21 Using regular expressions to parse text files

Regular expressions are one of those awkward topics. We often have students ask us to explain them, only to realize—halfway through the conversation—that they didn't need regular expressions at all. A *regex*, as a regular expression is sometimes known, is useful in text parsing, which is something you end up doing a lot of in UNIX and Linux operating systems. In PowerShell, you tend to do less text parsing—and you tend to need regexes less often. That said, we certainly know of times when, in PowerShell, you need to parse textual content such as a log file. That's how we cover regular expressions in this chapter: as a tool to parse text files.

Don't get us wrong: there's much more you can do with regular expressions, and we cover a few of those things at the end of this chapter. But to make sure you have a good expectation up front, let's be clear that we don't cover regular expressions comprehensively or exhaustively in this book. Regular expressions can get *incredibly* complicated. They're an entire technology unto themselves. We'll get you started, and try to do so in a way that's immediately applicable to many production environments, and then we'll give you some pointers for digging deeper on your own, if that's your need.

Our goal with this chapter is to introduce you to regex syntax in a simplified fashion and show you how PowerShell can use regular expressions. If you want to move on to more-complicated expressions on your own, you're welcome to, and you'll know how to use those within the shell.

21.1 The purpose of regular expressions

A regular expression is written in a specific language, and its purpose is to define a text pattern. For example, an IPv4 address consists of one to three digits, a period, one to three more digits, a period, and so forth. A regex can define that pattern,

although it would accept an invalid address like 211.193.299.299. That's the difference between recognizing a text pattern and checking for the validity of the data.

One of the biggest uses of regular expressions—and the use we cover in this chapter—is to detect specific text patterns within a larger text file, such as a log file. For example, you might write a regex to look for the specific text that represents an HTTP 500 error in a web server log file, or to look for email addresses in an SMTP server log file. In addition to detecting the text pattern, you might use the regex to capture the matched text, enabling you to extract those email addresses from the log file.

21.2 A regex syntax primer

The simplest regex is an exact string of text that you want to match. `Car`, for example, is technically a regex, and in PowerShell it'll match `CAR`, `car`, `Car`, `CaR`, and so on; PowerShell's default matching is case insensitive.

Certain characters, however, have special meaning within a regex, and they enable you to detect patterns of variable text. Here are some examples:

- `\w` matches "word characters," which means letters, numbers, and underscores, but no punctuation and no whitespace. The regex `\won` would match `Don`, `Ron`, and `ton`, with the `\w` standing in for any single letter, number, or underscore.
- `\W` matches the opposite of `\w` (so this is one example where PowerShell is sensitive to case), meaning it matches whitespace and punctuation—"nonword characters," in the parlance.
- `\d` matches any digit from `0` through `9` inclusive.
- `\D` matches any nondigit.
- `\s` matches any whitespace character, including a tab, space, or carriage return.
- `\S` matches any nonwhitespace character.
- `.` (a period) stands in for any single character.
- `[abcde]` matches any character in that set. The regex `c[aeiou]r` would match `car` and `cur`, but not `caun` or `coir`.
- `[a-z]` matches one or more characters in that range. You can specify multiple ranges as comma-separated lists, such as `[a-f,m-z]`.
- `[^abcde]` matches one or more characters that are not in that set, meaning the regex `d[^aeiou]` would match `dns` but not `don`.
- `?` follows another literal or special character and matches exactly one instance of that character. So, the regex `ca?r` would match `car` but would not match `coir`. It would also match `ca` because `?` can also match zero instances of the preceding character.
- `*` matches any number of instances of the preceding character. The regex `ca*r` would match both `cair` and `car`. It would also match `ca` because `*` also matches zero instances of the preceding character.
- `+` matches one or more instances of the preceding character. You'll see this used a lot with parentheses, which create a sort of subexpression. For example, the

regex `(ca)+r` would match `cacacacar` because it matches repeating instances of the `ca` subexpression.

- `\` (backslash) is the regex escape character. Use it in front of a character that normally has special meaning in the regex syntax, to make that character a literal. For example, the regex `\.` would match a literal period character, rather than allowing the period to stand in for any single character, as it normally does. To match a literal backslash, escape it with a backslash: `\\`.
- `{2}` matches exactly that many instances of the preceding character. For example, `\d{1}` matches exactly one digit. Use `{2,}` to match two or more, and use `{1,3}` to match at least one, but no more than three.
- `^` matches the beginning of the string. For example, the regex `c.r` would match `car` as well as `pteranocar`. But the regex `^c.r` would match `car` but would not match `pteranocar` because the `^` makes the matching occur at the beginning of the string. This is a different use of `^` than in the previous example, where it was used with square brackets, `[]`, to indicate a negative match.
- `$` matches the end of the string. For example, the regex `.icks` would match `hicks` and `sticks` (the match would technically be on `ticks` in that example), and would also match `Dickson`. But the regex `.icks$` would not match `Dickson` because the `$` indicates that the string should reach its end after the `s`.

There you have it—a whirlwind look at the basic regex syntax. As we wrote earlier, there's a lot more where that came from, but this is enough to get some basic work done. Let's look at some example regular expressions:

- `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}` matches the pattern of an IPv4 address, although it'll accept illegal data like `432.567.875.000`, as well as legal data like `192.169.15.12`.
- `\\\\\w+(\\w+)+` matches a Universal Naming Convention (UNC) path. All the backslashes make that regex hard to read—which is one reason it's important to test and tweak your regular expressions before you rely on them in a production task.
- `\w{1}\.\w+@company\.com` matches a specific type of email address: first initial, a period, last name, and then `@company.com`. For example, `sam.smith@company.com` would be a valid match. You do have to be a bit careful with these. For example, `Samuel.smith@company.com.org` or `Smith@company.com.net` would also be a valid match. The regex is fine with there being extra text before and after the matched portion. That's where the `^` and `$` anchors come into play in many situations.

NOTE You'll find more about basic regex syntax by running `help about_regular_expressions` in PowerShell. At the end of this chapter, we provide some additional resources for further exploration.

21.3 Using regex with -Match

PowerShell includes a comparison operator, `-Match`, and a case-sensitive cousin, `-CMatch`, that work with regular expressions. Here are some examples:

```
PS C:\> "car" -match "c[aeiou]r"
True
PS C:\> "caaar" -match "c[aeiou]r"
False
PS C:\> "caaar" -match "c[aeiou]+r"
True
PS C:\> "cjinr" -match "c[aeiou]+r"
False
PS C:\> "cear" -match "c[aeiou]r"
False
```

Although it has many uses, we're primarily going to rely on `-Match` to test regular expressions and make sure they're working properly. As you can see, its left-hand operand is whatever string you're testing, and the right-hand operand is the regular expression. If there's a match, it outputs `True`; if not, you get `False`.

> **TRY IT NOW** This is a good time to take a break from reading and try using the `-Match` operator. Run through some of the examples we just mentioned, and make sure you're comfortable using the `-Match` operator in the shell.

21.4 Using regex with Select-String

Now we reach the real meat of this chapter. We're going to use some web server log files as examples, because they're exactly the kind of pure-text file that a regex is designed to deal with. It'd be nice if we could read these logs into PowerShell in a more object-oriented fashion but, well, we can't. So a regex it is.

Let's start by scanning through the log files to look for any 40*x* errors. These are often File Not Found and similar, and we want to be able to generate a report of the bad files for our organization's web developers. The log files contain a single line for each HTTP request, and each line is broken into space-delimited fields. We have some files that contain 401 and so forth as part of their filename—for example, error401.html—and we don't want those to be included in our results. We specify a regex such as `\s40[0-9]\s` because that specifies a space on either side of the 40*x* error code. It should find all errors from 400 through 409 inclusive. Here's our command:

```
PS C:\logfiles> get-childitem -filter *.log -recurse |
 select-string -pattern "\s40[0-9]\s" |
 format-table Filename,LineNumber,Line -wrap
```

Notice that we change to the C:\LogFiles directory to run this command. We start by asking PowerShell to get all files matching the *.log filename pattern and to recurse subdirectories. That ensures that all of our log files are included in the output. Then we use `Select-String` and give it our regex as a pattern. The result of the command

is a `MatchInfo` object; we use `Format-Table` to create a display that includes the filename, the line number, and the line of text that contains our match. This can be easily redirected to a file and given to our web developers.

> **NOTE** You may have noticed that we used `Format-Table`. We did this for two reasons. The first is because we wanted to wrap the text on the screen, and the second is because we are simply making the screen look cleaner, and we aren't outputting any of the information.

Next, we want to scan the files for all access by Gecko-based web browsers. Our developers tell us they've been having some problems with customers accessing the sites using those browsers, and they want to see which files in particular are being requested. They think they've narrowed the problem down to browsers running under Windows NT 10.0, meaning we're looking for user-agent strings that look something like this:

```
(Windows+NT+10.0;+WOW64;+rv:11.0)+Gecko
```

Our developers have stressed that the 64-bit thing isn't specific, so they don't want the log results limited to just `WOW64` user-agent strings. We come up with this regex: `10\.0;[\w\W]+\+Gecko`. Let's break that down:

- `10\.0;`—This is 10.0. Notice that we escaped the period to make it a literal character rather than the single-character wildcard that a period normally indicates.
- `[\w\W]+`—This is one or more word or nonword characters (in other words, anything).
- `\+Gecko`—This is a literal +, then *Gecko*.

Here's the command to find matching lines from the log files, along with the first couple of lines of output:

```
PS C:\logfiles> get-childitem -filter *.log -recurse |
Select-string -pattern "10\.0;[\w\W]+\+Gecko"
W3SVC1\u_ex120420.log:14:2012-04-20 21:45:04 10.211.55.30 GET
     /MyApp1/Testpage.asp - 80 - 10.211.55.29
     Mozilla/5.0+(Windows+NT+10.0;+WOW64;+rv:11.0)+Gecko/20100101+Firefox/11.
     0 200 0 0 1125
W3SVC1\u_ex120420.log:15:2012-04-20 21:45:04 10.211.55.30 GET /TestPage.asp -
     80 - 10.211.55.29
     Mozilla/5.0+(Windows+NT+10.0;+WOW64;+rv:11.0)+Gecko/20100101+Firefox/11.
     0 200 0 0 1 109
```

We left the output in its default format this time, rather than sending it to a format cmdlet.

As a final example, let's turn from IIS log files to the Windows Security log. Event log entries include a `Message` property, which contains detailed information about the event. Unfortunately, this information is formatted for easy human reading, not for easy computer-based parsing. We'd like to look for all events with ID 4624, which indicates

an account logon (that number may differ in different versions of Windows; our example is from Windows Server 2008 R2). But we want to see only those events related to logons for accounts starting with WIN, which relates to computer accounts in our domain and whose account names end in TM20$ through TM40$, which are the specific computers we're interested in. A regex for this might look something like `WIN[\W\w]+TM[234][0-9]\$`. Notice how we need to escape the final dollar sign so that it isn't interpreted as an end-of-string anchor. We need to include `[\W\w]` (nonword and word characters) because it's possible for our account names to include a hyphen, which wouldn't match the `\w` word character class. Here's our command:

```
PS C:\> get-eventlog -LogName security | where { $_.eventid -eq 4624 } |
select -ExpandProperty message | select-string -pattern
"WIN[\W\w]+TM[234][0-9]\$"
```

We start by using `Where-Object` to keep only events with ID 4624. We then expand the contents of the `Message` property into a plain string and pipe it to `Select-String`. Note that this will output the matching message text; if our goal was to output the entire matching event, we would have taken a different approach:

```
PS C:\> get-eventlog -LogName security | where { $_.eventid -eq 4624 -and
➥ $_.message -match "WIN[\W\w]+TM[234][0-9]\$" }
```

Here, rather than outputting the contents of the `Message` property, we simply look for records where the `Message` property contains text matching our regex—and then output the entire event object. It's all about what you're after in terms of output.

21.5 Lab

NOTE For this lab, you need any computer running PowerShell v7 or later.

Make no mistake about it, regular expressions can make your head spin, so don't try to create complex regexes right off the bat—start simple. Here are a few exercises to ease you into it. Use regular expressions and operators to complete the following:

1. Get all files in your Windows or /usr directory that have a two-digit number as part of the name.
2. Find all modules loaded on your computer that are from Microsoft, and display the name, version number, author, and company name. (Hint: Pipe `Get-module` to `Get-Member` to discover property names.)
3. In the Windows Update log, you want to display only the lines where the agent began installing files. You may need to open the file in Notepad to figure out what string you need to select. You may need to run `Get-WindowsUpdateLog`, and the corresponding log will be placed on your desktop.

   For Linux, find your history log and display the lines where you installed packages.

4 Using the `Get-DNSClientCache` cmdlet, display all listings in which the `Data` property is an IPv4 address.
5 If you are on a Linux (or Windows) machine, find the lines of the HOSTS file that contain IPV4 addresses.

21.6 Lab answers

```
1 Get-ChildItem c:\windows | where {$_.name -match "\d{2}"}
  -  Get-ChildItem /usr | where {$_.name -match "\d{2}"}
2 get-module | where {$_.companyname -match "^Microsoft"} |
  Select Name,Version,Author,Company
3 get-content C:\Windows\WindowsUpdate.log |
  Select-string "[\w+\W+]Installing Update"
  Get-content ./apt/history.log | select-string "[\w+\W+]Installing"
```

4 You could get by with a pattern that starts with one to three numbers followed by a literal period, like this:

```
get-dnsclientcache | where { $_.data -match "^\d{1,3}\."}
```

Or you could match an entire IPv4 address string:

```
get-dnsclientcache | where
{ $_.data -match "^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"}
```

```
5 gc /etc/hosts | where {$_ -match "^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"}
```

21.7 Further exploration

You'll find regular expressions used in other places in PowerShell, and many of them involve shell elements that we don't cover in this book. Here are some examples:

- The `Switch` scripting construct includes a parameter that lets it compare a value to one or more regular expressions.
- Advanced scripts and functions (script cmdlets) can utilize a regular expression–based input-validation tool to help prevent invalid parameter values.
- The `-Match` operator (which we covered briefly in this chapter) tests for string matches against a regular expression, and—something we didn't share earlier—captures matched strings to an automatic `$matches` collection.

PowerShell utilizes industry-standard regex syntax, and if you're interested in learning more, we recommend *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly, 2006). A gazillion other regex books are out there, some of which are specific to Windows and .NET (and thus PowerShell), some of which focus on building a regex for specific situations, and so forth. Browse your favorite online bookstore and see if any books look appealing to you and your specific needs.

We also use a free online regex repository, http://RegExLib.com, which has numerous regex examples for a variety of purposes (phone numbers, email addresses, IP addresses, you name it). We've also found ourselves using http://RegExTester.com, a website that lets you interactively test regular expressions to get them dialed in exactly the way you need.