Chapter 8. Coding in Regex

The patterning syntax known as *regular expressions* or *regex* is as notorious for its alleged steep learning curve as for its confirmed usefulness. Nobody seriously questions the *efficacy* of this set of scripting rules, but the efficiency of writing regular expressions is sometimes unclear. Knowing when they're essential takes some familiarity, so we'll consider a few use=case types.

Regular expressions can be used used not only to scan for predefined patterns in arbitrary text; expressions can extract contextualized data from within those matched patterns, passing it to an application for further manipulation and storage.

The concept of regular expressions is notoriously difficult to learn. I admit I didn't appreciate their potential for the first many years I begrudgingly used them to solve otherwise impossible data conversion and other IT tasks. Appreciating regex can make a world of difference to learning regex.

If you have never or barely used regex, perhaps it is about to become your first coding language. If you've used it before but never taken the time to learn its full potential, this chapter should be helpful as well.

Seasoned regex coders can keep me honest. This book is open source; help me bring the power of this intimidating but mercifully straightforward scripting syntax to struggling documentarians everywhere.

What-ex?

I pronounce the shortened term *REJ-ex*, though I believe I've heard more developers use *REGG-ex*. Or maybe it just strikes me more starkly when I hear it pronounced differently than I say it.

What is Regex?

A regular expression is a way of describing a pattern using common characters in order to search for the described pattern in another body of characters. The target can be pretty much any form of text, such as a conventional document or a flattened dataset. A regex parsing engine uses these expressions to evaluate target texts for matches.

Put differently by the regex-focused site Rexegg, "A regex is a text string that describes a pattern that a regex engine uses in order to find text (or positions) in a body of text, typically for the purposes of validating, finding, replacing, or splitting."

At its simplest application, think of regex as a much smarter version of the `*` or `%` wildcard symbols you've probably used in other contexts. Where a conventional wildcard finds *any and all text*, regex can find shockingly detailed patterns without false positives.

The string `codewriting.org` is matched by the following regex patterns:

- `[a-zA-Z0-9][a-zA-Z0-9\-]{1,61}[a-zA-Z0-9]\.[a-zA-Z]{2,}`
- `.*@\.[a-z]+`
- `codewriting\.org`

You'll note that all of these are more complicated than the non-regex search term:

- `codewriting.org`

Probably in most cases of a find-and-replace task, I don't get to use regex.

But what if we needed to find all instances of `http://` protocol indicator for a `codewriting.org` URL in our docs and replace `http:` with `https:`. We don't want this for every instance of `http:` in the document, but we also cannot count on the domain string to always appear intact, as `http://codewriting.org`, which we could easily replace with `https://codewriting.org`. Making matters more complicated, numerous instances of a `codewriting.org` URL appear with a hostname or "subdomain", such as `docs.codewriting.org`. To make sure we caught them all manually, we'd first have to create a list, and then iterate through it one by one replacing the whole string.

Here's the simple regex pattern that will save us all this trouble:

```
http\:\/\/([a-z]+\.)?codewriting\.org
```

This regex would efficiently match the following strings:

- `http://codewriting.org`
- `http://git.codewriting.org`
- `http://staging.codewriting.org`

The single set of parentheses in the regex will capture any "group" it matches and store it as a variable we can later reference to reinsert the captured string. The `$1` in the following replace string represents the first (and only) such group captured in each of our examples.

```
https://$1codewriting.org
```

The `$1` will insert whatever was captured for each instance: either an empty string, `git.`, or `staging.`. Therefore, one find/replace procedure will handle the whole fix.

The above pattern assumes I know that all subdomains are lowercase letters (`[a-z]+`), or it would get more complicated (`[a-zA-Z0-9][a-zA-Z0-9\-]+`). This pattern indicates a single alphanumeric character followed by any combination of alphanumeric or the `-` symbol, the only allowable format for hostnames.

Let's say for some reason we also have various top-level domains in addition to different subdomains. Maybe we also bought `codewriting.net` and have different apps running on different hosts, and our docs need to reflect the distinction.

We need to upgrade our pattern to catch these new variations.

```
http\:\/\/([a-zA-Z0-9][a-zA-Z0-9\-]+\.)?codewriting\.([a-zA-Z]{2,})
```

Now we can match several new patterns that we were unable to previously, along with a second group for reflecting the top-level domain.

- http://codewriting.org
- http://git.codewriting.org
- http://staging.codewriting.org
- http://shop.codewriting.com
- http://members.codewriting.net
- http://dating.codewriting.net
- http://codewriting.ca
- http://1staging.codewriting.org

Our replace string is still pretty simple, with the second capture group token added in place of the definite `org`:

```
https://$1codewriting.$2
```

Why (and When) to Use Regex

Make no mistake about it, regular expressions are a labor-saving tool. They are for automating routines. You might think that means they either save you time or they're not worth bothering with, but some people might disagree.

When it comes to these everyday use cases, it's not always clear up front that employing regex will actually save you time. It will almost certainly be *more fun* to spend even an hour trying to write just the perfect expression that catches everything you want and nothing more. Remember, the alternative is performing a routine task with numerous slight variations, change after manual change, without introducing any errors.

> Of course errors can be introduced by improperly written regular expressions, but these mistakes tend to be consistent: they can be tested for and fixed using — you guessed it — regex. Errors introduced manually are more likely to vary. You may someday discover having mistyped `"` as `'`, but elsewhere another typo: `''`.

Still, that hour-long regex challenge will only save you time if manually finding and replacing every instance would take more than an hour to accomplish, with the added fact that you might introduce mistakes and not catch them. (You'd really want to use regular expressions to catch those mistakes, which begs the question.)

What's more, as with pretty much all open-ended task work, estimating the difficulty of writing a regular expression to solve a one-off problem can be challenging. I readily admit I have on numerous occasions spent more time struggling with a regular expression than I would have saved if I'd perfectly coded the pattern on the first try.

Nevertheless, I think overall it has saved me hundreds of hours on day-to-day one-off tasks, including dataset migrations and HTML refactoring. So it was always useful for big, onerous tasks, and an occasional fun puzzle work would throw my way.

But when regex is used to automate a routine task you'd otherwise have to start and repeat on a regular basis, the value can become immeasurably great. We'll explore such a case in Part Four.

Suffice it to say, becoming wise about the up-front burdens imposed by regex is important, but any serious documentarian is bound to encounter cases where the right regex will save critical hours, or at least tedious hours. So let's demystify this powerful tool before you're expected to put it to use on a real-world task.

Digging Into Regex

Detailed information about new features often comes in to me from engineers one of two ways. Either they refer me to the **source code** and expect me to extract all necessary details for it, or they give me a **text document** of some kind containing lots of loosely but consistently formatted clusters of data. My job is to translate some portion of those documents into user-facing information. In both cases, I often turn straight to regex.

```
"Volume": Modulates how loudly the music will play. Optional. Defaults to 5.

"Bass": Modulates the deeper frequencies of sound. Optional. Defaults to 7.

"Treble": Modulates the higher frequencies of sound. Optional. Defaults to 7.

"Booster": Hypes the whole system up. Required.
```

Let's assume a we're imagining the following modest target format.

Target output for regex transformation

```
Volume::
Modulates how loudly the music will play
+
[horizontal]
Default::: `5`
+
Required::: No

Bass::
Modulates the deeper frequencies of sound
+
[horizontal]
Default::: `7`
+
Required::: No

Treble::
Modulates the higher frequencies of sound
+
[horizontal]
Default::: `7`
+
Required::: No

Booster::
Hypes the whole system up
+
[horizontal]
Default:::
+
Required::: *Yes*
```

Just so you catch my aim, the unstyled Asciidoctor output of this AsciiDoc source is pretty basic, but good enough for our example. We'll loop results like the following.

Volume

Modulates how loudly the music will play

Default

`5`

Required

Optional.

Getting back to the task at hand, we're essentially just trying to rearrange some things. But those basic changes render standard find and replace nearly useless.

Quite usefully, a regex pattern can store portions of matches as variables for insertion during a subsequent procedure. Wrapping pattern groups in parenthesis indicates that we want to store the captured content as a variable. By default, these groups are tokenized numerically and can be expressed later as `$1`, `$2`, and so forth.

Here is an appropriate regex pattern to flexibly match these entries, including discrete portions of the content, which we want to carry over in a new arrangement.

Regex matching string with capture groups

```
^\"(.*)\"\:\s(.*)\.\s(Optional\.|Required\.)(?:\sDefaults to )?(.*)\.?$
```

This probably looks both sloppy and intimidating. I still find these big patterns a bit challenging to read, let alone write, but regex isn't that hard to get comfortable with. I can't say I'm not enjoying the shock value, though.

We are attempting to capture four groups wrapped in `( )` marks. Meanwhile, we're ignoring a fifth group (the one with the `Defaults to` string in it), as we have no use for that text.

Once you've captured your groups, writing the parsing template is relatively straightforward. Simply use the `$n` token, where `n` is the capture order slot for that variable. This way they can be reproduced out of order.

Regex replace template for find/replace operation

```
$1::\n$2\n+\n[horizontal]\nDefault::: `$4`\n+\nRequired::: $3\n
```

The `\n` token denotes a newline marker, of course you see four numbered group tokens we discussed a moment ago. The rest of the markup you'll recognize from our AsciiDoc source example above. Let's look at what this find/replace operation produced — you'll recommend most of the other replacement template in here.

Regex find and replace result

```
Volume::
Modulates how loudly the music will play
+
[horizontal]
Default::: `5`
+
Required::: Optional.


Bass::
Modulates the deeper frequencies of sound
+
[horizontal]
Default::: `7`
+
Required::: Optional.


Treble::
Modulates the higher frequencies of sound
+
[horizontal]
Default::: `7`
+
Required::: Optional.


Booster::
Hypes the whole system up
+
[horizontal]
Default::: ``
+
Required::: Required.
```

This is not perfect. We'll still need to do another round or two of changes against this result to get it how we need it to look. But you'll see now why I captured the `.` character in `Optional.` and `Required.`. Our next move will be to run a simple (non-regex) find and replace procedure on the string `Required.`, which will miss the `Required` in `Required:::`. We want to replace `Required.` with **`Yes`**.

Finally, we can replace `Optional.` with `No`. Now our document matches our target format precisely.

Truthfully, this entire transformation could have been performed with one replace statement using conditionals, but that is way beyond the scope of this primer.

What did we save ourselves? This obviously depends on how hard it was to create the patterns. As soon as you get them right, you're pretty much done. And we did not have to manually remove quotation marks, insert carriage returns, numerous newlines, that repetitive `+` symbol, and all those infernal colons. If this example were about three times the size, I would certainly have opted for regex to save

time.

As a bonus, we got to solve a cool little puzzle! Seriously what more could a technical documentarian ask for?

Proficient use of regex does not require memorization of all regex patterns — they can always be looked up. Besides, you'll use the same handful of character combos most of the time, adapting common patterns to your specific content or data scenario. Getting the concepts right early on is far more important than memorizing symbols, so let's start there.

Regex (as) Coding

So far we've been sticking patterns into Find fields and onto simple configuration files — not exactly delving deep into agile programming. And truth be told, while regex patterns certainly qualify as computer instructions, and thus are programs, you're not quite writing software until you save your patterns as a script that will be run and rerun.

Unlike true programming languages, you won't be writing full applications in regex. But you may well have opportunities to write an expression that will be stored and processed, maybe even countless times a minute. We'll see some powerful cases of this in Part 4, but for now let's take on a simple one I think many of my readers have encountered or are likely to.

I'm talking about the `.htaccess` file lots of us have used when we've needed to deal with website managing URLs on Apache-based webservers. This file, stored in the site's root directory, is checked every time a request comes to the site. If aspects of the request matched patterns we designated in that file, the server would be instructed to behave as we saw fit. A common use is to force all browsers to use HTTPS protocol, or when URL paths change and we want to redirect browsers and search engines to the new address.

Example — .htaccess rewrite

```
RewriteCond %{SERVER_PORT} 80
RewriteRule ^(.*)$ https://example.com/$1
```

The first line tests the server port. If the requested server port is `80`, the connection is insecure.

The second line commands resetting the request URL by capturing everything in the URL path after the home directory and then forcing that onto a new, secure URL. So if the request is for the URL `http://example.com/some-path/index.html`, and the `.htaccess` file is in the domain root directory (`/`), the above `RewriteRule` captures all text (`.*` matches *any content*) from the beginning of the string (`^`) to the end of the string (`$`). The parentheses indicate content to capture and store in a variable — in this case, *everything*.

In our sample URL, this will have matched `some-path/index.html`, which we now want the webserver to append to anther call to our domain, this time using HTTPS protocol. The next string is our explicit root

domain followed by $1, the token that indicates we want to insert the our first captured string (in this case, our only captured string) right in that spot. Our URL becomes `https://example.com/some-path/index.html`.

Let's try another common case.

Example — .htaccess rewrite

```
RewriteRule ^index\.php\?page\=(.*)$ $1.html [L,R=301]
```

Here we have upgraded our old PHP-based CMS to an awesome new static site. The pages have all been replaced, but of course the URLs to our support docs are littering StackExchange and hundreds of hacker blogs. We want all those links to find what was intended, and we want search engines to know the change is permanent. All of which we can do in one line. This code captures the value of a query parameter from our old site and makes it the base filename of the new static page we've replaced it with.