Summary

This is part of a longer, standalone, 3-hour workshop, It is currently incomplete here but since it is intended to fit only within 1/2 the time it is expected that it will be fine in its current state:

1. Brief (<10 min) overview of cases where regular expressions have been useful. This is meant to provide some context for their value.
2. Hands-on building of regular expressions. This will be the bulk of the workshop. Each concept introduced will conclude with a skill test that has participants reading and writing regular expressions.
3. (We won't get to this in the Software Carpentry Workshop) Looking at how regular expressions are deployed in three common environments: command line, Python, and R. This is about the basics of deployment, just enough to get people started since familiarity with these environments will not be assumed (registered participants will be polled in advance and this section tweaked as a result). 30 min

Overview of Examples

Cases to consider:

1. Address extraction for the CWRC library locations. http://cwrc.ca/rsc-src/. Massive time savings over having graduate students do this work "by hand".
2. Finding and returning words in context.
3. Easily fit within a programming/scripting environment.
4. Just simple regex for the Research Data Management Week tool. Every year I need to capture exactly the names of the workshops from a Google registration sheet. Finding and replacing within TextWrangler makes formatting this easy.
5. Filtering files on a computer with terminal. Show how files with a common naming convention that are distributed over a set of directories can be brought together in one place with the command line, or broken up into different locations. Add in the importance of working with text files and keeping your documents as text for processing. Example with a Word or Excel file that makes using RegEx hard? Also renaming files...

Setup

We'll use https://regex101.com/ for all the initial examples. Why?

- Everyone will be able to access it.

- It has built in help tools.
- It explains submitted RegEx in plain English (and does a reasonable job at this).

Our example text will be the nonsense poem Hunting of the Snark by Lewis Carol. Note that this will need the head and tail cut off to capture the "pure" poem.

#1 Sequences of Characters

Cut and paste the first piece of sample text into the TEXT STRING box. In the REGULAR EXPRESSION box we'll begin searching with the following string:

```
snark
```

As you are typing you'll notice that different parts of the sample text are highlighted in blue and then disappear. This is the regex engine parsing away as you enter the search expression. When complete you'll see the number of matches, steps, and time taken to carryout the search. In this case it should look something like `1 match, 2185 steps (~9ms)`. Despite all the steps the search is pretty fast, much faster than you or I scrolling through the text to find a text string.

Note the Explanation box to the right of the REGULAR EXPRESSION box. It will provide you with an English language explanation of what your expression is saying in regex syntax. It won't always be perfect but it can be quite helpful when trying to understand what is happening. In this case we are told that

> /snark/g
>
> **bodkin** matches the characters `bodkin` literally (case sensitive)
>
> Global pattern flags **g modifier**: global. All matches (don't return after first match)

This also tells us what the greyed out **g** is at the end of the REGULAR EXPRESSION box, it is the search globally flag (when on it will have the regex engine find *all* matches, not just the first). Clicking on the icon of the flag to the right of this g will allow other flags to be turned on or off, adding, or removing, letters from the end of this line. The syntax that is being applied here is what is used in command line tools that use regex, such as sed, the **S**tream **ED**itor.

The "snark" that we have currently captured is from the line:

> When a vessel is, so to speak, “snarked.”

This is not the "Snark" that we are looking for. We have two options at this point:

1. Turn on the case insensitive flag.

2. Put a capital S on our original "snark".

What are the consequences of either?

We can do searches like this for any word. Try it for `beaver` and `friend`. Note the differences with case insensitivity and using a capital letter at the start of each word.

> What happens when spaces are included in the regular expression?

#2 - Tokens and Quantifiers

So far the regular search box in almost any software that works with text, such as web browsers and word processors (CTRL-F on Windows/Linux and CMD-F on Macs will bring up this tool in most cases), can mostly do what we've seen so far. The real power of regular expressions comes with their ability to provide generalized search by specifying patterns of characters for the regex engine to match.

Let us prompt an exploration of the generalization capabilities of regex by finding *all* and *only* the words ending with `ing`.

We start by making the following our regular expression:

```
ing
```

This will return 101 matches and directly we should note that there are two problems with what is returned by looking at the match list:

1. Only the `ing` portion is actually returned, not the entire word that it is part of.
2. Cases where `ing` appears in a word at a place other than the very end are also returned. One such word causing trouble is "finger".

We can begin to deal with the first problem by noting that the character pattern `\w`, known as the word character token, will search for and return *any* word character. We can thus add `\w` when we know we need a word character but we are not sure which one.

> a "word character" is any alphabetical character.

> The backslash character `\` is used as an escape character by the regex engine, a character that says "do not treat the character that follows as you would normally". Here, the backslash is modifying the w, creating a new character sequence that the regular expression engine understands as the "any word character" character. Sequences of characters like this are known as "tokens".

[Need to do a better job of explaining tokens.]

Let's try it now:

```
\wing
\w\wing
\w\w\wing
```

Unfortunately this approach demands that we know exactly how many characters are before the suffix that we are after and this can often vary (e.g. here we have "thing" with two characters out front all the way to "recollecting" with nine characters out front). We need a way to generalize not only what characters are captured but also the number of times those characters appear. This can be done by adding a quantifier.

```
\w+ing
```

The `+` is modifying the character immediately in front of it, in this case the `\w` (recall that the `\` is bound to the `w` making `\w` a token that designates a single word character), and telling the regex engine to "match the previous character one or more times". You will see that this now solves the first problem and we are able to capture the entire word. We will now turn our attention to capturing *only* the words that *end* with "ing".

Recalling that spaces matter it might be tempting to add a space to the end of "\w+ing ":

```
\w+ing
```

This returns 81 matches. Looking through the poem though it will be clear that some matches are missing. For example:

- "But the principal failing occurred in the sailing," in *Fit the Second* is skipped because a "," is not a " "
- "There is Thingumbob shouting!” the Bellman said" in *Fit the Eighth* is skipped because "!" is not the same as a " "

We can make use of another token, in this case the word boundary token `\b`, to solve this problem.

\w+ing\b

This gives us 90 matches.

3 Further Quantification

At this point we've got the bulk of the matches but there are a few remaining cases to deal with. The most

obvious is that "lace-making", from the beginning of *Fit the Sixth*, is currently not matched. At least the "lace-" part is not and we likely want to include it.

The problem here is that the regex engine is chopping up hyphenated words because it interprets a "-" as not being a word character and so it is not captured our word character token. So, we can add a hyphen and a quantified word character token in front of it.

```
\w+-\w+ing\b
```

This captures "lace-making" but at the cost of excluding all our previous matches! We clearly need some way to make the leading characters in the hyphenation optional to match.

To do this we will use a new quantifier, the `*`. Where the `+` says, "one or more of the preceding token should be matched" the `*` says, "zero or more of the preceding token should be matched".

```
\w*-*\w+ing\b
```

This captures more of what we want, "lace-making" is now returned in full *and* our previous matches have been restored. Our total count is still 90 matches but now these matches are more accurate. However, it takes *significantly* more steps to carryout, 347,123 vs 85,967.

but this solution has two detriments, it is not specific enough to handle all circumstances and it is not efficient. From the specificity perspective the statement leaves open the possibility that strings of words joined by hyphens will be overlooked (e.g. `fine-lace-making`) and that words with multiple hyphens (e.g. `lace---making`) will be included (a possible problem because in some cases multiple hyphens are used to stand in for dashes). We may be fine with the latter case on the reasoning that it is a spelling error but we would likely want to capture the first one no matter how many words and hyphens. One way to do this is to specify a set of characters, any of which might be chosen, as follows:

```
[\w-]*\w+ing\b
```

This works but we should immediately see that we can reduce it by removing the `\w+` since the `[\w-]` makes it redundant.

```
[\w-]*ing\b
```

This move also significantly reduces the number of steps to find all 87 matches.

> How could cases with multiple hyphens side-by-side be ignored?

Can use the ability to specify the number of times a character is repeated with {n,m}.

It would seem that we are done unless one is either familiar with the text or stumbles across a pair of troublesome cases, as we do here: 2. "bathing-machines", towards the end of *Fit the Second*, is currently included. At least the "bathing" part is. Same goes for "Thing" in "Thing-um-a-jig" in the middle of *Fit the First*. 3. "charity-meetings", near the middle of *Fit the Fifth*, is missed entirely.

> Whether or not these cases are to be included or not is a matter of the searcher's preference. We'll consider all the cases here, getting our hands quite deep in the process. Note that this is usually how working with regex goes, 90% or more of the information is captured with a simple initial expression and it then becomes a matter of fine tuning the expression to capture all and only what is wanted. Capturing *only* what is wanted is usually the easier fix because false positives are captured, allowing them to be displayed making the need for changes easier. Capturing *all* of what is wanted is more difficult because familiarity with both the underlying text and the possible variations that it might contain is needed, something that becomes increasingly difficult as the size of the corpus being searched is increased.

To get rid of `Thing-um-a-jig` we need a way to assert that the `ing` is at the end of the word. Another way to put this is that once we find an `ing` we want to make sure that unwanted things like other word characters and hyphens do not follow it. For this we use a negative look ahead at the end of the potential match `(?! ...)`.

```
[\w-]*ing(?!-|\w)
```

By doing this the words returned drop from 87 to 85 as we drop `Thing-um-a-jig` and `bathing-machines`.

> Find all the words that have "ed" in them somewhere.

First and Last words

(Good for anchors) It may be necessary, as when doing some analysis in literature to pull all the first or last words from a poem or similar for analysis. This is easy to do by hand for short poems but quickly becomes tedious with longer verses. *The Hunting of The Snark* is a case in point.

This task becomes fairly straightforward with two tokens: `^` and `$`. These tokens are known as *anchors* because they have fixed locations relative to the other characters and tokens that we have seen so far. A `^` matches the beginning of the string and a `$` matches the end of the string, neither capture any characters. So we can start grabbing the first word in a line with: `^\w+?`.

You'll note that this doesn't match anything! There are two obstacles that we must overcome. The first is that between the first word and the start of each line is a lot of space and since regex is very precise we must deal with this.

That's right, by adding a `*` or a `\W` to make `^ *\w+` or `^ \W\w+`. The problem now is that we are capturing all of this, including the spaces. The easiest way around this is to add a capturing group: `^ *(\w+)` or `^ \W(\w+)`

We'll still capture the entire match but we'll also capture separately just the word that we are after. You'll see that in the regex101.com match information box. We will come back to capturing groups.

Now you'll note that currently the regex engine is looking at the poem as a single string so it only has one start and so we are only grabbing the first word in the file. To grab the first words on each line we need to turn on multi-line strings otherwise we'll be stuck with only the start and end of the file for anchors. We do this in regex101 by clicking on the flag at the end of the line and choosing the multi-line option. With this option on a `^` will match the end of each newline as well as the very start of the file.

Ideally we'd be able to just use `^\w+` but we can't here because of the spaces. How can we include all the spaces?

Word in context

`(\w+\W+){0,7}snark(\W+\w+){0,7}`

Use the above to introduce `\W` `\s` and `\S`. Note the order of the `\w`'s and `\W`'s.

Note that ` ` is not the same as `\s`! A ` ` is a literal space character from the spacebar. Depending on the regex flavour a `\s` will include additional whitespace characters like tabs (`\t`), newlines (`\r` and/or `\n`), form feeds (`\f`), and vertical spaces (`\v`).

Duplicate words

(capturing groups and group tokens) \W(\w+?)\W+?\1\W

Example: Swapping name and email in a form

Wildcards `{[^}]*}` vs. `{.*?}` Saying what you don't want

Lookarounds

Fancy Moves

`\K` Resetting the Match

Resources

Tutorials

http://www.rexegg.com/ http://www.regular-expressions.info/

Online Testers

https://regex101.com/ http://www.regexpal.com/ = http://www.regextester.com/ http://regexr.com/
http://myregexp.com/

Source material

Hunting of the Snark Moby Dick