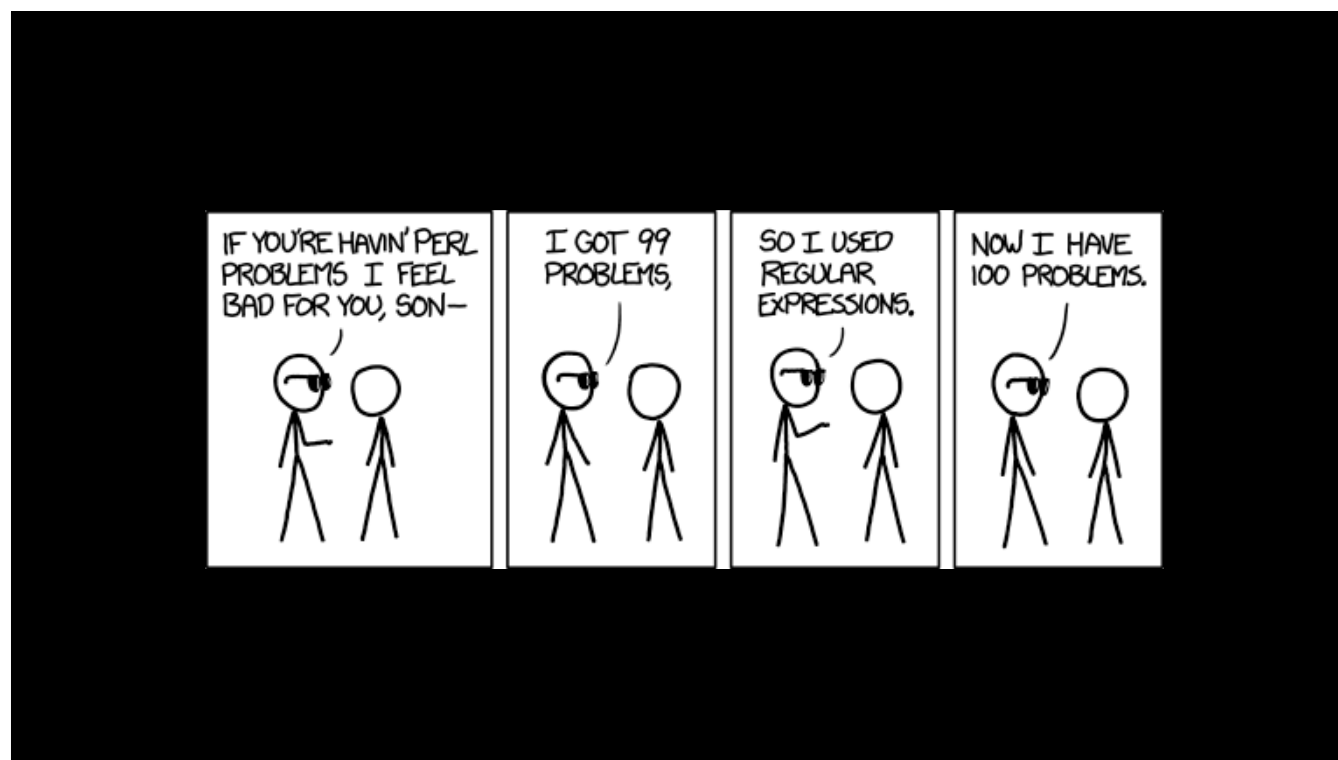
99 problems…
and regex

Mike Dowler

About me



Title came from this XKCD cartoon.

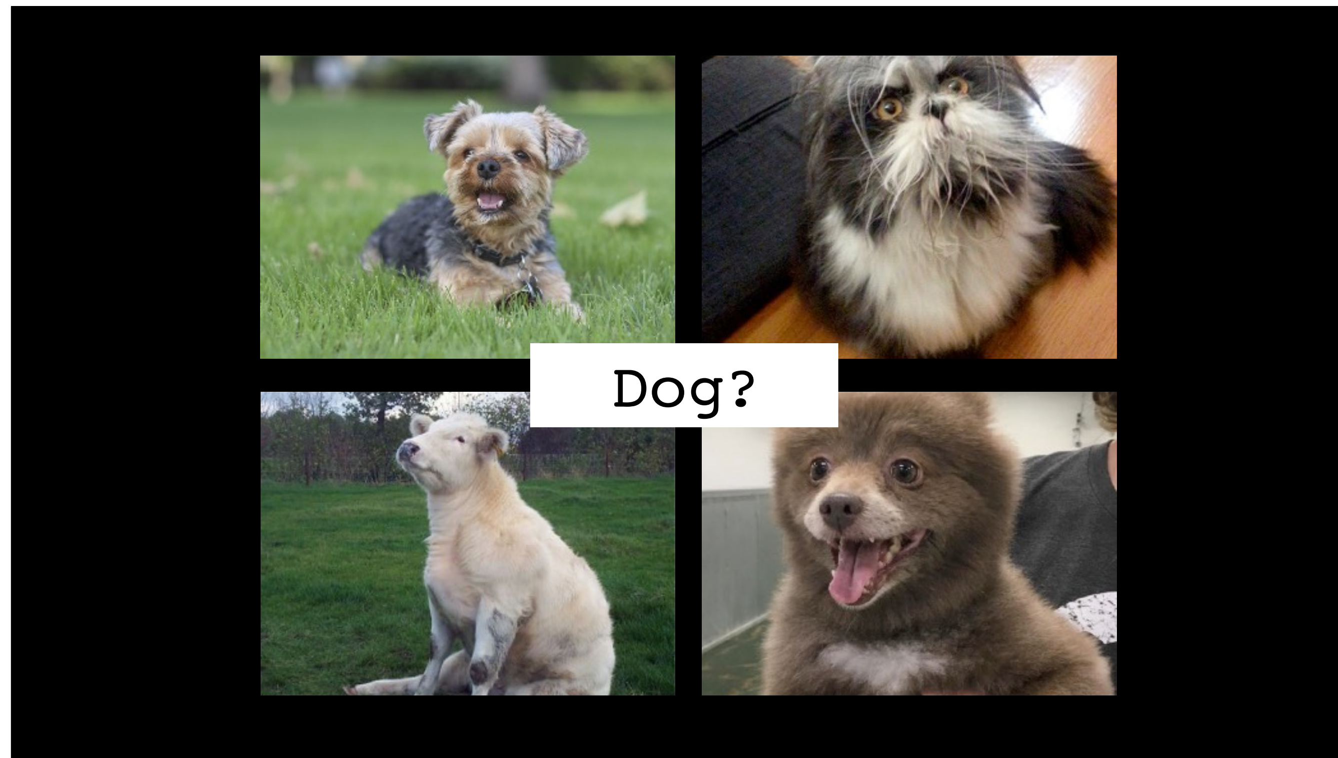
Regex has a reputation of being very powerful *in theory*, but hard to use in practice.

Aim of this talk is to provide practical guide to getting started with regex.

Regex, huh?
What is it good for?

Regex - or regular expressions - is about matching *patterns* in text strings.

(I'm going to say regex. You can also say regex. I don't really care.)



I have 3 kids. When my oldest was a toddler, he started out calling all animals “dog”. It’s kinda easy for adults - you can probably see that only one of these animals is a dog. But children have to learn that skill. And if you think about, trying to explain how you know whether an animal is a dog or not, is actually really difficult.

It’s a similar level of difficulty if we want a computer to look for particular patterns in a text. We have to think about the edge cases - what are the features that, if present, would cause a string to match or not match a pattern.

My aim in this talk is just to introduce regex. We won’t be covering any super advanced features. I want to show how you might go about using regex in a couple of practical scenarios.

Scenario #1: Version numbers

How do we check if a software version meets or exceeds a given value?

4.9.0.504

5 ✔ 4.9.9.9.9 ✔ 4.9.0.0.504 ✘

4.9.0.503 ✘ 4 ✘ 4.10.0 ✔

Problem posed by @Tobes on MacAdmins Slack. (e.g. we want to avoid scoping a software installation to computers which already have that version)

Difficult because:

* the number of groups of numbers might change
* the number of digits within each group might change

Write a script?

* split the version string into groups.
* compare each group separately
* store the outcome somewhere
* read the value to decide on the appropriate action (and hope that the installed version hasn't changed in the meantime!)
* what if it includes letters?

Strict comparison? (i.e. version number IS NOT)

* fine when you only have 2 or 3 versions to cope with
* doesn't work well in the wild

Start simple

```
4.9.0.504
```

```
^4\.9\.0\.504
```

```
3.14.9.0.504.7
```

When I'm writing a regex, I like to start with the most simple case, and build it up from there. So, let's start by just matching the version number exactly.

Regex is looking for a string pattern, so we can just use the string we want to match. Except that, in this case, the period acts as a special character, so we need to escape it with a backslash.

That *would* match our version number, but there's a problem - it would also match this version number, where it appears in the middle. And that version number is actually lower, so that would give us a false result.

Regex allows us to avoid that with two more special characters. Caret (^) at the start means "look for this at the start of the line we are comparing against", whilst dollar ($) at the end does the same thing at the end of the line. If we have both, we are saying look for this text, and only this text - nothing in front of it on the line, and nothing after it.

Now in our case, we don't care about anything coming after, so we just need the caret. And it's pretty good practice to use it if you can - it will improve your performance.

Build it up - one digit

```
4.9.0.504
```

```
^4\.9\.0\.50[456789]
```

```
^4\.9\.0\.50[4-9]
```

OK - so we had a regex to match the version *exactly*, but we were looking for the version *or greater*. We need to think about how the version number string will change as the version increases.

And, obviously, the first digit that might change is this one at the end. If we want to allow the version number to increase at this digit only, then it can be anything from a 4 all the way up to a 9.

Regex allows us to specify alternative characters at a single location using square brackets. Inside the brackets we can list the alternative characters individually, or we can set out a range. (This works for letters as well as numbers)

Build it up - two digits

4.9.0.504

```
^4\.9\.0\.5[1-9][0-9]
```

```
^4\.9\.0\.5[1-9]\d
```

As that last group in the version number increases again, it's going to tick over to the next digit, so that becomes 1. And the last digit will start again from zero - giving 4.9.0.510. We can keep going like that all the way up to 599.

Remember, we DO need to include that last digit, because we don't want to match a 2-digit number like 51. It has to be 510 or greater.

Because the last digit now covers the full range from 0 to 9, we can use a shorthand of \d to mean "any digit".

Build it up - one or two digits

```
^4\.9\.0\.50[4-9]
```

OR

```
^4\.9\.0\.5[1-9]\d
```

```
^4\.9\.0\.(50[4-9]|5[1-9]\d)
```

So, now we have two alternative expressions - one for the exact version number or changes to the last digit only, and one for changes to two digits.

You could have those as alternative regex lookups, but that gets unwieldy. We want to combine them in a single expression.

Regex uses a single pipe to indicate alternatives. And we wrap the section in parentheses - that way we don't need to repeat the sections that don't change.

It can seem unsatisfactory to have to use two alternatives, but that's because regex is looking for *string* patterns. It doesn't know or care whether those represent a number, and so it can't do maths. We need to put in the hard work to determine what the number will look like as a string.

Build it up - three digits

4.9.0.504

```
^4\.9\.0\.(50[4-9]|5[1-9]\d|[6-9]\d\d)
```

```
^4\.9\.0\.(50[4-9]|5[1-9]\d|[6-9]\d{2})
```

We can continue that logic as we expand to cover the last *three* digits. Beyond 599, we are looking for any 3 digit number starting with a 6 or greater.

Instead of repeating the \d character in that last option, we can say that we want that character to appear exactly 2 times. We do that by including the number in curly braces.

So now we have a regex that will match version numbers from 4.9.0.504 all the way up to 4.9.0.999.

Build it up - beyond three digits

```
4.9.0.504
```

```
^4\.9\.0\.(50[4-9]|5[1-9]\d|[6-9]\d{2}|\d{4})
```

```
^4\.9\.0\.(50[4-9]|5[1-9]\d|[6-9]\d{2}|\d{4,})
```

Suppose the developer keeps releasing minor updates to this software, and that last section of the version number ticks over from 999 to 1000. Up until now, we've only been looking at the first three digits, so we want to make sure that we don't ignore 4-digit numbers.

We do that with another option in our group here, for a 4 digit number.

But that only works for a 4-digit number - what if the developer goes crazy, and starts issuing releases 4.9.0.10000 and beyond???

Instead of just specifying a single number in the curly braces, we can set a range defined by a minimum and maximum, separated by a comma. We can even omit one of the minimum and maximum, and the range will extend as far as possible.

So we have "4," without a maximum, we are saying "look for any number with 4 or more digits".

Quantifiers

```
? = {0,1}
```

```
* = {0,}
```

```
+ = {1,}
```

There are a couple of shorthands here as well. ? means zero or one. * means zero or more. And + means one or more

Version numbers

`4.9.0.504`

```
^(4\.(9\.(0\.(50[4-9]|5[1-9]\d|[6-9]\d{2}|\d{4})|[1-9])|[1-9]\d)|[5-9]|[1-9]\d)
```

Following this logic through, we get this regex. Simples!

Scenario #2:
Log file parsing

How can we extract (semi-)unpredictable data from predicatable contexts?

```
<backupConfig>
      <backupSets>
        <backupSet id="x">
          <name>yyyyy</name>
          …
        </backupSet>
        <backupSet id="x">
          <name>yyyyy</name>
          …
        </backupSet>
```

This is another real-world problem. I needed to read from a log file and extract some data.

Specifically, I wanted the backup set id “x” (numeric, number of digits unknown) and name “yyyyy” (generally alphabetic or alphanumeric, possibly including special characters) for each backup set. We don’t know how many of those there might be.

Find the string - part 1

```
^\s*<backupSet id="\d+">
```

`\s matches space, tab or newline`

`\d matches any digit`

I know that, in the log file format, the XML has been prettified. The tags we want will be at the start of a line, but may be indented.

I don’t really care if the indents are spaces or tabs (sacrilege!)

I know the rest of the tag, and I know that the id will be a number with at least one digit.

Find the string - part 2

```
^\s*<backupSet id=“\d+”>\s*<name>.+<\/name>
```

. matches anything *except* newline

/ needs to be escaped

Then we have a bunch of whitespace before the <name> tags (which includes a newline, and possibly some spaces on either side).

The text in the name tag could be anything at all. The only thing the period *doesn’t* match is newlines, so we can be sure that it won’t go beyond the end of this line. This probably isn’t the best implementation, but it’s good enough for our purposes.

Find the data in the string

```
^\s*<backupSet id=“\d+”>\s*<name>.+<\/name>
```

```
^\s*<backupSet id=“(\d+)”>\s*<name>(.+)<\/name>
```

```
^\s*<backupSet id=“(?P<id-cap>\d+)”>\s*<name>(?
P<name-cap>.+)<\/name>
```

The regex we have so far will find the information we need, but the trouble is that it is buried in a whole lot of context that we need to be there, but we’re not really interested in. It would be better if we could just capture the data we want.

Fortunately, regex lets us do this using parentheses to define capture groups. We saw these used previously when we were working with alternatives, but they can also be used to return specific portions of the data.

That’s good, but it can get pretty confusing if we have several capture groups defined. We can make it even better by assigning a name to each group - we can then refer to the capture groups by name in the results. There are a couple of different syntaxes we can use, but this is my preferred one for avoiding ambiguity: question mark, upper case P, then the capture group name in angled brackets

Here we have defined two named capture groups: id-cap for the numerical backup set id, and name-cap for the name of the backup set

Practice!

https://regex101.com/

Really useful site for building and testing regexes.

1. Enter the regex
2. Enter test strings - think about the edge cases!
3. Explanation of your regex - check that it is matching what you expect
4. Reference, in case you can't remember what special character/syntax to use

Practice!

https://regexcrossword.com/

	UB|IE|AW	[TUBE]*	[BORF].
[NOTAD]*			
WEL|BAL|EAR			

If you like nerdy word puzzles…

Putting regex to work - Jamf

Test group

Computer Group | Criteria | Show in Jamf Pro Dashboard

AND/OR		CRITERIA	OPERATOR	VALUE	
	(	Application Title	is	some_application.app	
and		Application Version	does not match regex	`^(4\.(9\.(0\.(50[4-9]|5[1-9]\d|[6-9]\d{2}|\d{4`	)

As I hinted earlier, Jamf allows regex directly for version numbers in the definition of a smart group. This is the only way to cover *ranges* of version numbers, and is a great place to start.

Putting it into practice - python 3

```
>>> import re

>>> result = re.match( r'^\s*<backupSet id="(?P<id-cap>\d+)">\s*<name>(?P<name-cap>.+)<\/name>', applog )

>>> result.group('id-cap')
'66'

>>> result.group('name-cap')
'Mike's Backup'
```

https://docs.python.org/3/library/re.html

In terms of scripting, python offers great support for regexes.

The re module contains a bunch of functions we can use - match is the simplest. In this example, we are using the regex I showed earlier to match against the string stored in the variable applog. We than call those named groups by name.

One caveat to be aware of is that regex's use of the backslash conflicts with python's use of the same character for the same purposes. This would normally require any backslash to be escaped (with another backslash). A better approach is to use "raw string notation", by prefixing the string with an r. This ensures that python does not attempt to interpret backslashes in the regex string.

Putting it into practice - zsh

```
% regex="<backupSet id=\"([[:digit:]]+)\">"

% if [[ $applog =~ $regex ]]; then
    echo $MATCH         // <backupSet id="66"
    echo ${match[1]}    // 66
  fi
```

http://zsh.sourceforge.net/Doc/Release/Conditional-Expressions.html

Both bash and zsh provide support for regex. Of course, you shouldn't be writing scripts in bash anymore, so I'm just going to talk about zsh. There are various ways to make use of regex, but the most straightforward is to use the regex-match operator (=~) in an if statement.

If you Google zsh regex, you might be excited to find out that it can support PCRE regex - the same format as we have been using all along. HOWEVER, zsh on macOS doesn't include the PCRE module by default.
It does support POSIX regex. There are a couple of differences - the \ character classes don't work (\s, \w, \d etc). Instead we can specify character classes with [:digit:] etc. These have to be enclosed in the normal square brackets used to indicate alternatives.

The main search result is returned as $MATCH. If there were subgroups (parentheses) in the regex, then they are returned in the array ${match}. NB by default zsh array indexes start at 1! I haven't yet found a way to use named groups in zsh.

Summary

- Methodology
- Positional anchors - ^ and &
- Alternatives - […]
- Classes - \d, \s, \w
- Quantifiers - {x,y}, ?, +, *
- Groups - (…)

Questions?