SLIDE: Title Slide

Let’s talk about Regular Expressions.

Honestly, the first time I saw regular expressions I was intimidated.

SLIDE: Scary Regular Expression

They seemed like cryptic, over my head, and frankly scary gibberish.
In this lesson I want to show you that regular expressions don’t have to be scary, how extremely powerful and elegant they are, and a few of the awesome features Ruby has to offer.

SLIDE: So what is a Regular Expression?

SLIDE: A Regular Expression is a pattern for a string

It’s a pattern that you can match a string against. It’s just a pattern. You can take a string and test whether it contains this pattern.

SLIDE: What can you do with Regular Expressions?

You can test a string to see whether it matches a pattern

You can extract a section or sections of a string that match all or part of a pattern

You can change a string, replacing parts of the string that match a certain pattern

We’re going to cover each of these scenarios in detail.

First, though, let’s talk about creating a regular expression.

SLIDE: Creating a Regular Expression

To create a regular expression, you put the pattern between two forward slashes. This marks it as a regular expression in your Ruby code.

We’re going to see lots more examples of this.

SLIDE: Scenario #1: Testing Strings

Let’s test some strings.

SLIDE: “dog and cat”
PickAxe give a pretty good example. Let’s say we have string “dog and cat”

SLIDE: /cat/

And I want to find out if this string contains the pattern “cat,” I want to know if “cat” appears anywhere in the string. This is a very simple example.

SLIDE: /cat/ =~ “dog and cat”

In Ruby, I can use a pretty simple bit of code to find this out. This bit of code asks “does dog and cat contain the pattern /cat/?

SLIDE: =~ Operator

This operator, the equals sign and the tilde, matches a string against a pattern. It then returns the character number where that pattern started. So if I ran this code in irb:

SLIDE: irb

I would get back the number 8. This means the pattern “cat” starts at the eighth character of the string. It not only tells me the pattern is there, it also tells me where it is.

SLIDE: irb 2

If the string DOES NOT contain the pattern I’m looking for, the statement will return nil.

SLIDE: Match method

So those are some pretty simple examples from the book. Let’s look at something a bit closer to the real world

SLIDE: Valid Emails

Let’s say I have a program that takes in emails from users. This could be a command line app or a webform. I want to make sure that any email is in the format for an email address.

SLIDE: thing@thing.thing

An email address needs to have some characters, followed by an @ sign, followed by a dot, followed by more characters. That’s just the way email addresses are. We need to make a pattern, a regular expression, and test whether the user input matches that pattern.

SLIDE: Rubular

In doing that, I want to introduce you to a great tool called Rubular.

(Pull up Rubular)

Rubular is a tool that lets you test a regular expression against a string.

So I want to test that my regular expression will accept a valid email address. I'm going to put mine in here

nellshamrell@gmail.com

Now let's do a simple regular expression

/nellshamrell@gmail.com/

You can see that it matches. But...it will only match my personal email address. I certainly hope not all visitors to a website that uses my code will not input my email address!

Let's add something to the list...how about Renee?

renee@nird.us

You can see it doesn't match Renee's email address. I could hardcode her address in here as well using this pipe character.

SLIDE: Alteration

This is known as alteration in regular expressions.

I use the pipe symbol, this vertical line, to provide two options, two alternate patterns that can be matched. By the way, you can do this with more than two options. Just divide each option with a pipe.

But, I'm still having the same problem. I've hardcoded in two email addresses, when there are millions of possibilities for email addresses.

I need metacharacters.

SLIDE: Metacharacters

These are special characters in regular expressions, that mean special things.

A "dot" or "period" can stand in for any single character

A "star" or "asterisk" next to a character means that character can appear any number of time
So if we put them together...

SLIDE .*

And this would match any character appearing any number of times. So going back to Rubular, I could do this

```
.*@.*\..*
```

This means the string can be anything, followed by an @ sign, followed by anything, followed by a . (notice I had to escape the . with a back slash. Remember, . is a metacharacter. Escaping it using this backslash tells the computer that I want to use a literal dot. Followed by anything./

But...what if I don't want to include ANYTHING. What if I want to limit email addresses to being just letters?

I can do this using a range

SLIDE: Range

A range lets me specify a range of letters, or a range of numbers. So I can specify that I will only accept letters between a and d, or I can specify that I will only accept numbers between 1 and 4.

So if I only wanted to include letters, I could specify

```
/[a-z]*@[a-z]*\.[a-z]*
```

But what if I want to let users enter capital letters? Say their caps lock key is stuck, or they just prefer it?

I can modify my ranges to allow this. Now I'll accept any characters a-z lowercase, and any characters A-Z upper case

```
[a-zA-Z]*@[a-zA-Z]*\.[a-zA-Z]*
```

But what about email addresses that contain numbers?

I, again, can modify my ranges to allow this.

```
[a-zA-Z0-9]*@[a-zA-Z0-9]*\.[a-zA-Z0-9]*
```

Now it accepts any characters a-z lower case, any characters A-Z upper case, and any digit 0-9.

That's a lot of code to write, isn't it? What if I told you there's a shorthand for this?

SLIDE: Shorthand!

/w stands for any word character, this includes all lower case letters, all upper case letters, all digits, and underscores. Regular Expressions have many shorthands like this, you can see that Rubular has a list of some of them. There are many more.

Back to Rubular.

Let's write our regex using this shorthand

/\w*@\w*\.\w*/

Just one more thing. The star operator means that a character can appear ANY number of times. This includes 0 times. So an email address with nothing before the @ sign or nothing after the . sign would still be accepted. Let's use a different metacharacter, the plus sign.

SLIDE: +

So let's modify our regular expression again

\w+@\w+\.\w+

So at this point, I'm saying that any email address submitted to my program must have at least one word character, followed by an @ sign, followed by at least one word character, followed by a ., followed by at least one word character.

So, were I to use this in my ruby code, I could write a method like this

SLIDE: Code

That was a LOT of information I just threw at you. Know that regular expressions take time to learn. There is a learning curve, and it's steeper than a lot of things in programming. But it is surmountable when you break things down. Don't try to go for a super complicated regular expression right away, do it step by step, testing as you go. Rubular is a great resource for that.

SLIDE: Scenario #2: Extracting Strings

I want to cover extracting strings using Regular expressions. Let's say I have a text file with a list of phone numbers.
(Show in Rubular)

I'm just going to make a list of some random phone numbers with Washington area codes. I want to extract all numbers with a 206 area code and print them on the screen.

We’ve already learned about ranges. I could just use several of them here

206-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]

This works, but it looks really ugly. Wouldn’t it be better if there were a shorthand way to represent a digit 0-9.

We’re in luck, there is!

SLIDE: /d

The /d shorthand stands for any digit 0-9. I can modify my regex to say this:

206-\d\d\d-\d\d\d\d

That’s still a lot of typing. I can actually make this even shorter using curly braces.

SLIDE: Curly braces

Curly braces let me specify EXACTLY how many times a I want a character to appear. So if I use \d{3}, it looks for exactly three digits.

American phone numbers always have an area code, followed by three digits, followed by four digits.

So if I use this regex 206-\d{3}-\d{4}

I’m looking for 206, followed by a hyphen, followed by three digits, followed by four digits

One final thing with this scenario. Let’s say we have two people entering in this data. One person uses hyphens, the other does not. Let’s add a few hyphenless numbers in here. My regular expression as it stands isn’t going to pick these up. I want to make this hyphen options. Fortunately , there is a way to do this.

SLIDE: Question Mark
When I add a question mark after a character, it means that character is optional.

206-?\d{3}-?\d{4}

It makes the hyphen optional. The hyphen can occur zero or 1 time.

So if I have a text file of phone numbers, with a little Ruby magic I can extract all the phone numbers with a 206 area code, all the ones that match this pattern, and print them out, insert them into another file, etc.

SLIDE: Scenario #3 Changing strings

The final scenario I’m going to cover is changing strings with regular expressions.

Let’s say I have this string.

SLIDE: String

“Renee is the teacher of the class right now. Renee is teaching about regular expressions.”

Well, this string is not accurate. I’m teaching the class right now, and I’m the one teaching about regular expressions. So I want to replace any part of the string that says “Renee” with “Nell”

Luckily, Ruby has a few methods I can use.

SLIDE: Sub

The first of these is the sub method. This method takes a pattern and some replacement text. Anywhere it finds a match for the pattern, it will replace that match with the text.

So, let’s look at this one in irb, actually.

I’m going to assign my string to a variable.

string = “...”

Then call sub on it

new_string = string.sub(/Renee/, “Nell”)

I’m telling it to look for “Renee” in the string. If you find it, replace it with “Nell”

One thing to note is that the sub method only changes the FIRST match it finds. If you want to replace all of the matches, you need to use gsub.

SLIDE: gSub
The g stands for global, and it will replace ALL the matches.

new_string2 = string.gsub(/Renee/, “Nell”)

SLIDE: sub! and gsub!

Notice that sub and gsub don’t change the original string, they create a new string. If I wanted to change the original string, I would need to add a ! to it, or a bang.

```
string.sub!(/Renee/, “Nell”)
string.gsub!(/Renee/, “Nell”)
```

SLIDE: Conclusion

So, welcome to the wonderful world of regular expressions. I’ve just barely scratched the surface with this lesson, there’s much, much more out there to learn. Some resources include the O’Reilly books on Regular Expressions and the online resource “Regular Expressions: The Hard Way” by Zed Shaw.