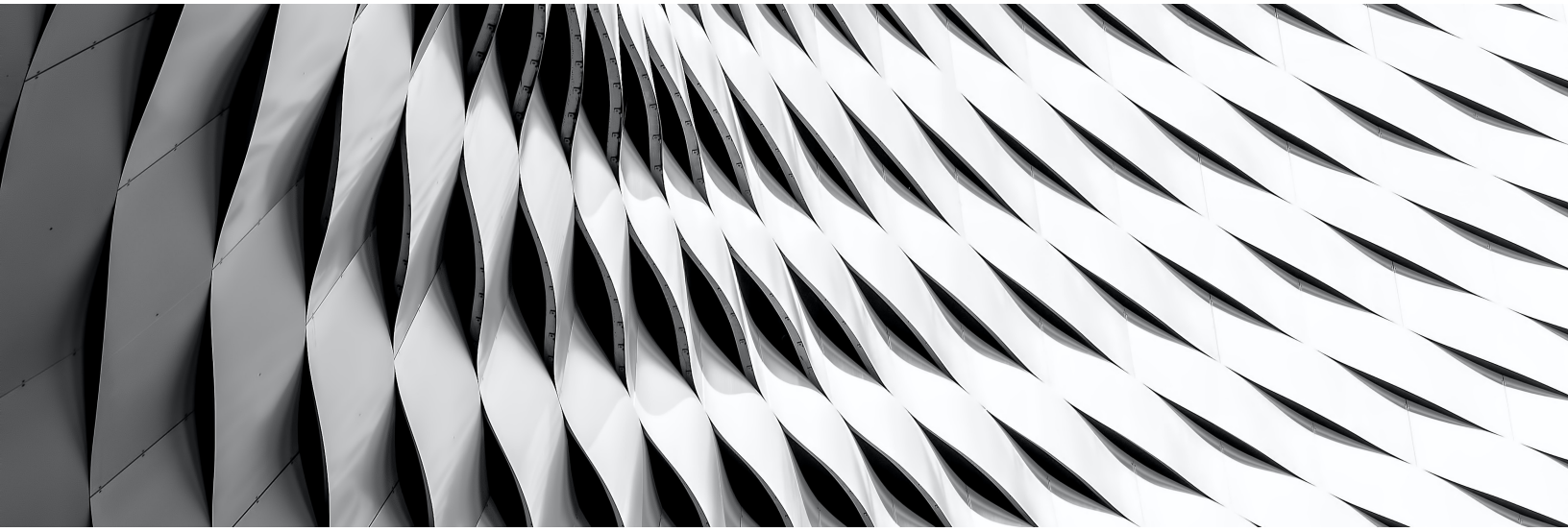


Intro to Regular Expressions

The Basics

Regular Expressions, or Regex for short, are special text strings which are used to find and match patterns within text. Regex syntax is almost uniform across all programming languages, so once you learn the basics you can utilize it in any language (with some minor tweaks).

A Simple Problem

Imagine we needed to write some code which could determine whether a given string contained a 24 hour time in it. Something like:

> "Let's meet at 11:45"

or

> "I ran a 03:30 marathon"

How would we go about checking for the presence of a time in the string and extracting it? We could do it manually, using loops and a bunch of complex logic, but it's not that easy! It's not as simple for checking for a pattern of two numbers and then a colon and then two more numbers. We need to make sure the times are actually valid rather than any two digit numbers. What if we wanted to also allow the option to add in seconds?

> "I ran a 03:30:19 marathon"

This could be really really frustrating to write, but fortunately we have Regular Expressions!

When are regular expressions useful?

Regular Expressions are commonly used for validating data, performing advanced searches, replacing/updating text, and a whole bunch more. Let's focus on validating user data for now. Regex are incredibly useful when you need to verify that information is formatted correctly. Here are a few of the extremely common formats we might need data to follow:

- Email addresses
- Phone Numbers - (415) 222-3455 or 415-222-3455 or 415 222 3455 or 4152223455
- Strong passwords - at least 6 chars, 1 or more uppercase letter, 1 or more lowercase letter, 1 or more number, and 1 or special character
- Usernames - 3 to 16 characters, must consist only of numbers, letters, and underscores

Trying to verify these patterns without the use of Regular Expressions is a nightmare 😭. It's possible, but it means writing tons of loops and complex logic.

Testing and Playing with Regular Expressions

> 💡 I highly recommend using an online editor like https://regexr.com to test out your Regular Expressions. I'll be using it throughout this video.

RegEx Syntax

Standard RegEx Format

When we write Regular Expressions in most programming languages, we wrap them in forward slashes:

```
/ your regex goes here /
```

Those forward slashes aren't actually matching anything; they simple denote the start and end of a RegEx much like the quotes we put around a string: `"hello world"`

⚠️ Note that regexr.com already includes the forward slashes for you, as you can see in the image below. **To make my code easier to copy & paste, I will omit the forward slashes from my examples**.

Our First Simple Regex

Let's start nice and simple. Take a look at this basic Regular Expression:

> z

Yes, the letter z. This RegEx will match any lowercase z characters in your test string. For example:

> Zebras say "zzz" when they are dazed

/tog

Our RegEx finds 4 matches in the above string. Notice the uppercase Z is not matched. We could try this almost any character, though some characters like `.` (period) and `$` have their own special meanings. If we need to actually match one of those characters, we can use an escape character which we'll see in a bit.

Regex can do a lot more than matching simple characters within a string. The first group of special characters we'll look at are the character codes:

Character Codes

📝 **Exercise**

Given this Regular Expression:

> \d[13579]

How many matches are found in this string:

> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Aa Regex	Meaning
[abc]	Matches any of the include characters
[^abc]	Matches any characters NO specified in brackets
[a-z]	Matches any characters within specific range
.	Matches any character except linebreaks
\w	Matches any word characte equivalent to [A-Za-z0-9_]
\d	Matches numeric digits
\s	Matches any whitespace (spaces, tabs, & linebreaks

▶ View The Answer

Next, up we have a group of special characters which allow us to specify the quantity of tokens within a sequence:

Quantifiers & More

📝 **Exercise**

Given the following Regular Expression:

> o?k+

How many matches are found in this string:

> ok kk k okkkk

▶ View The Answer

Anchors

Regex	Meaning
+	Matches 1 or more of preceding token
*	Matches zero or more of th previous token
{3}	Matches specific quantity o previous token
{1,3}	Matches quantity within specified range. Between 1 and 3 in this case.
{4,}	Matches the specified quantity or greater
?	Matches 0 or 1 of previous token. Basically it makes something optional
|	Similar to boolean OR. Matches whatever is to the left or right of the | charact

Regex	Meaning
^	Matches the beginning of a string
$	Matches the end of a string

> ^[A-Z]{2}\s\d{5}

Which of the following strings contain a valid match?

1. San Francisco CA 94103
2. CO81611
3. NY 10003
4. MT 59868 CA 92657

▶ **View The Answer**

Escaped Characters

As we've seen, many characters have special meanings in a Regular Expression. If we want to match characters like `+` or `?`, we need to first escape them using `\`

For example, `\+` will match a literal `+` character:

```
\d\+\d=\d would match: 1+1=2 6+3=9
```

This is the full list of characters you need to escape in order to match literally:

> `+ * ? ^ $ \ . [ ] { } ( ) | /`

📝 Another Exercise!

Write a RegEx that matches an exponential expression like: 4^2, 12^5, or 3^1122. Make sure it also works with negative numbers like: 5^-3 or -15^6.

▶ View The Answer

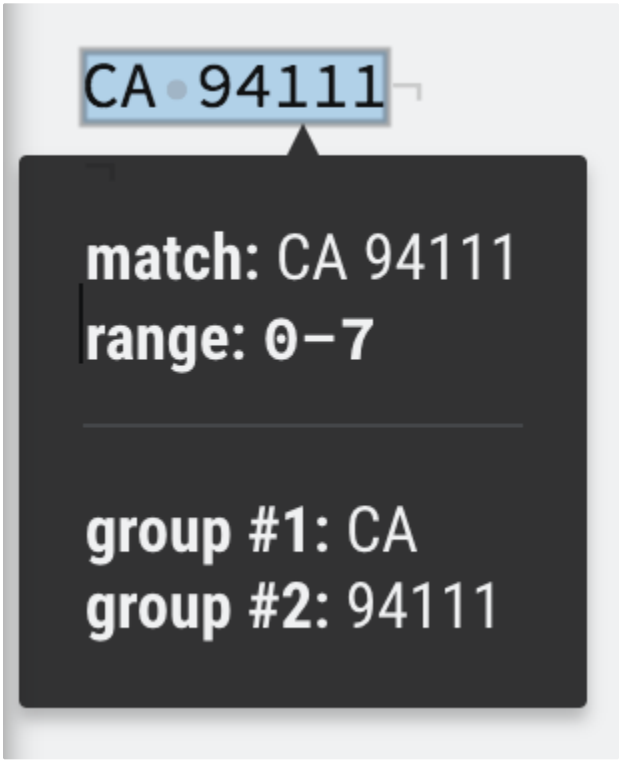
Capture Groups

We can use parentheses to denote capture groups, which allow us to extract individual pieces of information from a matching string. Let's return to to this example RegEx which matches a US state abbreviation followed by zip code:

> `[A-Z]{2}\s\d{5}`

Without capture groups, we can't isolate the individual state and zip code pieces in our match. Instead, we just get one match containing both. If we add in parentheses, we now have access to the separate components.

> `([A-Z]{2})\s(\d{5})`



Putting it all together

Let's write something a little bit longer. We'll write a regular expression which should match a 24 hour time format. All of the following should be considered valid times:

```
23:50:32 14:00 23:00 02:45 2:45 9:30:01 19:30
```

Notice that we must include hours and minutes, and we can optionally include seconds. Also, we can write a time with a leading 0 like `02:45` or we can omit it: `2:45`. The Regex should only work for valid times, so things like `25:80` should not work!

▶ View My Solution

Writing Code w/ RegEx's

Remember that the Regular Expression syntax is nearly identical across programming languages. I'm going to show some examples using JavaScript and Python, but you are not limited to those languages!

JavaScript & RegEx

JavaScript comes with a handful of built-in RegEx methods including:

- `exec`
- `test`
- `match`
- `matchAll`
- `search`
- `replace`
- `split`

Before we can use any of the above methods, we need to write our RegEx in JavaScript. Fortunately, it's nice and easy. We simply wrap our RegEx inside of `/your regex here/`

```
const timeRe = /\b([01]?[0-9]|2[0-3]):([0-5][0-9])(?::([0-5][0-9]))?/
```

Now let's use our time expression. We'll start with test, which tests for a RegEx match in a string and return true or false.

```
const timeRe = /\b([01]?[0-9]|2[0-3]):([0-5][0-9])(?::([0-5][0-9]))?/ timeRe.test("45:15") //false timeRe.test("10:36") //true timeRe.test("04:11:59") //true timeRe.test("He ran a marathon in 3:20:15") //true timeRe.test("10-23-54") //false
```

If we want to do more than test is a string contains a match, we can use the exec method which will return information about the specific match. If exec finds a match, it returns an array. The returned array has the matched text as the first item, and then one item for each capturing parenthetical in the matching text:

```
const timeRe = /([01]?[0-9]|2[0-3]):([0-5][0-9])(:[0-5][0-9])?/
timeRe.exec("12:45") // returns ["12:45", "12", "45", ... ]
```

Now we could use the entire match, or just the isolated hours, minutes, and seconds.

Python + RegEx

Python comes with its own suite of RegEx methods, which are included in the re module. I'm just going to focus on one: search. The search method will scan through string looking for the first location where the regular expression pattern produces a match. It returns a corresponding match object, which has its own set of methods you can use.

Let's use the exact same RegEx as before:

```
import re time_re= r'\b([01]?[0-9]|2[0-3]):([0-5][0-9])(?::([0-5][0-9]))?' test_str = "it is 4:30:47" result = re.search(time_re, test_str) #Now we have a match object which we can access result.group(0) # returns entire match '4:30:47' result.group(1) # '4' result.group(2) # '30' result.group(3) # '47'
```

What I didn't cover

This is just a brief intro to Regular Expressions, and I didn't have time to include every single topic. Here are some of the concepts I omitted that you may want to look into:

- Numeric References
- Positive and Negative Lookarounds
- Substitutions
- Flags