is not the most efficient approach. Fortunately there are some shorthand classes built in to make things easier for us. `\w` matches all the "word" characters (the alphabetic characters (upper and lower case), the digits 0-9 and the underscore _).

If we wanted to match three letters we could use the pattern `\w\w\w`

```
Pattern:  \w\w\w
Input:  abc (matches)
Input:  defg (matches)
Input:  ab (does not match)
Input:  dog (matches)
```

The two other shorthand classes that you'll frequently use are `\d` for digits 0-9 and `\s` for whitespace (including tabs). The dot (period) `.` matches all characters regardless of type. There are others, but you'll have to look them up as you need them.

You'll notice that our three word character pattern above matched a four character word. Since it contains three characters, it's a correct match, but often we want to make sure that we only match words of specific length. So if we would like to match `abc` and not `defg` we need to be able to find where the edge of words are. the shorthand `\b` finds word boundaries. So to fix our above problem we can use the pattern `\b\w\w\w\b`

```
Pattern:  \b\w\w\w\b
Input:  abc (matches)
Input:  defg (does not match)
Input:  ab (does not match)
Input:  dog (matches)
```

You can also use the special characters '^' and '$' to match the beginning and end of the line respectively.

```
Pattern:  ^\w\w\w\b
Input:  abc (matches)
Input:  defg (does not match)
Input:  defg abc (does not match)
Input:  dog (matches)
```

```
Pattern:  \b\w\w\w$
Input:  abc (matches)
Input:  defg (does not match)
Input:  defg abc (matches)
Input:  dog (matches)
```

If you need to use any of the special characters such at `$` or `.` you will have to "escape" them. You do that by putting a backslash \ in front of the character you need to escape.

We can repeat characters in our patterns, but what if we want to match three or four of the same character? What if we want to match any number of the same character? Fortunately there are some ways where we can say how many of a character a pattern should match. First we can include a number in braces {`#`}, after the character which tells it to match that character exactly that many times. If we want to include a range of characters we can include that range in the braces {`#,#`}.

```
Pattern:  do{3}g
Input:  dooog (matches)
Input:  dog (does not match)

Pattern:  do{2,4}g
Input:  doog (matches)
Input:  dooog (matches)
Input:  doooog (matcher)
Input:  dog (does not match)
```

There are also quantifiers for regular expressions. The first is `?` which lets you specify that there should be zero or one of the character. The second is `+` which lets you specify that there should be one ore more of the character. The third is `*` which lets you specify that there should be zero or more of the character.

```
Pattern:  colou?r
Input:  color (matches)
Input:  colour (matches)
Input:  dog (does not match)

Patern:  do+g
Input:  dog (matches)
Input:  doog (matches)
Input:  doooooooooooooog(matches)
Input:  dg (does not match)
Input:  dig (does not match)

Patern:  do*g
Input:  dog (matches)
Input:  doog (matches)
Input:  doooooooooooooog(matches)
Input:  dg (matches)
Input:  dig (does not match)
```

That covers the very basics of regular expressions. As with a lot of things you'll want to pick up more when you have a particular problem to face (for example when you want to extract an e-mail address from a string). The website `http://www.regular-expressions.info/` provides a good tutorial and reference when working with regular expressions and generally is aware of implementation specific irregularities.

As with many things practice is the best way to learn, so I recommend the interactive tutorial at `http://regexone.com/` as an interactive tutorial to practice your skills and to cover some of the material we skipped over here.

You will also find that people have already created regular expressions for a lot of common problems (such as verifying e-mail addresses, postal codes or phone numbers). It's a good way to save time, but you should always carefully test to make sure that any regular expression you take from someone else actually does what you think it does.

Finally it's good to have a tool to test regular expressions with. The Java matcher provided is a very basic tool, so you might want to look at the testers at `http://www.regexr.com/`, `http://www.regextester.com/` or `http://regexpal.com/`.

1.0.1 Questions

1. Complete the `http://regexone.com/` tutorial.

2 Regular Expressions on the *nix Command Line

`find` and `grep` are two of the most powerful tools available on the *nix command line. `grep` allows you to use regular expressions to find lines in files that match patterns. The second is find which allows you to find specific files and execute dynamic commands on the files it found.

2.1 `grep`

`grep` takes options, then a pattern and then files to search. `grep` searches through all of the files it's given and prints (to standard output) all of the lines that match the pattern.

```
$ grep .*oom short_file.txt
Did eyes preadamite first see the bloom,
Fairer than roses brightening life's gloom,
Emblazoned banners, and the booming gong;
```

The options for `grep` that you're likely to need the most often are `-n` which causes `grep` to print the line numbers. `-i` tells grep to ignore the case of characters in the pattern. `-v` tells grep to print all the lines that *don't* match the pattern (which can often be much easier to create).

Because `grep` is so powerful it's `manpage` is a little complex. I recommend starting by looking at some introductory tutorials such as the one at `http://www.uccs.edu/~ahitchco/grep/`. The tutorial at `http://www.grymoire.com/Unix/Grep.html` is also good.

`grep` has a myriad of uses. If you don't have access to an IDE to edit code, `grep` is your number one tool for finding usage and definitions in your code. When you're processing data `grep` is your friend for cleaning out bad entries or lines that you don't want to deal with.

> It's worth mentioning that most of the time on a modern system when you call `grep` you are actually calling a more modern `egrep`, which implements a more complete set of regular expression commands. This will almost never matter, but on rare occasions you might find yourself faced with and old or an odd system where grep is not quite what you expect.

2.1.1 Questions

2. Using the `text_files` from workbook 1 to answer the following questions:

3. How many lines include the word 'thy' in short_file.txt. (Don't forget piping and `wc`.

4. What is the first line in long_file.txt that mentions pizza.

2.2 `find`

`find` is probably the most complex *nix utility. It allows you to query the *nix system and find files that match specific criteria.

```
$ find .  -name '*.pdf' -print
./tealistSO.pdf
./icebreaker.pdf
./tealist.pdf
./TAiR_Report_Kendon_2015.pdf
./confrencereceipt.pdf
./signs.pdf
```

`find` is sufficiently complex that you'll mostly want to learn each function of find as you need. As with `grep` because `find` is so complex its `man page` can be hard to parse. Again I recommend finding tutorials to teach the parts you need as you need them. As a general tutorial `http://www.grymoire.com/Unix/Find.html` is a good introduction as is `https://danielmiessler.com/study/find/`. `http://alvinalexander.com/unix/edu/examples/find.shtml` provides a good resource for specific find commands.

2.2.1 Questions

5. Use `find` to find all the files you haven't accessed in the last month.

6. Use `find` to list the first five lines of every text file in your home directory (don't forget `head`).

7. Use `find` and `grep` to find all of the comments in all of your Python or Java files.