Summary

Regular expressions are used to find strings in text documents. Strings are sequences of characters and regular expressions excel at finding them. If you need to find anything beyond *exact* character strings with or without general case insensitivity then it is time to consider leaving behind standard tools for finding text and embracing regular expressions.

This walkthrough is meant to be used as a loose script and general resource to be shared as part of a standalone, 3-hour workshop at DHSI2017.

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. This section will be organized by the difficulty of solving the problem rather than by technique.
3. *MAYBE*: Looking at how regular expressions are deployed in three common environments: command line, Python, and R. If this is incorporated then it will only be about the basics of deployment, just enough to get people started since familiarity with these environments should not be assumed. 30 min

Overview of Real-World Examples

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. 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.
3. Word in context.
4. 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...

Workshop Cases

In order of presentation and increasing difficulty:

1. Exact string matching.
2. Working with case sensitivity.

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 principle 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 (Participants can just paste the poem in the regex101 window and delete these from there). The Preface should also be removed. All examples will be based on this case unless explicitly noted.

Everyone should open two tabs, each with https://regex101.com/. One for the snark and one for smaller examples.

Principles to illustrate:

Declare these up front and then return to them over the course of the workshop to illustrate them

1. Balance effectiveness and efficiency.
2. Know your data.
3. False positives are easier to detect than false negatives.
4. Specificity is speed.

#1 Exact String Matching

Cut and paste the first piece of sample text into the `TEXT STRING` box. In the `REGULAR EXPRESSION` box we'll begin searching for instances where the concept of a snark is being used 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
>
> **snark** matches the characters `snark` literally (case sensitive)
>
> Global pattern flags **g modifier**: global. All matches (don't return after first match)

This also hints at 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. Note that there is currently a mistake with the description of the global flag/modifier. Turning it on *does* return matches after the first.

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.

At this point 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.

2. Case Sensitivity

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 because while, possibly, related to the idea of *a* snark it is not the same thing a snark. 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 each?

> How many different capitalization variants are there for the string "snark"?

Suppose for a moment that we don't care about the concept snark, just the string *and* that we believe that the title is *not* properly part of the poem. If this is the case then we need to ignore the all caps cases and grab both the cases with a captial "s" and those with a lower case "s". There are three ways to do this:

search for `snark` then search for `Snark` and then use some outside tool to combine them

or

```
snark|Snark
```

or

```
[Ss]nark
```

or

```
(snark)|(Snark)
```

The second uses (capturing) groups and disjunction while the third uses sets.

> Use regex101 to have the participants explore the differences between these three options.

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. There is now lower-case instance of "beaver" in this poem, but there could be (Carroll could easily have had a character "beavering about") and we might want to exclude this by ignoring instances with lowercase letters. 'Friend' is a word where we we'd want both the lower and the upper case variant.

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

#2 - Tokens and Quantifiers

Let us prompt a deeper 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.

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".

> How can we use regex101.com to tell us *exactly* what a `\w` means?
>
> What characters might we think are part of a word that regular expressions exclude? What characters might we think are not part of a word that

> regular expressions include?

Let's try using `\w` 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
```

> Use regex101 to find out what the `+` means.

You will see that this now solves the first problem and we are able to capture entire words ending in "ing". 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.

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

This gives us 87 matches.

> Removing the `\b` from the beginning gives us the same matches, so why include it? Investigate with regex101 to find out.

> What does a `\s` stand for? What changes when we substitute it for the `\b` tokens in our previous solution?

An important aside about being specific about the range of quantification

It is entirely possible that what we are interested in is not *any* word ending in "ing" but only the subset of this group that has a certain number of characters, like six or even the set between five and six characters.

To do this we can use the ability to specify the number of times a character or set of characters is repeated with {n,m}.

As an example, suppose that we wanted to capture all instances where the string ended with "ing" where the total length of the word was either six, seven, or eight characters long. This could be done with:

```
\b\w{3,5}ing\b
```

We must always specify a floor when using this quantification syntax and it can be zero or any larger integer. It is not necessary to specify a ceiling though. If you wanted to capture all the words ending in "ing" that had six or more characters then you could do so with:

```
\b\w{3,}ing\b
```

3 Further Complications

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 by our word character token. So, we can add a hyphen and a quantified word character token in front of it.

```
\b\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".

```
\b\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:

```
\b[\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.

```
\b[\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?

It would seem that we are done unless one is either familiar with the text they are working with or stumbles across a pair of troublesome cases, as we do here:

1. "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*.
2. "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 `(?! ...)`.

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

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

Whether or not pluralizations count when capturing words ending in "ing" is a debate for elsewhere. Here and now we just want to make sure that we could capture them if we needed to. We can include such words by using the `?` quantifier.

> What does regex101 say that the `?` quantifier does and how might we use it to include pluralized versions of words ending in "ing" in our matched set?

Knowledge test

You should now have enough knowledge about regular expressions to be able to build a seach that will grab all and only things that *you* count as words from *The Hunting of The Snark*. Before we start this note that there are really two steps to the problem. First and foremost is identifying the sorts of cases that you expect so make sure that you scan the poem and make a list of these (possibly in a second regex101 window). With this list in hand you can move forward with writing the regex. You can forego the first step but this is a rookie mistake in all but the simplest cases. Unless you know the documents you are working with *better* than the proverbial back of your hand *and* you have lots of regex experience the two step approach is the better one.

Note that it is likely the case that during this process you may hit a "timeout". You can increase the allowable execution time by opening the menu from the top left corner, choosing "settings", clicking on the main page and then modifying the timeout default.

> 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.

> With this information in hand write a pair of regular expressions, one to select the first word of each line and one to select the last.

Word in context

At this point this is a challenge that the workshop should be able to handle as well since they have seen all the elements.

`(\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

The `.`

So far we've been silent about this, which is good because good regex is as explicit as possible about what is being matched. Still, the `.` is a powerful token, especially when combined with greedy and non-greedy quantifiers.

Here we'll use it to grab all the quoted lines in the poem to demonstrate its power.

Start by noting that in the copy of *The Hunting of the Snark* from Project Gutenberg the opening quotes are `“` rather than `"` so copy these into the REGULAR EXPRESSION box. You will note that all the opening quotes are now matched. Add a `.` after the `“` and note that now the first character following the `“` is captured *no matter what it was*. Of course we want to repeat this capturing until we get to the closing quotations so let's add a "one or more" quantifier and the closing quotations (again, it is likely best to copy up the `”`).

```
“.+”
```

Scrolling through the search results will reveal two problems:

1. Quotations over multiple lines are ignored entirely.
2. Lines with two quotations in them are treated as one match.

We'll fix the first problem by turning on the single line flag so that `.` is allowed to match line breaks. As soon as this is done we see that the the `.`, as with any quantified token, is *greedy*. This means that it will only pass off to the next token when it can no longer be satisfied by the given string. In this case it matches everything from the first opening quotation to the last closing quotation. This is not quite what we wanted.

This is identical to the second problem just noted except in that case the problem was truncated because we disallowed the `.` from matching new lines and so it was greedy only within each line.

The greedy problem is fixed by adding a `?` after the quantifier. This tells the regex engine to, on each step, check to see if the next token would match and, if it does, to pass control to that token.

We use the `?` here like so

```
“.+?”
```

Note that we could also have used

```
“[\w\W]+?”
```

Both return 54 matches and take 3826 steps.

> Write a regular expression that doesn't use the `.` or `\W` and compare the result to the previous.

WARNING: AVOID OVERUSE OF `.`. IF YOU CAN BE MORE SPECIFIC THEN STRONGLY CONSIDER BEING MORE SPECIFIC.

Input Validation

Imput validation aka Finding/testing for email, phone, or postal/zip codes. This protects web applications from having to deal with crap.

Here's matching a password: `^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{6,12}$`

- 6 to 12 characters in length
- Must have at least one uppercase letter
- Must have at least one lower case letter
- Must have at least one digit
- Should contain other characters

Here's last name: `^[a-zA-Z''-'\s]{1,40}$`

Here's matching a url:
`^(http|https|ftp):[\/]{2}([a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,4})(:[0-9]+)?\/?([a-zA-Z0-9\-\._\?\,\'\/\\\+&%\$#\=~]*)`

- Must start with http or https or ftp followed by ://
- Must match a valid domain name
- Could contain a port specification (http://www.sitepoint.com:80)
- Could contain digit, letter, dots, hyphens, forward slashes, multiple times

Match html tag: `<([\w]+).*>(.*?)<\/\1>`

- The start tag must begin with < followed by one or more characters and end with >
- The end tag must start with </ followed by one or more characters and end with >
- We must match the content inside a TAG element

Things for a future expansion

Matching duplicated words: `\b(\w+)\b(?=.*\1)`

- The words are space separated
- We must match every duplication – non-consecutive ones as well

Replacing content

Swapping content

Lookarounds

Fancy Moves

`\K` Resetting the Match

Resources

Tutorials

- http://www.rexegg.com/
- http://www.regular-expressions.info/
- https://www.sitepoint.com/demystifying-regex-with-practical-examples/
- https://www.loggly.com/blog/regexes-the-bad-better-best/

The Book

- http://shop.oreilly.com/product/9781565922570.do

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