Introduction to Regular Expressions

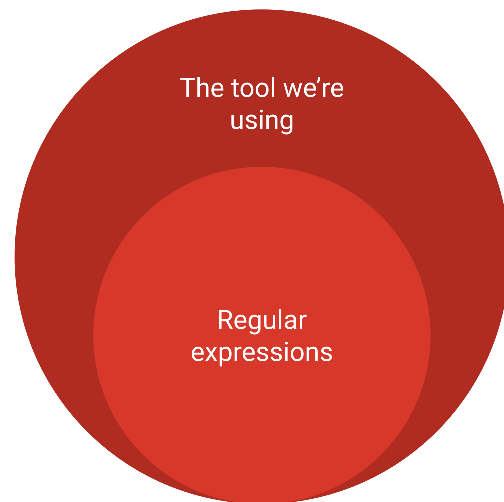
Workshop

A high-level look at Regular Expressions

What can we do with RegEx?

“Regular expressions are the key to powerful, flexible, and efficient text processing. **Regular expressions themselves… allow you to describe and parse text**. With **additional support provided by the particular tool being used**, regular expressions can **add, remove, isolate, and generally fold, spindle, and mutilate** all kinds of text and data.”

Mastering Regular Expressions, 3rd Edition




```
1 const p = 'The quick brown fox jumps over the lazy dog.'; // The string
2
3 const regex = /dog/; // The Regular Expression
4
5 console.log(p.replace(regex, 'ferret')); // JavaScript String.replace() provides 'additional support'
```

Where do we use them?

There are many places we can use Regular Expressions including:

- **A host programming language** - support is either included in the Standard Library (Python etc.) or built into the syntax (Perl, JavaScript etc.) of modern languages
- **IDEs** and **text editors** (very handy for ‘find’ and ‘find and replace’ operations)
- At the **command line** (using Unix tools like grep)

…and lots of other places where there’s a need to search text.

What do they look like?

Regular expressions are sequences of characters that **define a search pattern**. The characters in a regular expression may be:

- **literal** (actual text), or
- **metacharacters** (special characters with special meanings)

Here's a simple example RegEx that makes use of the the 'd' and 'g' literal characters and the '.' metacharacter:

```
/d.g/
```

This matches 'dig', 'dog', 'dzg', etc...

There are different ‘flavours’ of RegEx

- The ways regular expressions are implemented is not always ‘regular’.
- We’ve got BRE (Basic Regular Expressions), ERE (Extended Regular Expressions), PCRE (Perl Compatible Regular Expressions)
- Different tools will use different ‘flavours’ (some, like PHP, support multiple).
- **If something’s not working, it might be that the particular tool you’re using doesn’t support the RegEx feature you’re trying to use. JavaScript, for example, doesn’t support ‘lookbehinds’ - as I learned when first trying to use them!**

Let's start using them

Matching single characters

Here's an example to explore:

- Matching single literal characters
- Regex 'flags'
- Using a '.' metacharacter
- Escaping characters

Using alternation

Use alternation to match either *A* or *B*

The ‘pipe’ character provides alternation in Regular Expressions.

Here’s a very simple example:

- https://regexr.com/4cu8v

Using character sets

- In regular expressions a set of characters is defined using the metacharacters [and]
- Everything between them is part of the set and any one of the set members must match

Here's an example to explore this
https://regexr.com/4bv1d

Using character set ranges

In addition to literal character set ranges, we can use the ‘-’ metacharacter to define character set ranges.

Here’s an example: https://regexr.com/4c0mr

“Anything but” matching

Negating a character set using the
^ **metacharacter**

The ^ metacharacter negates all characters in a set or range.

Here’s an example:

http://regexr.com/4c3o5

A closer look at metacharacters

Simple metacharacters and their usage

Metacharacter	Purpose	Usage	Meaning
[ ]	Define a set	[cm]at	Match ‘cat’ or ‘mat’
.	Match any character	.	Match any single character
-	Specify a range	[0-5]	Match 0, 1, 2, 3, 4 or 5
\	‘Escape’ the next character	file\.txt	Match file.txt
^	Negate a character set*	[^cd]	Match anything but ‘c’ or ‘d’

* we won’t go into this now, but ^ has a different meaning when used outside a character set

Other metacharacters and their usage

Metacharacter	Purpose	Notes
\s	Match any whitespace	Includes tabs, spaces and carriage returns
\S	Match any non-whitespace	Equivalent to [^\f\n\r\t\v]
\d	Match any digit	Equivalent to [0-9]
\D	Match any non-digit	Equivalent to [^0-9]
\w	Match any alphanumeric and _	Equivalent to [a-zA-Z0-9_]
\W	Match any **non** alphanumeric	Equivalent to [^a-zA-Z0-9_]

Repeating matches

Using quantifiers

Often we will not know how many characters are in the patterns. We can handle this uncertainty with quantifiers.

Here’s are some examples to experiment with:

https://regexr.com/4cfm9

	Match the previous character...
+	one or more times
?	Zero or one time
*	Zero or more times
{2,4}	Two to four times
{,5}	Up to five times

Quantifiers, greed and lazyness

It's important to understand how quantifiers behave in certain circumstances

By default, quantifiers are **'greedy'** meaning that they will continue matching characters until they're unable to consume any more. We can change this by appending a **?** to the quantifier

Here's an example: http://regexr.com/4cfoh

Position matching

Use word boundaries

Allows us to specify that the match should be at the beginning or end of a word.

- \b will match a word boundary
- \B will match 'not' a word boundary

Here's an example:

http://regexr.com/4cu4l

Note: the boundary metacharacters match a position, not a character

Use string boundaries

Match based on position within the string

- ^ for start of string*
- $ for end of string

Here are some examples:

- https://regexr.com/4cu6n - using ^
- https://regexr.com/4cu79 - adding $ and multi-line mode

Note: we've seen the ^ metacharacter before. It's one of several characters that can have a different meaning depending on context.

Sub-expressions and backreferences

Grouping with sub-expressions

Creating our own 'parts' of a match

Sub-expressions allow us to group parts of an expression so that they are treated as a single entity (for **quantification** purposes or to be clear about **alternation**)

Here's a simple example: https://regexr.com/4cu7u

And here's a common alternation 'gotcha': https://regexr.com/4cu8m

Using back-references

Back references - as the name suggests - allows us to capture a match and refer back to what was matched. This is incredibly powerful:

Here's an example:

- https://regexr.com/4cub1

That's all... for now.

This session was intended to be an *introduction* to Regular Expressions. While we've covered quite a bit of ground, there are some areas we've not covered but I hope this has at least provided a starting point.

I love Regular Expressions, so would be absolutely delighted to help you solve any pattern matching problems you come across.

Here are some useful and fun resources to play with:

- Regex Golf https://alf.nu/RegexGolf
- Regex Crossword https://regexcrossword.com/

Thank you