CHAPTER 1

An Introduction to Regular Expressions

I still remember my doomed encounters with regular expressions back when I tried to learn them. In fact, I took pride in not using regular expressions. I always found a long workaround or a code snippet. I projected and blamed my own lack of expertise on the hard readability of regular expressions. This process continued until I was ready to face the truth: regular expressions are powerful, and they can save you a lot of time and headache.

Fast-forward a couple of years. People I worked with encountered the same problems. Some knew regular expressions, and others hated them. Among the haters of regular expressions, it was quite common that they actually liked the syntax and grammar of their first programming language. Some developers even took courses on formal languages. Therefore, I made it my priority to show everyone a path toward their disowned knowledge to master regular expressions.

Why Are Regular Expressions Important?

In today's world, we have to deal with processing a lot of data. Accessing data is not the main problem. Filtering data is. Regular expressions provide you with one type of filter that you can use to extract relevant data from

Z. Nagy, *Regex Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-3876-9_1

the big chunks of data available to you. For instance, suppose you have an XML file containing 4GB of data on movies. Regular expressions make it possible to query this XML text so that you can find all movies that were filmed in Budapest in 2016, for instance.

Regular expressions are a must-have for software developers.

In front-end development, we often validate input using regular expressions. Many small features are also easier with regular expressions, such as splitting strings, parsing input, and matching patterns.

When writing backend code, especially in the world of data science, we often search, replace, and process data using regular expressions. In IT infrastructure, regular expressions have many use cases. VIM and EMACS also come with regex support for finding commands, as well as editing text files.

Regular expressions are everywhere. These skills will come handy for you in your IT engineering career.

What Are Regular Expressions?

Regular expressions, or *regexes*, come from the theory of formal languages. In theory, a *regex* is a finite character sequence defining a search pattern. We often use these search patterns to

- Test whether a string matches a search expression
- Find some characters in a string
- Replace substrings in a string matching a regex
- Process and format user input
- Extract information from server logs, configuration files, and text files
- Validate input in web applications and in the terminal

A typical regular expression task is *matching*. I will now use JavaScript to show you how to test-drive regular expressions because almost everyone has access to a browser. In the browser, you have to open the Developer Tools. In Google Chrome, you can do this by right-clicking a web site and selecting Inspect. Inside the Developer Tools, select the Console tab to enter and evaluate your JavaScript expressions.

Suppose there is a JavaScript regular expression `/re/`. This expression looks for a pattern inside a string, where there is an `r` character, followed by an `e` character. For the sake of simplicity, suppose our strings are case sensitive.

```
const s1 = 'Regex';
const s2 = 'regular expression';
```

In JavaScript, strings have a `match` method. This method expects a regular expression and returns some data on the first match.

```
> s1.match( /re/ )
null
```

```
> s2.match( /re/ )
["re", index: 0, input: "regular expression"]
```

A regular expression is an expression written inside two slash (`/`) characters. The expression `/re/` searches for an `r` character followed by an `e` character.

As an analogy, imagine that you loaded a web site in the browser, pressed Ctrl+F or Cmd+F to find a substring inside the web site on the screen, and started typing `re`. The regular expression `/re/` does the same thing inside the specified string, except that the results are case sensitive.

Notice that `'Regex'` does not contain the substring `'re'`. Therefore, there are no matches.

The string `'regular expression'` contains the substring `'re'` twice: once at position 0 and once at position 11. For the sake of determining the match, the JavaScript regular expression engine returns only the first match at index 0 and terminates.

JavaScript allows you to turn the syntax around by testing the regular expression.

```
> /re/.test( s1 )
false

> /re/.test( s2 )
true
```

The return value is a simple Boolean. Most of the time, you do not need anything more, so testing the regular expression is sufficient.

Each programming language has different syntax for built-in regex support. You can either learn them or generate the corresponding regex code using an online generator such as `https://regex101.com/`.[1]

Frustrations with Regular Expressions Arise from Lack of Taking Action

According to many software developers, regular expressions are

- Hard to understand
- Hard to write
- Hard to modify
- Hard to test
- Hard to debug

[1] `https://regex101.com/`

As I mentioned in the introduction of this chapter, lack of understanding often comes with blame. We tend to blame regular expressions for these five problems.

To figure out why this blaming exists, let's discover the journey of a regular developer, no pun intended, with regexes. Many of us default to this journey of discovery when it comes to playing around with something we don't know well. With regular expressions, the task seems too easy: we just have to create a short expression, right? Well, oftentimes, this point of view is very wrong.

Trial and error oftentimes takes more time than getting the pain handled and getting the lack of knowledge cured. Yet, most developers work with trial and error over and over again. After all, why bother learning the complex mechanics of regular expressions if you could simply copy and paste a small snippet? Learning the ins and outs of regular expressions seems to be too hard at first glance anyway.

Therefore, my mission is to show you that

- Learning regular expressions is a lot easier than you thought
- Knowing regular expressions is fun
- Knowing regular expressions is beneficial in many areas of your software developer career

You can still easily master regular expressions to the extent that they will do exactly what you intended them to do. This mastery comes from understanding the right theory and getting a lot of practice.

Regular Expressions Are Imperative

Regular expressions are widely misunderstood. Whenever you hear that regular expressions are *declarative*, run from that tutorial or blog as far as you can. Regexes are an imperative language. If you want to understand regexes as declarative, chances are you will fail.

By definition, regexes specify a search pattern. Although this is a true statement, it is easy to misinterpret it because you are not specifying a declarative structure. In the real world, you specify a sequence of instructions acting like a function in an imperative programming language. You use commands and loops, you pass arguments to your regex, you may pass arguments around inside your regex, you return a result, and you may even cause side effects.

Learning regular expressions as an imperative language comes with a big advantage. If you have dealt with at least one programming language in your life, chances are, you know almost everything to understand regular expressions. You are just not yet proficient in the regex syntax. As soon as you familiarize yourself with this weird language, everything will fall into place.

The Language Family of Regular Expressions

When I talk about regular expressions, in practice I mean a family of different dialects. Similarly to genetics, regular expressions keep evolving, and new mutations surface on a regular basis. Although the principles stay the same in most languages, every single dialect brings something different.

Standardization of regular expressions began with *BRE* (Basic Regular Expressions) inside the POSIX standard 1003.2. This standard is used in the editors `ed` and `sed`, as well as in the `grep` command.

The first major evolution of regular expressions came with the *ERE* (Extended Regular Expressions) syntax. This syntax is used in, for example, `egrep` and `notepad++`.

For completeness, I will also mention the *SRE* (Simple Regular Expressions) dialect, which has been deprecated in favor of BRE.

Some editors such as EMACS and VIM have their own dialects. In the case of VIM, the dialect can be customized with flags, and this technique provides even more variations. These dialects are built on top of ERE.

The regular expressions used in most programming languages are based on the *PCRE* (Perl Compatible Regular Expressions) dialect. Each programming language has its own abbreviations and differences. These programming languages include PHP, JavaScript, Java, C#, C++, Python, R, Perl up to version 5, and more.

To make matters more complex, Perl 6 comes with a completely different set of rules for regular expressions. The Perl 6 syntax is often easier to read, but in exchange, you have to learn a different language.

As an example, let's write a regex for matching strings that contain at least one non-numeric character.

Dialect	**Expression**
BRE, ERE, EMACS, VIM, PCRE	`/[^0123456789]/`
Perl 6	`/<-[0123456789]>/`

As you can see, in this specific example, all dialects but Perl 6 look identical. Without getting lost in the details too much, let's understand what this expression means in BRE, ERE, EMACS, VIM, and PCRE.

- `[0123456789]` matches one single character from the character set of digits.
- ^ inside an enumeration negates the character list. This means `[^0123456789]` matches any character that's not a digit.
- As the regular expression may match any character of the test string, a match is determined as soon as you find at least one character in the test string that's not a digit. Therefore, `123.45` matches the regular expression, while `000` does not.

The Perl 6 syntax works in the same way; the syntax is just different.

Let's now write a regular expression that matches the `0`, `1`, or `2` character, using the `or` operator of regular expressions.

```
BRE:                or operator is not supported
ERE,PCRE,Perl 6:    /0|1|2/
EMACS,VIM:          /0\|1\|2/
```

An equivalent BRE expression would be `/[012]/`, using a character set. You will study character sets in detail in Chapter 6.

As studying six groups and many different variations would take a long time, I highly recommend you stick to one specific dialect and practice your skills focusing on the one and only dialect you use in practice. You can come back to study other dialects later. When it comes to the PCRE dialect, different languages give you different variations. I have personally found it beneficial to build and execute regular expressions in multiple programming languages. This way, I had an easier time solidifying my regex knowledge from different angles.

Summary

In this chapter, I defined a regular expression as a finite character sequence defining a search pattern. As an example, you saw a test execution of a simple JavaScript regular expression in the console. Although the tested regular expression was simple, oftentimes people have a hard time constructing and understanding regular expressions. This is because regular expressions represent a compact imperative language, and therefore, they are often not intuitive to understand. To make matters more complicated, regular expressions consist of multiple languages, which means that the JavaScript syntax is completely different than the syntax used in Perl 6.