Chapter11.2

So, now we're going

to talk about extracting data.

Up to now, we've just been playing with the search which gives us

back a true or a false depending on whether it matches or not.

But now we're going to actually pull stuff out.

So, we're going to start by looking at a different regular expression,

a new regular expression.

The square bracket is kind of weird and that it is one character.

So, that is describing in between the square brackets what we mean by a single character.

We can have a range in here.

We can have a list of things,

like AEIOU would be vowels.

Zero through nine is digit.

So, bracket zero dash nine bracket is a single digit.

But then, we added a plus to it and that says one or more digits.

Now, if we put a star that zero or more digits which is kind of silly.

But one or more digits,

and now we're going to use a function called findall,

a function in the regular expression library called findall.

So, what we're saying here is this is the string we're looking through,

x, and we're looking for the pattern,

one or more digits.

So, then it's going to look and say,

"Oh let me see, one or more digits."

Oh! That looks good I like that one.

Let's keep looking. That's good and let's keep looking and that's good.

So, it may find zero,

it may find one, or it may find more than one.

So, what it does is it runs

all the way through the texts that you've asked it to look for,

checking to see when this matches,

and it gives us back a list of the matches.

So, it extracts out the pieces.

So, this is kind of like a split in a for loop,

and checking to see if it's a number and a whole bunch of

stuff all rolled into one in one little programming.

Because findall- findall, if it gives us nothing,

will give us an empty list but in this case,

it's given us three strings.

Now they're not numbers,

this is the string 2,

this is the string 19,

and that's the string 42.

But that's what we get back.

We get back a list from findall of all the possible matches.

Okay. Pretty powerful.

Okay. So, it returns zero more things,

we can, in this case,

we asked for one or more digits.

In this case, I'm saying one or more,

so that's a single character.

It's an uppercase vowel,

A E I O U, all uppercase.

Plus means greater than one or greater than or equal to one.

So, there's got to be at least one and you say "Are there any uppercase vowels in here?"

No no no no no no no.

So, it doesn't find it.

So, I get back nothing. So, it has to give me a list.

Find all of the substrings that

match that regular expression and given back to me, there were none.

So, you have an empty list.

So, you do have to check to see how many things you got back?

Because you might get one, you might get zero,

you'll get like in 25 things back

from a particular regular expression when you give it a line.

Now, as you're thinking about this,

you think of the regular expression.

It's almost like a stamp or it's going stamp stamp stamp.

Can I? Is this piecework?

Is this piecework? Does this piece match? Does this piece match?

The problem is there is this notion in the matching called Greedy Matching,

and unless you say otherwise,

the regular expression library attempts to give you

the largest possible version of the string that you're matching.

So, here we have the first character as an F. Any character,

one or more times,

and then I start with a colon.

If this is the text that we're looking at, you would say "Oh,

yeah there's the beginning F, and there's a characters and there's a colon, were done."

The problem is that it doesn't stop there.

It's like "Oh, wait a sec."

Technically, this also matches, so what do we get back?

Do we get back the from?

Or do we get back the whole thing?

Greedy Matching says you're going to get back the larger thing,

and that's exactly what you get,

and so all else being equal,

you got to be careful when you construct these things.

Now, I could have put non blank in there but I'm doing this to make the point

to say that in a sense this is pushing. That's the greediness.

Is that this wants to be as big as it

possibly can be and then still match the entire expression.

So, if you're thinking stamping this expression on that string,

you can stamp it on the small thing or you can stamp it on the big thing,

it says "I'll take the big thing."

Now, you can override this.

But basically, you can think of this kind of these wildcards is very pushy.

Very pushy outwards, greedy as larger possible string,

and that's what we mean by greedy.

Both the asterisk and the plus push outwards as far as push as wide as they can.

But just like everything in regular expressions,

so you can fix that with another character.

So, now we have a three character sequence.

To the plus or the asterisk,

we can add a question mark.

So, this says any character,

one or more times,

but don't be greedy.

So, now it looks at it and says "Okay, I've got a beginning F,

and I can stop here,

or I can stop here,

but I am not greedy."

So, the Non-Greedy prefers the shortest.

The greedy prefers the longest.

The Non-Greedy prefers the shortest, and so this is what we get.

Again, when you're writing code using regular expressions,

it's really important that you test your code,

so that you see kind of weird anomalies like this like "Wow!

Why did I get that? What's going on?

Why not that?" Then you run it and then you rise, "Oh yeah.

It's Greedy Matching. It pushed really hard."

Usually, it doesn't take too long to figure that out,

but you do have to sometimes check it and so

sometimes you got to like do something like I had this question mark, don't be greedy.

Okay, so just a fascinating thing, you're coding.

That's like an if statement.

The question mark is like an if statement.

Hey, do the shortest one and you communicate that in a single letter.

That's why they're kind of fun or like a whole programming language in characters.

Okay. So, here we have- we're looking for the email address.

The common one of the things we're trying to do is

take those from lines and terrible part, right?

So, what we say is "Hey,

let's go find everything that matches a non-blank character,

one or more non-blank characters followed by an @ sign,

followed by one or more non-blank characters."

So, yeah. This a non-blank character, but there's no @ sign.

This is non-blank character,

oh yeah there's an @ sign followed by some non-blank characters.

So, that's a yes match,

and then none of these other sets of non-blank characters match that, right?

So, that comes out, and so there you go and we get out exactly what you'd

expect we get the non-blank characters

followed by an @ sign followed by some more non-blank characters.

I've gotten pluses to make them be one or more.

\S is a non-blank character,

if you go back to the cheat sheet,

that was part of the non-blank character.

Okay. You can think of this as also greedy meaning they're kind of pushing.

So, this technically d@u

would be a one or more non-blank characters followed by an @ sign,

followed by one or more non-blank characters.

But with greediness, it pushes outward,

and so it goes as far as it can unless when you do want to be greedy so you get this.

If you made this Non-Greedy,

you would get d@u.

So, that also kind of helps you understand how greediness and Non-Greedy wants.

Now, we can adjust how findall works by using parentheses,

but this is not really using parentheses here,

so we'll do that next.

So, we can fine tune the string extraction,

and have more that we're matching than we're extracting.

So, if we look at this particular example,

where we add carrot from space and

then some non-black- one or more non-blank characters followed by an @ sign,

one or more non-blank characters,

This matches this, right?

So, it's a from, followed by a space,

followed by one or more non-blank characters,

followed by an at sign, followed by one or more non-blank characters.

It's like this part here is a match.

But we don't actually want to get back the whole thing,

and so we can add parentheses.

So, what I'm doing is I'm saying,

start extracting after the space,

from space is part of the match but the extracted part starts here,

and then the extracted part ends here.

So, that says, this is the part that I

want to extracted even though I demand this to match.

So, I'm demanding, I'm extracting less than what I'm matching.

I'm using the matching to be very precise as to the lines I want,

and then I'm using the parentheses that I add to pull out what I want.

So, here I get back exactly the email address,

even though now I'm already in this one thing,

making sure it's from lines that have a prefix of from space.

So, I've got lines of prefixes from

space extract the second thing and now it's not just anything,

but it's got to be from space and then immediately non-blank for characters,

followed by an at sign, followed by non-blank characters.

So, again this is really fine tuning.

Okay, so let's take a look at this thing that we're doing long

time ago but without regular expressions.

And so the idea is we want to pull this little bit out.

Here's the old one, we find the at position which is position 21 and so that gives us 21,

we start at that position, so,

we look up and we say when's the next space?

We get 31, and that comes into here.

So, we say we want to do a string slice from one

beyond the app position up to but not including the space,

remember up to but not including.

That prints us out this little piece.

But we can do a similar thing

with regular expressions and we've seen this with dual split.

So, this is the find way of pulling that out, dual split is,

we split it into words with spaces,

then we grab the second one,

we split that by at signs and then we grab the second piece of that.

So, we take the second word,

we split that second word by at sign,

and then we take the second piece, and then we get this.

So, we were able to do that with four lines,

a little more elegant.

But if we do regular expressions, we can say, hey,

go find me an at sign,

followed by some number of non-blank characters.

I don't want to extract the at sign,

see where I put the parentheses,

I want to start extracting after the at sign

and up to the rest of those non-blank characters.

So, that says "buf," I've got what I want.

So, it's a way to say in a little expression.

从 :10:40 开始播放视频并学习脚本10:40

Match a non-blank character, that's with the brackets,

so that's another syntax and that is,

this is a single character but if the first letter

of the set inside there is the character,

that means not, everything but.

So, that means everything but a space, that's non-blank.

So that's everything but a space asterisk.

There's other ways to do that,

but that's what this is saying.

That's a single non-blank character zero or more times but that's what I want to extract,

and again outcomes this little bit.

We can fine tune this by saying I want to start with from in the line,

I want to space,

but I want any number of characters up to an at,

and then I want to begin extracting all the non-blank characters,

and then end extracting.

So, this is adding this bit to it,

this fine tuning that's also in a way could be used to filter the line.

So, if you if you didn't have a from line,

you would get nothing back,

and you're not finding email addresses in the middle of text,

you're just finding email addresses on lines that start with from space.

So, you just build these things up,

you tell the regular expression what you want back,

and you get back a list.

Like I said, you got to check to see if the list is empty

because that is your way of knowing that it didn't match anything.

So, here's a little bit of code that sort of uses

regular expressions to both pick lines and extract data.

So, this is similar to one of

the assignments where you're going to look for lines that have a form like this,

they say X dash DSPAM Confidence colon,

space, and then a floating point number.

So, we're going to run through this,

we'll open the data, we'll read through the lines,

we're going to strip the data,

and now we're going to use, find all,

to look for lines that start with X dash DSPAM dash Confidence colon.

Quite a bit it's tough, it's got to match every character,

followed by a blank, then start extracting,

take zero to nine and period because we're looking for floating point numbers, so,

we want to get the period, bracket,

one or more times,

and that's what we're interested in.

Now, here's the part where you have to check.

If we are looking for a line that we looking for here,

there's going to be exactly one successful extraction.

If you don't have the prefix or don't have the number,

then you're gonna get zero extractions.

So, what I'm basically saying is,

this stuff is a list of the matches,

if that length is not one,

meaning it's bad if it's two because that means

there's more floating point numbers out here, how did that happen?

Do we know or who knows what?

It's unlikely that this is going to match more than one,

so, we're not going to do that.

It's one floating point number is what we got on the line,

then we're in good shape.

Otherwise, we're going to skip that line,

and so this is both a filtering like an if blah, blah, blah,

continue or if not starts with continue,

and if it finds it,

it also parsed the line and done the split and pulled all those things out.

So, that's how with regular expressions,

you can make programs more succinct.

When you see someone else's regular expression,

it might take you a little while to figure out what the heck this is doing.

You have to read it but the nice thing is it's not a bunch of lines,

and so it's a way to make your programs shorter, don't overuse it,

put a few comments in,

pound sign blah, blah, blah,

pound sign blah, blah, blah,

this is looking for a line that's of this particular syntax and blah,

blah, blah, blah, blah, blah,

some kind of comment that help your reader out.

But once you get used to them and you will start to see them,

they are often used for data validation,

for searching and extracting.

Now, we've got all these characters, weird little characters,

dollar signs, carrots etc.,

and sometimes we actually want to match those characters.

So, we have one more special character and it's the backslash,

and the backslash just can be prefixed otherwise active character,

so, dollar sign has meaning,

but slash dollar sign means it's a really a dollar sign.

So, if I'm looking for strings that start with a dollar sign,

have a numbers and dots,

and the non-blank characters, that says,

give me the strings that start with dollar sign,

one or more numbers and or dots,

and so that then matches this bit right here and pulls it out.

So, escape characters when you really want one of those characters like a bracket,

or an asterisk, or a plus, or a dot.

So, that's a quick zoom through regular expressions.

They're fun, they're fascinating,

they lead to elegant code when used appropriately.

I would suggest you don't overuse them,

but there are some times that they just are the right thing to do,

and so don't try to confuse your reader of your code,

because the reader of your code might be you in the future,

but they're really interesting and powerful and you'll probably see code that uses them.

So, thanks a lot.