Using Regular Expressions More, um, Regularly

a "medium deep" dive

Presented by Mark Minasi

@minasi

Agenda

- Why regex?
- Basics: Literals and metacharacters
- Basic wildcards
- Engine internals: how regex matches
- Greedy and lazy wildcards
- PowerShell's regex tools
- Character classes / class operations
- Anchors
- Groups
- Character class subtraction

What Can Regex Do for You?

- Very flexible text pattern match tool… when "-filter ADDC1*" won't do it
- The gateway to advanced pattern recognition ("is there PII in any of these files?") or more exacting search-and-replace
- Simplifies input validation
  - "Is that a valid email address?"
  - "Does that desired new password meet our complexity requirements?"
- Extract text from even weakly structured folders of files
- Find and update text, like web sites
- Run it against a file of all words and become a crossword fiend
- Not just PowerShell… Server file classification, ASP.NET, VBScript all support it
- Imagine it for OneNote
- About as cross-platform a tool as you can find

Regex Weaknesses

- It's really weird looking, almost a "write-only" language sometimes
- It's not as hard as it's made out to be, but it is *not* trivial
- It's not a programming language – you can't write procedures/functions in it, alias oft-used patterns and the like
- It parses well, but only to a point; it's lack of recursion means that going after, say, XML would be difficult – it's just the wrong tool for that
- (Another example: "is this word a palindrome?" using regex)

The World's Simplest Regexes

"Literals" and "Metacharacters"

- The regex pattern to match "be" is just those two letters – "be"
- "be" would first match
  - To be or not to be
  - Many consider Abe Lincoln the best President
- The letters in "be" are called *literals* (rather than *metacharacters*)
- Simplest metacharacter is . or "dot;" it matches any character other than a line termination (usually… more on that later)
- "be." would match bee, bet, abet, antebellum, bear, etc
- "be." would *not* match just "be"
- BTW, to actually match "period," use \.
- Example: to match will.i.am, use will\.i\.am

Reference: The Other Metacharacters

- \ backslash, "escape"-ish
- ^ and $ are "anchors"
- | acts like "or"
- * means "repeat 0 or more times," + "repeat 1 or more times"
- (and) surround *groups*
- [] surround classes
- {} specify how many matches to expect
- We'll see more of them later

Basic Regex Pattern Testing: -Match

- To see that "But Be or not to be" can match the pattern "be," type
- **`"But Be or not to be" –match "be"`**
- Returns $True or $False
- The matched text is stored in $matches
- -Match only returns one match; better tools soon!

```
PS C:\scripts> "But Be or not to be" -match 'be' ; $matches
True

Name                           Value
----                           -----
0                              Be
```

- PowerShell regex is by default case-insensitive, unlike most regexes

More on Dot

- In pattern b.e, "." only matches one character (albeit any character), so it's not that "wild" as wild cards go
- So it'd match oboe or able, but not bone or be – there must be one and only one character of some kind where the "." is,
- Suffix + to anything and it means "match this one or more times"
  - So .+ is like the familiar "*"
- Suffix * to anything and it means "match this *zero* or more times"
- Thus, b.+e matches "Bayer" but not "burn" and not "be"
- (Why not "be?")

The Regex Engine

Yeah, we did some basics. Here's why.

Meet the Regex Engine

yes, we have to know this to use regex effectively

- So… regex matches patterns to text, and it *seems* intuitive…
- …. but sometimes it isn't
- Knowing how it works is really key to making regex useful
- And, truthfully, answering, "Why doesn't this stupid thing *work*?")
- Consider finding "be" in "But Be or not to be"
- Regex scans left to right (although it can be told to right-to-left) looking for a match to the first token in the pattern, which is "b" in our case

Matching "be" to "But Be or not to be"

- Start at position 1
- There's a B! We're *matching*!
- Next in pattern is "e…" next in the string is "u." Dang. No match. Increment the "beginning of possible match"
- Position 2="u," no match w/"b"
- Position 3,4 no match
- Position 5=b, a match, we're *matching*!
- Next in the pattern is "e," check the string's next, and it's an e! We're still matching!
- Check the next character in the pattern, and… none left
- We matched at position 5 and 6!
  - Search returns "true"
  - $match = "be"

Note This Important Fact

- Big concept: regex always returns the leftmost answer, even if wasn't the one we expected or wanted
- That was a test of "But Be or not to be"
- Now what happens if we match "But Beer or not to Be"…. What matches?
- Yes, it's a trivial example but it points out that in "Beer or to Be," most of us would see the final "Be" as somehow a better match

Engines *Backtrack*… Try b.+e against "But Be or not to be"

- Again, "match pointer" starts at 1 – a "b" – so we're *matching!*
- What does .+ match? Everything! So the engine matches the rest of the string… we're *matching!*
- So now, "b" matches "B" and .+ matches "ut Be or not to be"
- Ah, but the pattern's not done – we got the b.+ but not the "e," as we used up all of the letters with .+
- Curses! Foiled by a mite too much matching
- In this situation, the engine knows to "backtrack." It backs up one position, and now .+ matches "ut Be or not to b" and so we've consumed all but the last letter, "e"
- The engine tries to match the pattern's last token (e) to the string's last token (e) and success!!… it matched "But Be or not to be" in its entirety

Greedy Versus Lazy

- Consider "berrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr" matched against b.*e – remember, * means zero or more
- (BTW, * and + are called "quantifiers" … we'll meet more)
- The obvious match is "be," but how many iterations would the engine need to backtrack to that?
- Or maybe b.*e.*r … do THAT one on a blackboard. :)
- This is why regex processing times can get exponential
- This behavior of * or + to consume as much as possible and then only backtrack when forced to is called "greedy"
- You can make any quantifier "lazy" by suffixing it with a question mark
- Our earlier pattern then becomes not b.+e but b.+?e

How the Engine Handles Lazy Quantifiers

"But Be or not to be" test matched to b.+?e

- As before, "match start pointer" at 1 finds a b and we're matching
- Next is .+? – "match any character one or more times, but be lazy doing it"
- *Now* .+? doesn't consume the rest of the string, it just consumes the first available character… "u"
- "Match so far" is "bu"
- Check next token in the pattern, which is "e"
- That fails so the engine "backtracks and expands," matching .+? to "ut"
- That fails but ultimately it expands to "ut B"
- Final pattern "e" matches
- Result: "But Be"

PowerShell and .NET's Regex Tools

There's way more than –match, thankfully

The PowerShell Tools: String Operators

- The –match, -replace and –split operators take regex
- Of them, only -match populates $matches
- Try "This is a sentence" –split " " for example
- There is also –cmatch (it's case-insensitive by default), -notmatch, creplace etc and the largely unnecessary –imatch, which forces case insensitivity

Case Matching Example

```
PS C:\scripts> "Return soon" -match "return"
True
PS C:\scripts> "Return soon" -cmatch "return"
False
```

The PowerShell Tools: [regex] Class

- PoSH's Regex is the .NET implementation and has an accelerator
- $myreg = [regex]'hel.'
- "Helo hela Held help" -match $myreg
- Reports "hela" match and "true"
- The Matches method is more powerful
- $matchups = $myreg.matches("helo hela helt help")
- That would return *four* matches… better than –match and $matches
- But the mext query only returns *two*; why?
- $matchups = $myreg.matches("Helo hela Helt help")
- Reason: .NET regex is case *sensitive* by default

Getting Multiple Matches w/.NET Regex

```
PS C:\scripts> $myreg = [regex]'hel.'
PS C:\scripts> $matchups = $myreg.matches("helo hela helt help")
PS C:\scripts> $matchups | format-table -auto

Groups Success Name Captures Index Length Value
------ ------- ---- -------- ----- ------ -----
{0}        True 0    {0}          0      4 helo
{0}        True 0    {0}          5      4 hela
{0}        True 0    {0}         10      4 helt
{0}        True 0    {0}         15      4 help
```

Getting Insensitive: Options and Mode Mods

- Two more ways to regain case insensitivity is via .NET regex options or regex "mode modifiers"
- $regex = new-object regex('hel.', ([System.Text.RegularExpressions.RegexOptions]::MultiLine,[System.Text.RegularExpressions.RegexOptions]::IgnoreCase))
- Or prefix your pattern with "(?i)," a mode modifier:
- $myreg = [regex]'hel.'
- Becomes
- $myreg = [regex]'(?i)hel.'
- There are more mode modifiers, more later

Handling Timeouts: In Case You Meet ∞…

- It's easy to accidentally create regexes that go on forever
- In .NET 4.5 and later, you can create a regex constructor with a timeout
- .NET needs the timeout built as a TimeSpan; example:
- $maxtime = new-timespan -seconds 1
- $myreg = New-Object -TypeName regex -ArgumentList 'A.', ([System.Text.RegularExpressions.RegexOptions]::MultiLine,[System.Text.RegularExpressions.RegexOptions]::IgnoreCase), $maxtime
- $matchups = $myreg.matches("Aces axis asking")
- BTW, on my Win 10 system it seems that all regex matches take at least about 4.5 ms

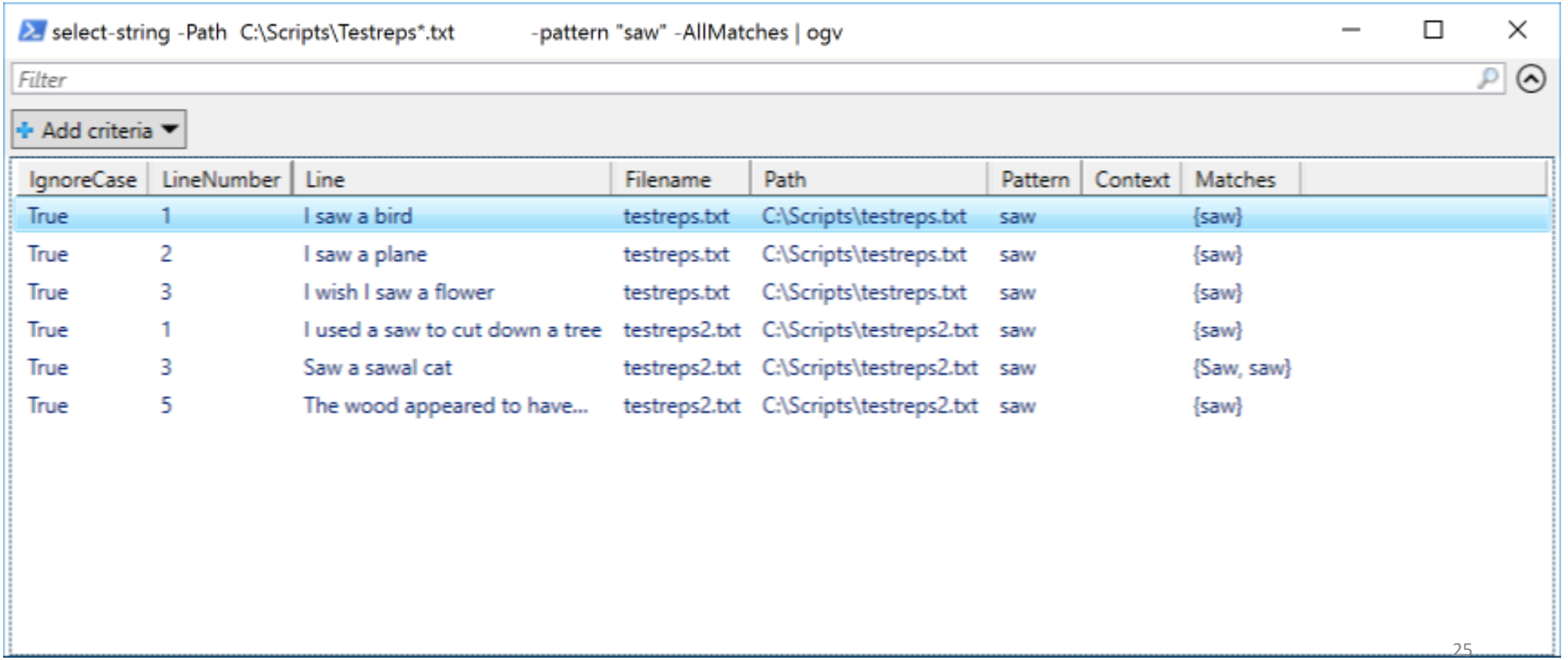
PoSH's GREP: Select-String

- Input a file via –path or a text string via –inputobject
- Put the regex in –pattern
- Follow case with –CaseSensitive
- Only reports one match per line unless –allmatches
- Returned object contains filename, line number, line
- Does "match" only, no –replace or –split type options
- Example:
- Select-String –path c:\files*.txt –pattern "saw" -allmatches

Select-String Is *the* tool

- Again, returns all matches if you want
- Info returned includes
  - Filename
  - Line number of match
  - Column of the match
  - The exact text that matched

Sample Output



IgnoreCase	LineNumber	Line	Filename	Path	Pattern	Context	Matches
True	1	I saw a bird	testreps.txt	C:\Scripts\testreps.txt	saw		{saw}
True	2	I saw a plane	testreps.txt	C:\Scripts\testreps.txt	saw		{saw}
True	3	I wish I saw a flower	testreps.txt	C:\Scripts\testreps.txt	saw		{saw}
True	1	I used a saw to cut down a tree	testreps2.txt	C:\Scripts\testreps2.txt	saw		{saw}
True	3	Saw a sawal cat	testreps2.txt	C:\Scripts\testreps2.txt	saw		{Saw, saw}
True	5	The wood appeared to have...	testreps2.txt	C:\Scripts\testreps2.txt	saw		{saw}

Dot-Weakness: Multi/Single Line

- The dot (.) matches everything but the line feed
- (Ancient historical reason – don't ask)
- You can turn on a mode to change that so all of the newline characters match a dot
- "Single line mode" means "dot matches everything" and it's the .NET "RegexOptions.Singleline" option and the (?s) mode modifier
- You'll want this when parsing things like an entire web page read into a variable
- Select-String is great in that it generally elides this – no need to worry, it takes care of it

Use Online Testers and Libraries

- Is this sounding scary?
- Save yourself time with some nice online testers
  - Regexr.com
  - Regex101.com
  - http://regexhero.net/tester/ is great because it's built atop .NET regex, but requires Silverlight, which a certain big company is killing for some reason
- There are also no end of sites with solutions to "how do I write a regex that matches…" questions
- One example: http://www.regexlib.com

Regex Tools: Character Classes

Back to syntax for a bit

Character Classes for a Range of Values

- Literals are easy matches – 1 matches just an actual "1"
- But if we want to match any digit, we could create a "custom character class" with a range, a set, or a combination of the two
  - Ranges look like [0-9] which says, "we're happy if any of these match"
  - "is th5ere a digit" –match "[0-9]"
  - Note the square brackets on a custom class
  - No escape needed to have "-" in a range, just make it unambiguous
  - Sets look like [0123456789] … custom classes are satisfied by *any* member
  - Or put them together: "is th5ere a digit" –match "[a-f567h-j]"

Regex Includes Some Predefined Classes

- [0-9] or [0123456789] works for digits, but there is a predefined class, "\d" that does that job
- And \D, which means "everything but digits"
- \w is "all 'word' characters," a-z, A-Z, 0-9, _ in ANSI ;\W is, again, everything but that
- \s is "white space," which means tab, line feed, vertical tab, form feed, carriage return, space (ASCII 9-13, 20) but not "start of line"
- \S is anything that isn't white space

Mandatory Regex Example

- We must search for any US social security numbers (or things that look like them) in a document or perhaps a folder full of files

```
PS C:\scripts> $s="My ID is 481-12-9256"
PS C:\scripts> $s -match "\d\d\d-\d\d-\d\d\d\d"
True
PS C:\scripts> $matches

Name                           Value
----                           -----
0                              481-12-9256


PS C:\scripts> _
```

Quantifiers: Beyond + and *

- Writing \d\d\d was irritating, so \d{3} works also, requiring exactly three digits
- x{2,7} matches from two to seven consecutive x's
- x{5,} requires five or more consecutive x's
- Again, they must be consecutive; x{2} would not match Xerox

The "Optional" Quantifier, ?

- Suppose we wanted to match either "color" or "colour"
- Could * help, as in colou*r ?
- Yes, but it'd also match colouuuuuuuuur
- So we have "?," the "optional" quantifier
- "Groups" and "Alternation" could solve this also, we'll get to them later
- ? Means "0 or 1," as in colou?r
- You can use parens to make groups of more than one letter optional
- So this very simple example works: colo(u)?r
- (Blame it on Daniel Webster)

A "Negation" Character Class

- What if we *don't* want something?
- Again, there are the predefined "anything but" classes like \D, \W
- Alternatively, to negate a custom character class, put ^ as its first token
- [^0-9] is the same as \D
- To match [^e], all a string must do is to have one or more characters in it that are *not* "e"

Or to Tweak a Class: Class Subtraction

- Suppose we wanted to match a letter:
- [a-zA-Z]
- But didn't want r or w
- Use this:
- [a-z-[rw]]

```
PS C:\> "Row" -match "[a-z-[rw]]"
True
PS C:\> "wwr" -match "[a-z-[rw]]"
False
```

- You can also subtract inside subtracted classes, etc.
- If subtracting from a negated group, the negation happens first, then the subtraction

Quantifier Examples: Finding Weird Words

- Example: find a word with five consecutive vowels, given a file crwords.txt with all English words:
- select-string -Pattern "[aeiou]{5}" -Path .\words.txt
- The character class is clearly vowels; the {5} says, "and exactly five of them." Six consonants would be
- select-string -Pattern "[a-z-[aeiouy]]{6}" -Path .\words.txt

Regex Groups

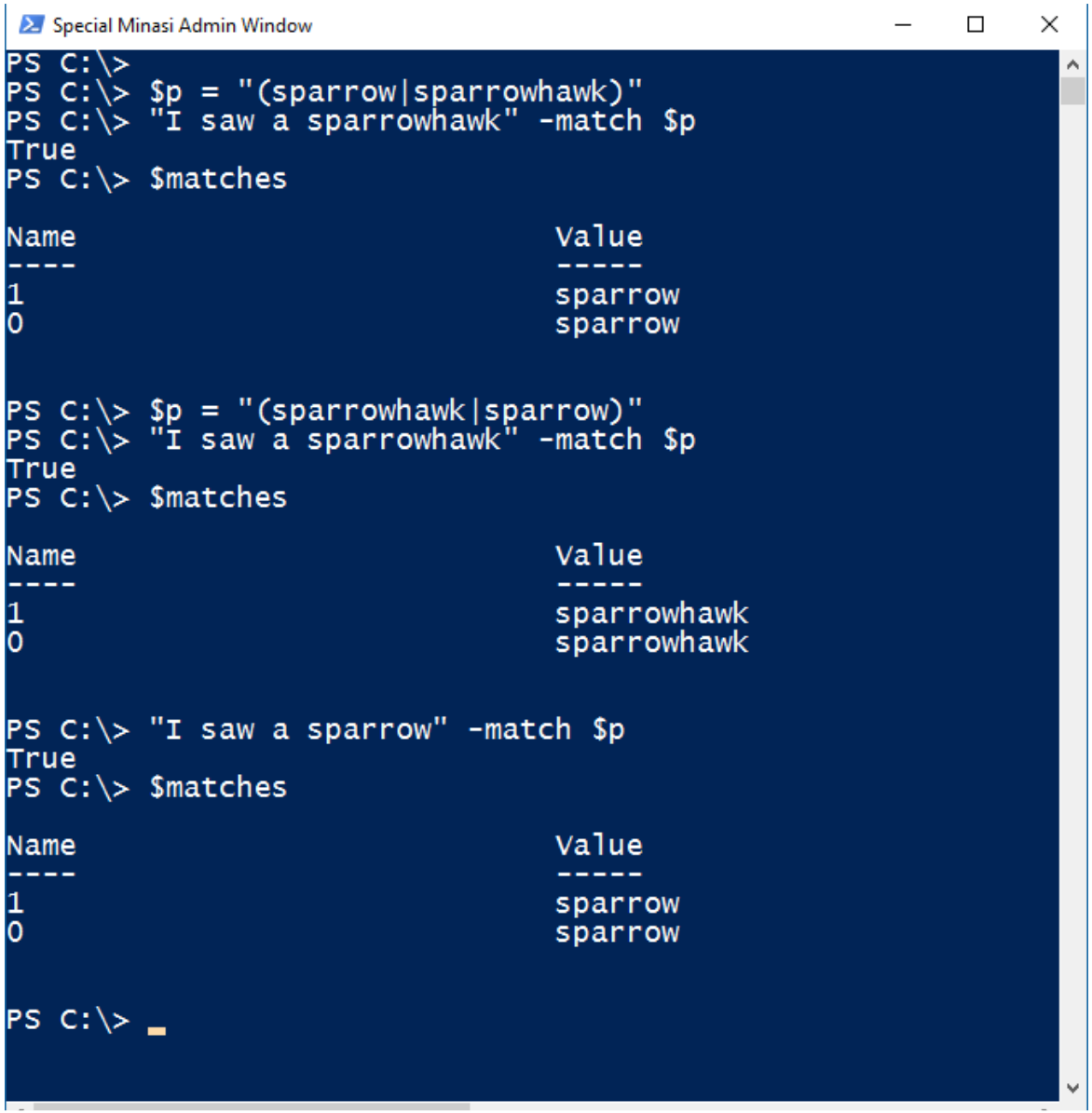
Clarifying, Quantifying, Alternation, Capture

Groups

- "Groups" has a special meaning in Regex
- You "group" part of your regexes with parentheses
- Groups have several functions
  - Sometimes they just clarify a regex visually
  - Like character classes, they take quantifiers
    - (a)(a)(a) is identical to (a){3}
  - They allow you to mark some of the matched pattern to "capture" for re-use later (for example, to find duplicate words or letters)
  - They define an "alternation," covered next

Alternation

- [af] is a convenient way to say, "I'll accept a or f to match," but how to say, "match either 'black' or 'white?'"
- With alternation, the pipeline symbol "|" as in
- (black | white)
- Note:
  - [af] and (af) are essentially identical, but you can't put ranges inside groups, just classes – "(a-z)" is a literal pattern a, -, z
  - Either way, groups and custom classes create a "fork in the road" for the regex engine
- This offers us another answer to the color/colour problem:
- (colour|color)
- Be careful of the order of the options, though…



"More Specific to the Left" Example

Also, even if your options don't step on each other, you can make the regex faster by putting the most-likely options to the left

Capture Groups

- Parentheses can also surround a subset of a group that you want not just to match but pull out into regex variables
- They also show up in $matches or the Select-String variables
- Here, we just match "I saw a [whatever]," no capture:

```
PS C:\> $p="I saw a \w+"
PS C:\> "I saw a ball today" -match $p
True
PS C:\> $matches

Name                                   Value
----                                   -----
0                                      I saw a ball
```

Now Refine It with a Capture

Now just put parentheses around the word, and it captures it into $matches[1]

```
PS C:\>
PS C:\> $p="I saw a (\w+)"
PS C:\> "I saw a ball today" -match $p
True
PS C:\> $matches

Name                                   Value
----                                   -----
1                                      ball
0                                      I saw a ball
```

Naming Captured Groups

- The first captured group is sometimes referred to as $1 or \1
- The second is $2 or \2 and so on
- Note that $1 and $2 are *not* PowerShell variables; regex has used those names for decades but PoSH gets confused, so surround them with single quotes, like '$1'
- Let's see how to actually use this
- The classic example is "find any doubled words, like this:

```
PS C:\>
PS C:\> "I am am here." -match "(\w+) \1"
True
PS C:\>
```

Looking for Special Words

- Are there any words with the same letter three times in a row?
- This tells us:
- sls -path .\words.txt -Pattern "(\w)\1\1"

Anchors

Anchors Refine a Pattern

- When part of the pattern is "... But only at the end of the line" or something like that, an anchor helps
- ^ matches "start of line, so ^hawk doesn't match "He was a deficit hawk" but would match "Hawkman is a pretty lame DC character"
- (Yes, ^ means "set negation" also, but only in […] constructs)
- End of line anchor is $
- \b = "must begin on a word boundary"
- \B = "must *not* begin on a word boundary"
- \A = like ^ but only the beginning of the *first* line
- \Z = like $ but only at the end of the last line
- \z = like \Z but ignores carriage returns \r
- \G = must start immediately after the last match ended

Anchor Example

- Suppose I take my words.txt file and wonder if there's a word starting with w and has two r's in it
- So I fire up Select-String and give it the pattern w.*r.*r.*
- Ooops… "Answerer" comes up
- So the better pattern is ^w.*r.*r.*

Multiline Mode

- Suppose you feed regex a string with newline characters
- Normally ^ means, "beginning of input string" and $ refers to the end of the input string
- If you enable multiline mode, ^ also matches the beginning of each line, and $ also matches the newline characters
- Turn it on as a regex option in a .NET new-object
- Or use the "m" mode modifier in your pattern… prefix the pattern with "(?m)"

Thank You Very Much!

- I hope I inspired you to learn enough about regexes to put them to work for you
- Remember:
  - It's okay to cheat. There are lots of great examples on the web
  - Use an online regex tester
  - Select-String is the power tool for attacking folders full of files
  - It only LOOKS weird… it *is* useful
- Thank you for attending, please don't forget to do an evaluation
- Have a safe trip home!