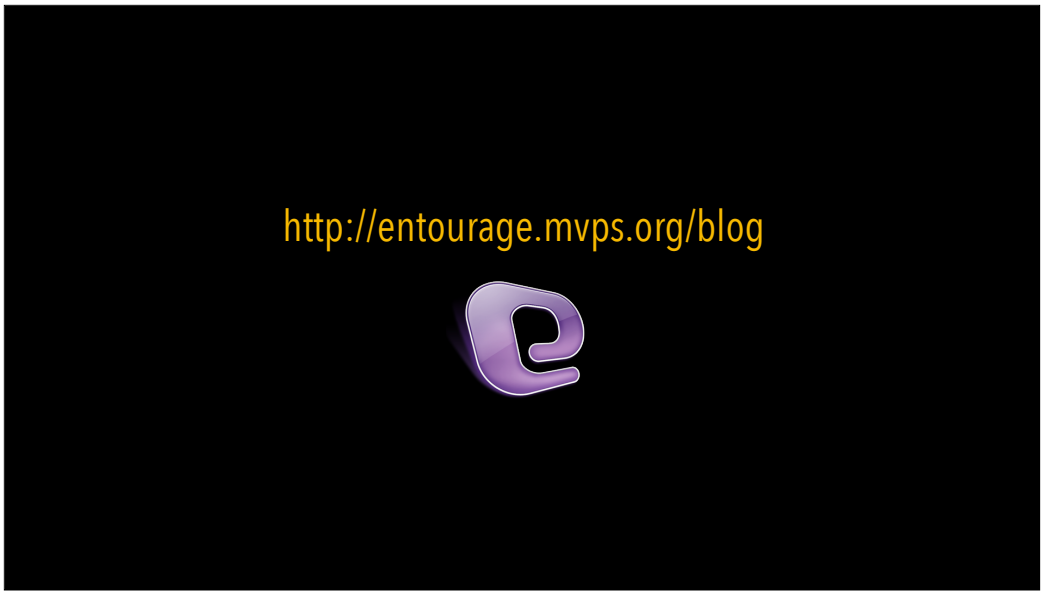
An Introduction to

(re.ex|re+gex|re?gex|re*gex){1}

Hi everyone!

I’m William Smith, a Professional Services Engineer with Jamf, and today we're going to talk about **finding trees in a forrest**.

Almost 15 years ago, I was asked by a colleague — Diane — to join her in **starting a blog** for her website about Microsoft Entourage. For those of you who aren't familiar with Entourage, it was the predecessor to today's Microsoft Outlook for Mac. We started the blog in 2007 and maintained it for three years as a source of help, tips, tricks and troubleshooting all things Entourage.



Almost 15 years ago, I was asked by a colleague — Diane — to join her in **starting a blog** for her website about Microsoft Entourage. For those of you who aren't familiar with Entourage, it was the predecessor to today's Microsoft Outlook for Mac. We started the blog in 2007 and maintained it for three years as a source of help, tips, tricks and troubleshooting all things Entourage.

Then in 2010, we learned Microsoft would be releasing Microsoft Office for Mac 2011 and **replace Entourage with Outlook**.

Well, there went the domain name.

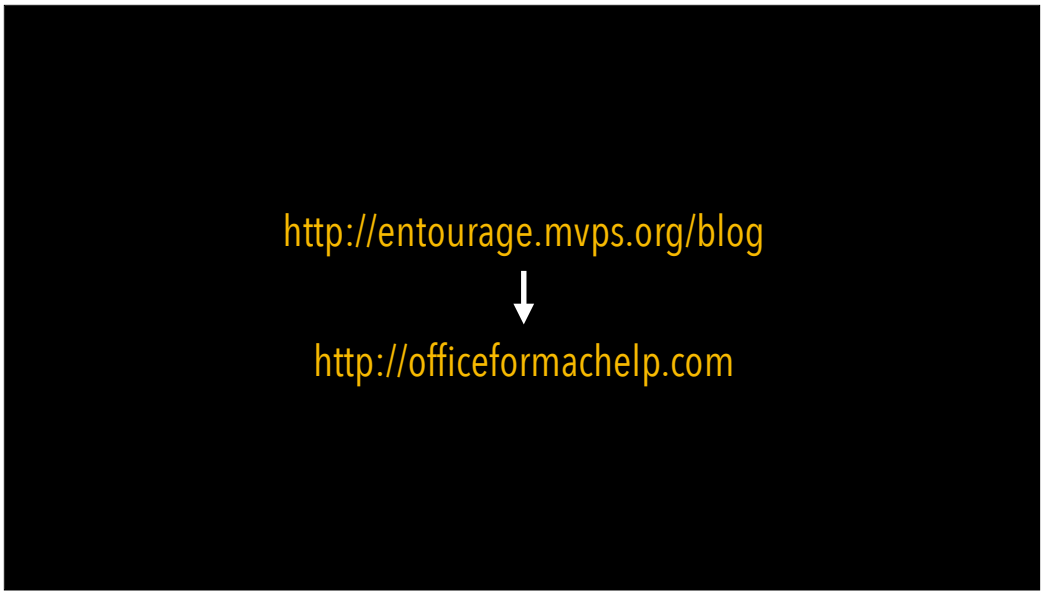
After some discussion, Diane and I decided to take The Entourage Help Blog and turn it into a website that would be **about all of Office for Mac**, including Excel, PowerPoint and Word. We decided to call it OfficeforMacHelp.com.

We worked for a couple of months on the design while waiting for Microsoft to release the new editions of Office and we knew ahead of time what date that would happen.

There was a problem, though.

No one had ever heard of **officeformachelp.com**. It was an unknown website. Diane had spent nearly seven years establishing The Entourage Help Page website and together we'd added content over the next three years. We didn't want to lose all that search traffic that Google Analytics said we were getting when we changed to a new domain name.

Thanks to **Search Engine Optimization**, at one point it was a toss-up for the **top Google search result** between The Entourage Help Blog and Entourage, the HBO series.



Then in 2010, we learned Microsoft would be releasing Microsoft Office for Mac 2011 and **replace Entourage with Outlook**.

Well, there went the domain name.

After some discussion, Diane and I decided to take The Entourage Help Blog and turn it into a website that would be **about all of Office for Mac**, including Excel, PowerPoint and Word. We decided to call it OfficeforMacHelp.com.

We worked for a couple of months on the design while waiting for Microsoft to release the new editions of Office and we knew ahead of time what date that would happen.

There was a problem, though.

No one had ever heard of **officeformachelp.com**. It was an unknown website. Diane had spent nearly seven years establishing The Entourage Help Page website and together we'd added content over the next three years. We didn't want to lose all that search traffic that Google Analytics said we were getting when we changed to a new domain name.

Thanks to **Search Engine Optimization**, at one point it was a toss-up for the **top Google search result** between The Entourage Help Blog and Entourage, the HBO series.

Then in 2010, we learned Microsoft would be releasing Microsoft Office for Mac 2011 and **replace Entourage with Outlook**.

Well, there went the domain name.

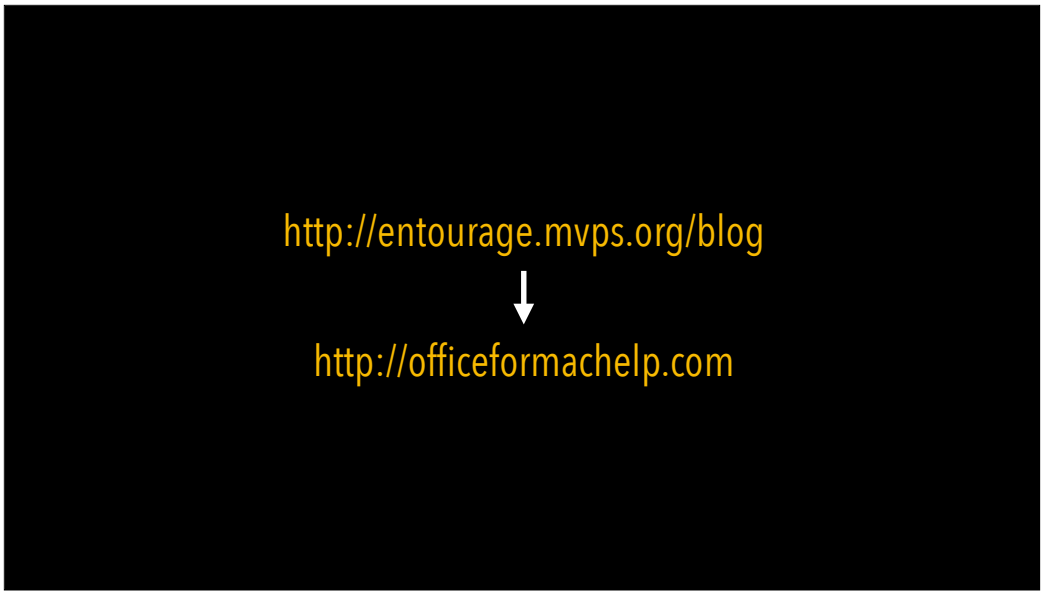
After some discussion, Diane and I decided to take The Entourage Help Blog and turn it into a website that would be **about all of Office for Mac**, including Excel, PowerPoint and Word. We decided to call it OfficeforMacHelp.com.

We worked for a couple of months on the design while waiting for Microsoft to release the new editions of Office and we knew ahead of time what date that would happen.

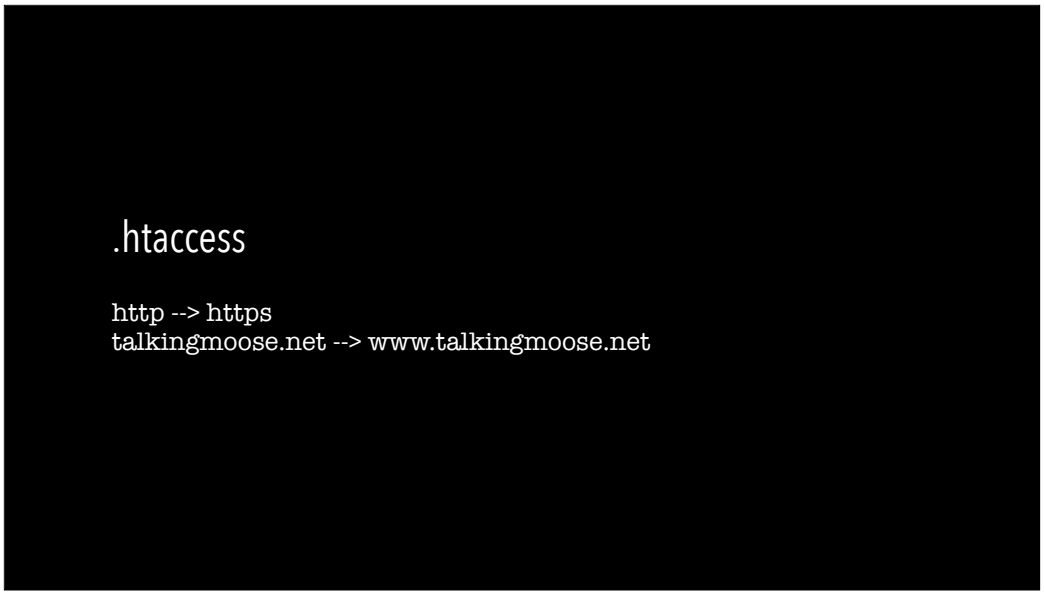
There was a problem, though.

No one had ever heard of **officeformachelp.com**. It was an unknown website. Diane had spent nearly seven years establishing The Entourage Help Page website and together we'd added content over the next three years. We didn't want to lose all that search traffic that Google Analytics said we were getting when we changed to a new domain name.

Thanks to **Search Engine Optimization**, at one point it was a toss-up for the **top Google search result** between The Entourage Help Blog and Entourage, the HBO series.



We asked our current hostmaster if there was anything we could do about this and he gave us a solution using regex. It was really simple and worked beautifully.



The root of most every website contains a **.htaccess** file. This is a configuration file for Apache web servers and it's quite often used to perform redirects.

For example, if someone tries to **access a page** on your site using **http**, a .htaccess file can redirect the page to **https**.

Or, if I wanted to **redirect someone** to my www page when entering just my domain, .htaccess can handle that.

In our case, we used a **rewrite** rule that included not only the domain name change but also the code for a permanent redirect, which is 301. The 301 told Google as well as visiting browsers to always redirect to officeformachelp.com if ever trying to visit entourage.mvps.org.

The regex that **made it all work** was this.

Basically, whenever someone visited the blog on entourage.mvps.org, we told the browser "Hey, we've moved to officeformachelp.com permanently. Just start going there from now on."

And to update Google searches, we simply had to submit a request online to have them rescan our site. In just a few days, a search for any of the **old content** linked visitors to the **new site**. Even if someone linked deep into the old site, the rewrite rule preserved the entire URL and visitors were directed to the same deep location on the new site.

So, what do all those backslashes, dollar signs, parentheses and asterisks mean? That's what we're going to talk about today.

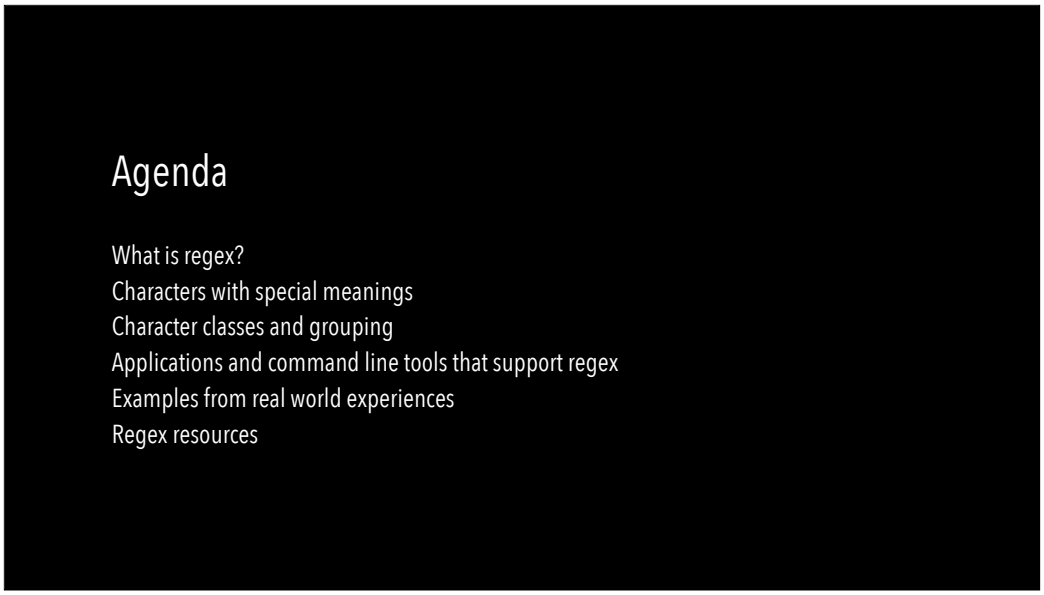
In our case, we used a **rewrite** rule that included not only the domain name change but also the code for a permanent redirect, which is 301. The 301 told Google as well as visiting browsers to always redirect to officeformachelp.com if ever trying to visit entourage.mvps.org.

The regex that **made it all work** was this.

Basically, whenever someone visited the blog on entourage.mvps.org, we told the browser "Hey, we've moved to officeformachelp.com permanently. Just start going there from now on."

And to update Google searches, we simply had to submit a request online to have them rescan our site. In just a few days, a search for any of the **old content** linked visitors to the **new site**. Even if someone linked deep into the old site, the rewrite rule preserved the entire URL and visitors were directed to the same deep location on the new site.

So, what do all those backslashes, dollar signs, parentheses and asterisks mean? That's what we're going to talk about today.



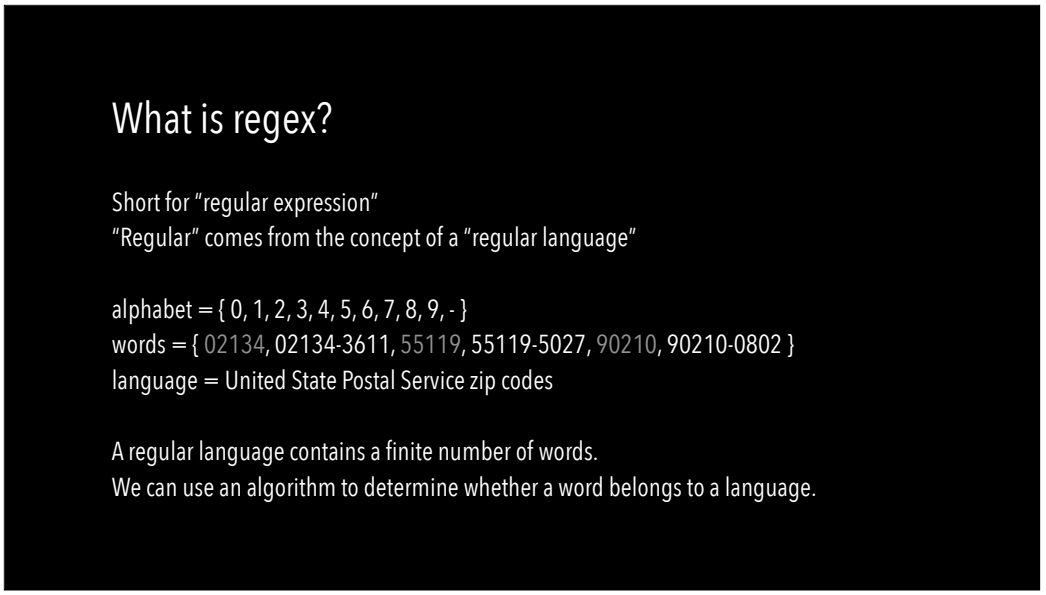
For the first half of my presentation today, I'm going to cover the basics including:

What is regex?
Characters with special meanings.
And character classes and groupings.

I don't have time to cover all the shorthand characters, but I'll cover a few, and I'll go over enough about regex to get you started using it.

Then, in the second half of my presentation, I'll talk about:

Applications and command line tools that support regex.
I'll give you some real world examples for using regex.
And then I'll give you some really useful resources for learning more regex and practicing.



So, let's get started.

What is regex?

Regex is short for "regular expression", which is a pretty confusing name.
The word "regular" comes from the concept of a "regular language", which is a computer science term.

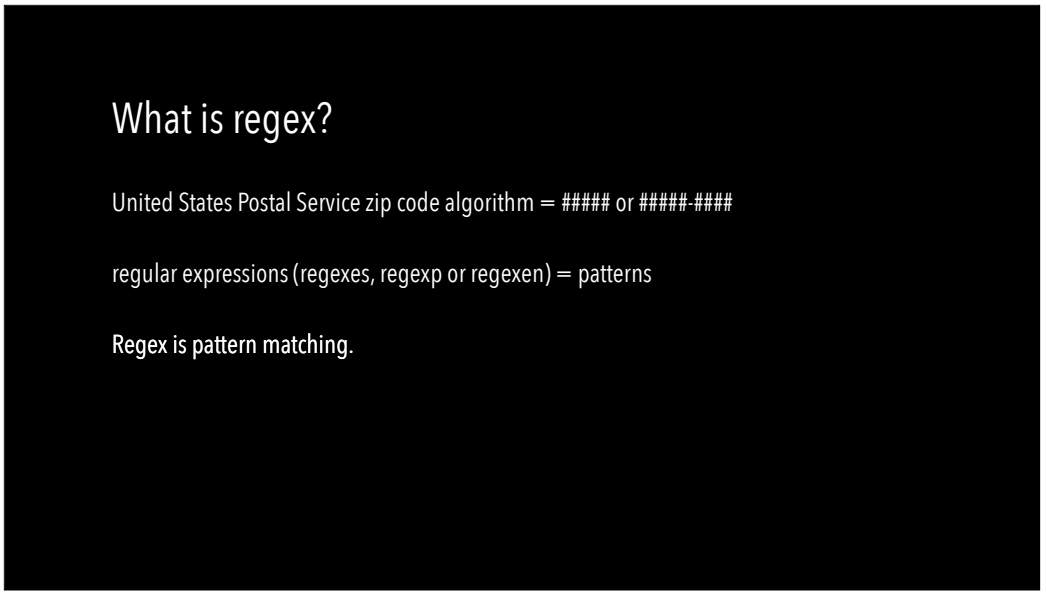
Basically, a set of symbols is called an "alphabet". But an alphabet doesn't need to contain letters. It can contain any set of symbols. In this example, the alphabet contains the numbers zero through nine. But notice also one of those symbols is a hyphen.

Now, putting **two or more symbols together** forms a "word". Here I have six examples of words. I've highlighted every other one just make the list easier to read. If you look at them closely, what kind of pattern do you see — what are these words? (Those of you from the United States should recognize them, but anyone else may not recognize the patterns.)

They're a list of 5-digit and 9-digit zip codes.

A "language" is a subset of all the possible words. The name of **this** language is "United States Postal Service zip codes". For those of you outside the United States, translate this to your own zip code structure. Or, you might use another example like the structure of telephone numbers.

Each word in our examples here contains either five digits or nine digits separated by a hyphen. That means there are only **so many possible words** in the US zip code language. A regular language can only contain a finite number of words. That means we can use an algorithm to determine whether a string of characters is really a word.

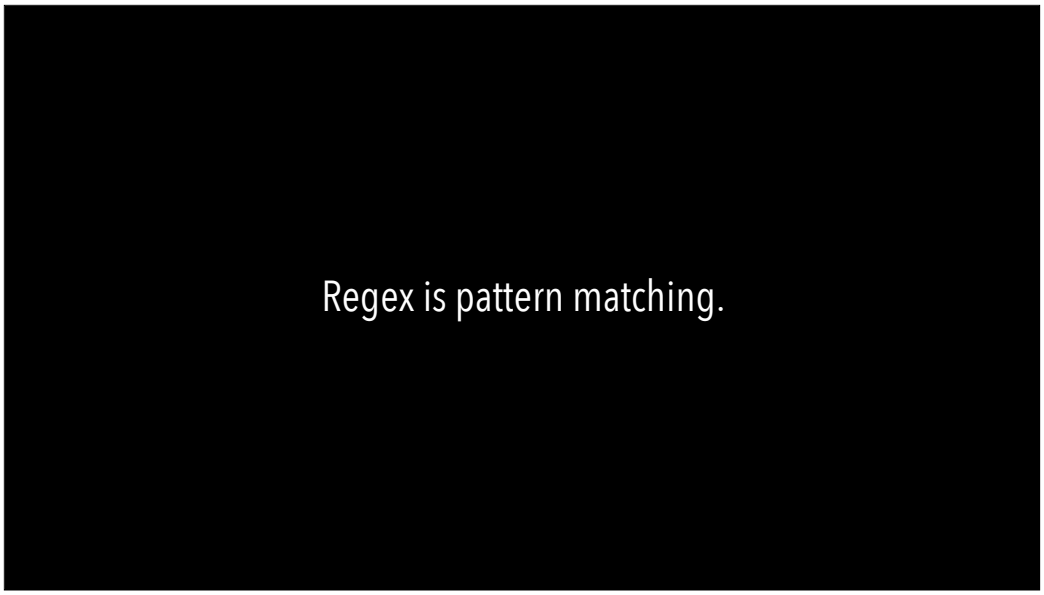


The algorithm for US zip codes is pretty simple. A valid word is either five digits or five digits followed by a hyphen followed by four more digits.

So, what we're effectively doing is looking for **patterns in the strings of words** we're examining. Regular expressions or regexes, regexp or regexen are patterns and when we use them, we're looking for **matching** patterns.

So, if **regex is about pattern matching**, what the heck then **is "pattern matching"** and what can I do with it?

In a word, we use regex for validation.



The algorithm for US zip codes is pretty simple. A valid word is either five digits or five digits followed by a hyphen followed by four more digits.

So, what we're effectively doing is looking for **patterns in the strings of words** we're examining. Regular expressions or regexes, regexp or regexen are patterns and when we use them, we're looking for **matching** patterns.

So, if **regex is about pattern matching**, what the heck then **is "pattern matching"** and what can I do with it?

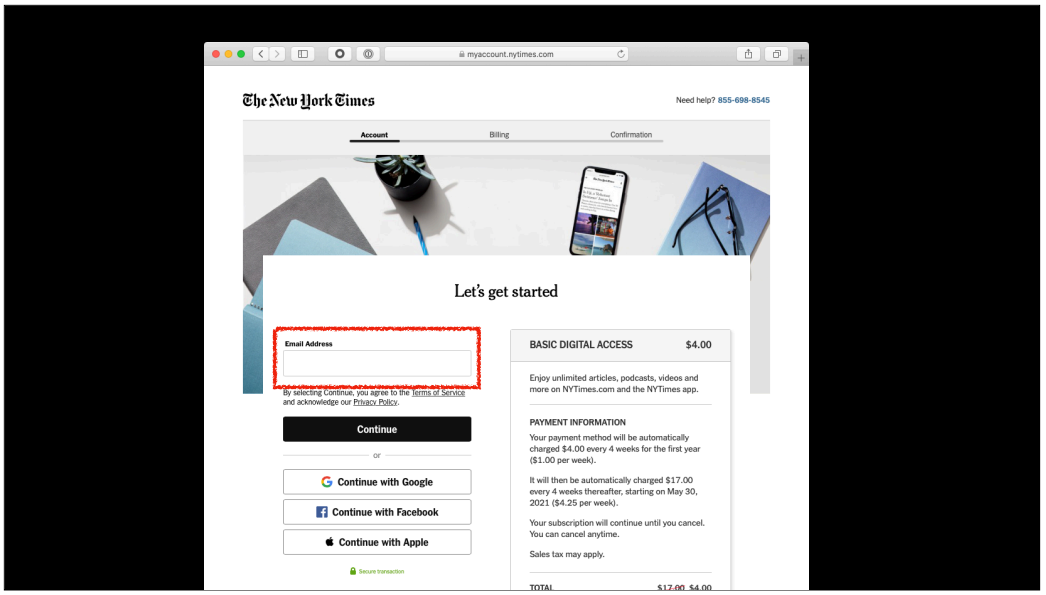
In a word, we use regex for validation.

The algorithm for US zip codes is pretty simple. A valid word is either five digits or five digits followed by a hyphen followed by four more digits.

So, what we're effectively doing is looking for **patterns in the strings of words** we're examining. Regular expressions or regexes, regexp or regexen are patterns and when we use them, we're looking for **matching** patterns.

So, if **regex is about pattern matching**, what the heck then **is "pattern matching"** and what can I do with it?

In a word, we use regex for validation.



Here's the subscription page for The New York Times. **Today**, your email address is your unique identifier and it's required for a lot of online transactions like subscriptions. It has to be a valid email address or The Times won't let you subscribe.

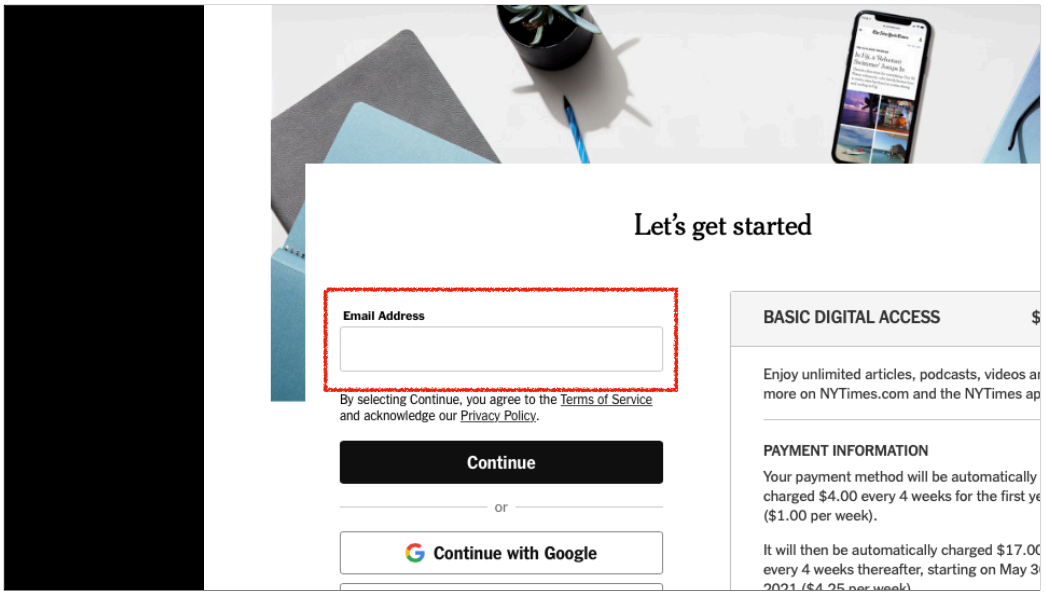
How does the web developer know that what you've typed is an email address?

What if I typed "jerry"? Is that a valid email address? The page doesn't seem to like it.

Or what if I put in a phone number instead of an email address? Does that work? No, it doesn't.

But if I put in my email address, the page seems to be OK with it. Why?

Is The New York Times really storing a database of every possible email address and comparing what I enter to that?



Here's the subscription page for The New York Times. **Today**, your email address is your unique identifier and it's required for a lot of online transactions like subscriptions. It has to be a valid email address or The Times won't let you subscribe.

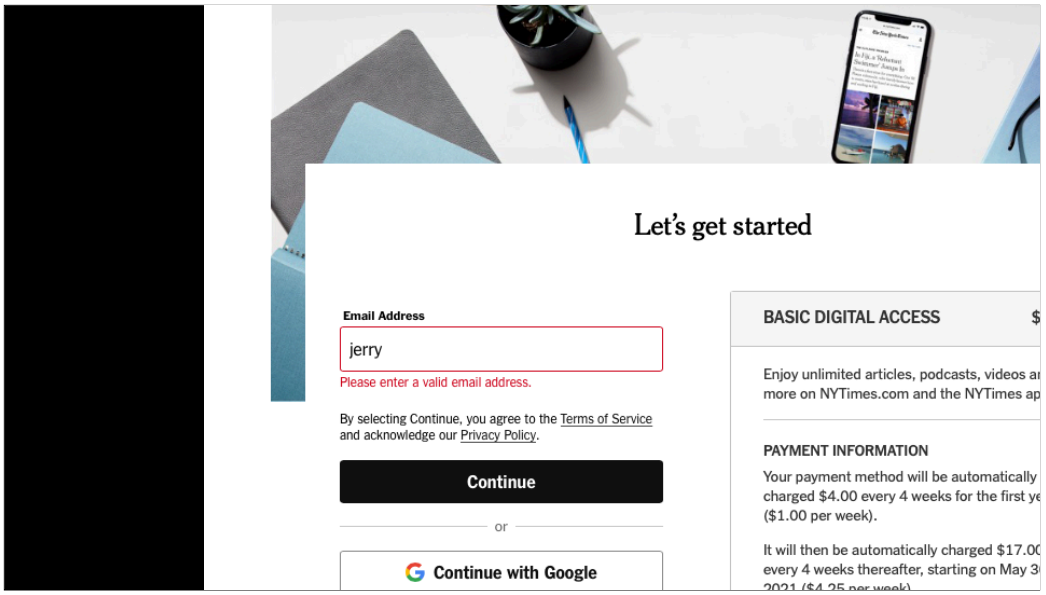
How does the web developer know that what you've typed is an email address?

What if I typed "jerry"? Is that a valid email address? The page doesn't seem to like it.

Or what if I put in a phone number instead of an email address? Does that work? No, it doesn't.

But if I put in my email address, the page seems to be OK with it. Why?

Is The New York Times really storing a database of every possible email address and comparing what I enter to that?



Here's the subscription page for The New York Times. **Today**, your email address is your unique identifier and it's required for a lot of online transactions like subscriptions. It has to be a valid email address or The Times won't let you subscribe.

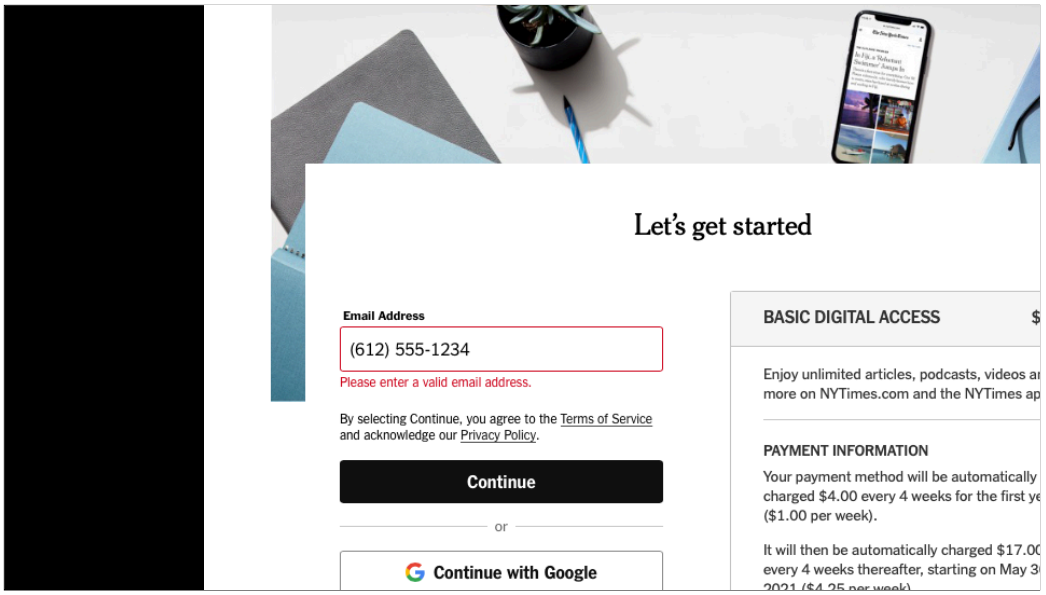
How does the web developer know that what you've typed is an email address?

What if I typed "jerry"? Is that a valid email address? The page doesn't seem to like it.

Or what if I put in a phone number instead of an email address? Does that work? No, it doesn't.

But if I put in my email address, the page seems to be OK with it. Why?

Is The New York Times really storing a database of every possible email address and comparing what I enter to that?



Here's the subscription page for The New York Times. **Today**, your email address is your unique identifier and it's required for a lot of online transactions like subscriptions. It has to be a valid email address or The Times won't let you subscribe.

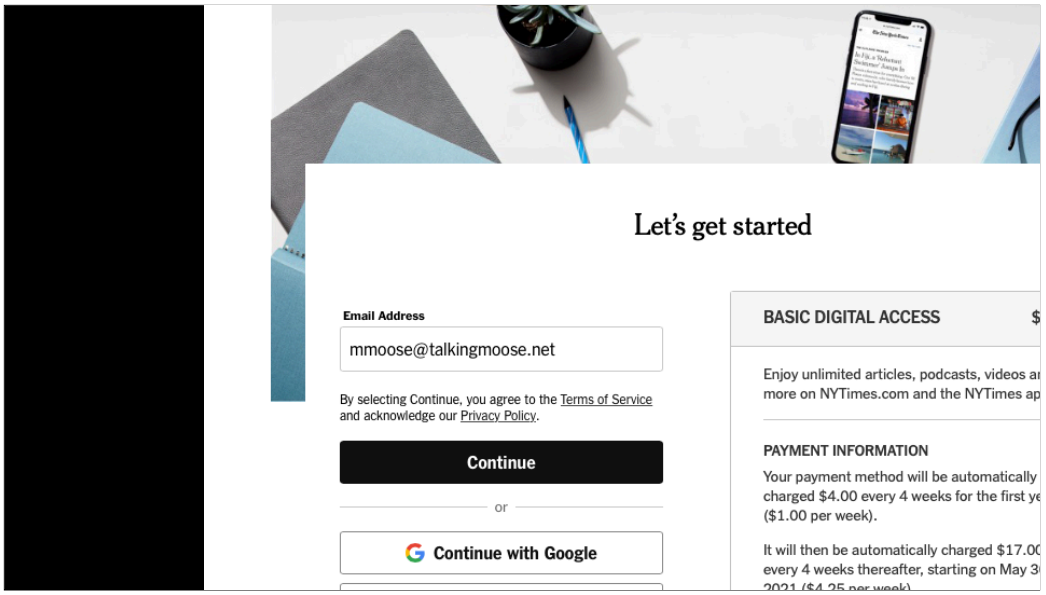
How does the web developer know that what you've typed is an email address?

What if I typed "jerry"? Is that a valid email address? The page doesn't seem to like it.

Or what if I put in a phone number instead of an email address? Does that work? No, it doesn't.

But if I put in my email address, the page seems to be OK with it. Why?

Is The New York Times really storing a database of every possible email address and comparing what I enter to that?



Here's the subscription page for The New York Times. **Today**, your email address is your unique identifier and it's required for a lot of online transactions like subscriptions. It has to be a valid email address or The Times won't let you subscribe.

How does the web developer know that what you've typed is an email address?

What if I typed "jerry"? Is that a valid email address? The page doesn't seem to like it.

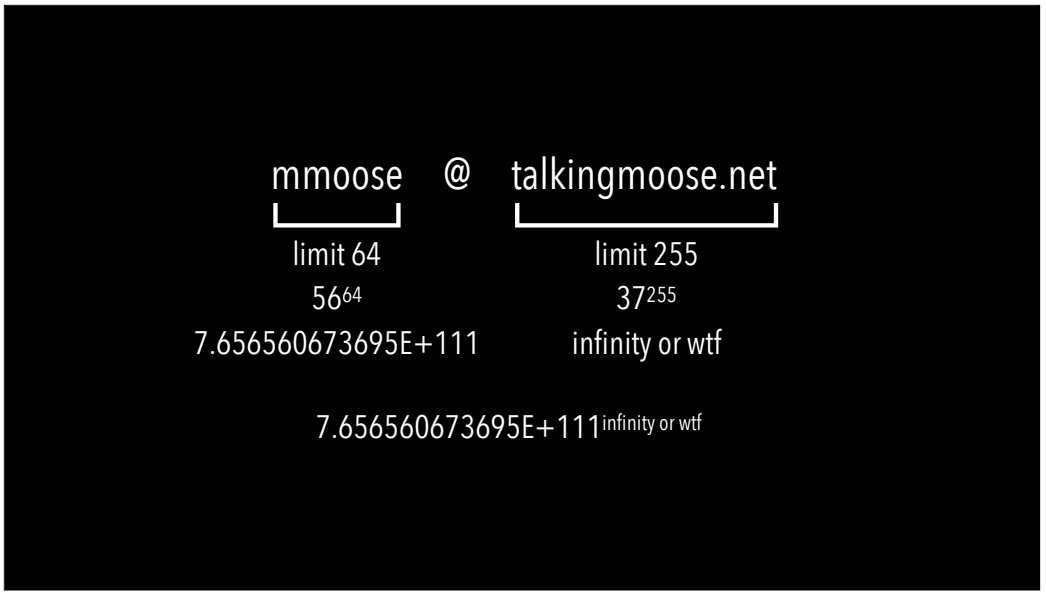
Or what if I put in a phone number instead of an email address? Does that work? No, it doesn't.

But if I put in my email address, the page seems to be OK with it. Why?

Is The New York Times really storing a database of every possible email address and comparing what I enter to that?

Well, email addresses are limited to 320 characters, which is a very finite number. Right?

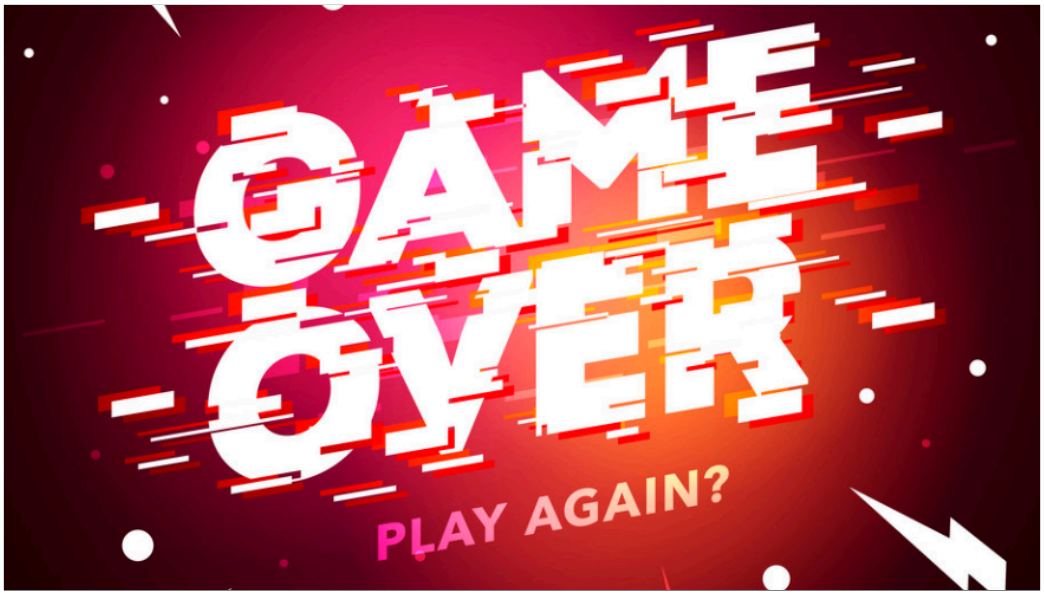
So, what are the permutations of that?



Given that the local part of the address (the part before the @-sign) is **limited** to 64 characters and **there are about** 56 allowable characters, we just need to **calculate** 56 raised to the 64th power. That's pretty straight-forward.

Now, the domain part of the address (the part after the @-sign) **has a higher limit** of 255 characters but about **half the allowable characters** at just 37. Again, we just need calculate 37 raised to the 255th power and for all **practical purposes** that equals roughly infinity.

Don't forget we now need to calculate the the potential of all those **local parts of the address** (the part before the @-sign) to the potential of all the domain parts of the address (the part after the @-sign) and...



Congratulations! You've just crashed the Internet.

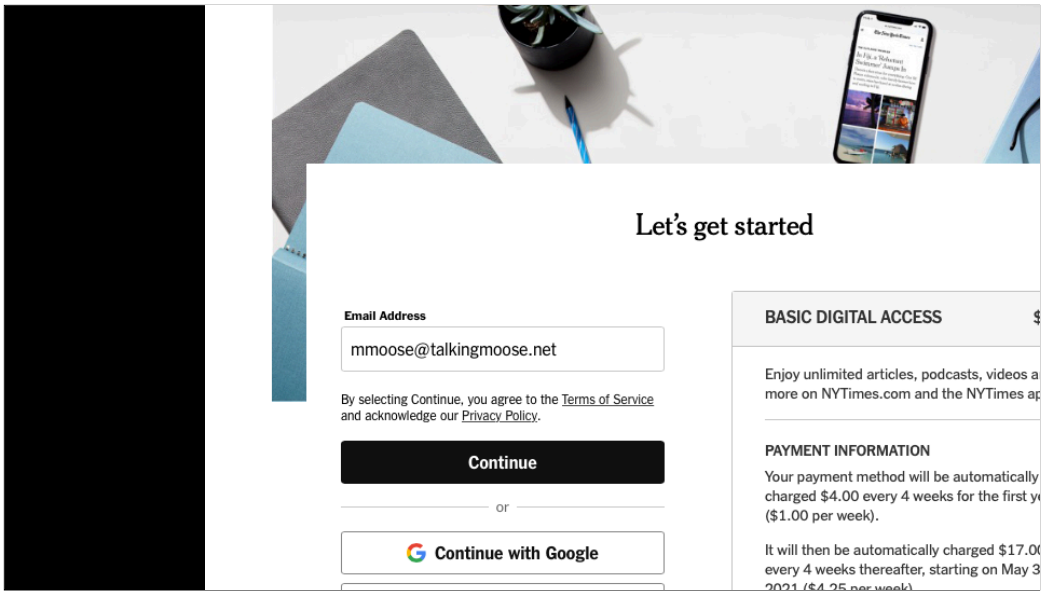
Not only is creating a database of all possible email addresses impractical, it's practically impossible!

Save the internet. Use regex.

Congratulations! You've just crashed the Internet.

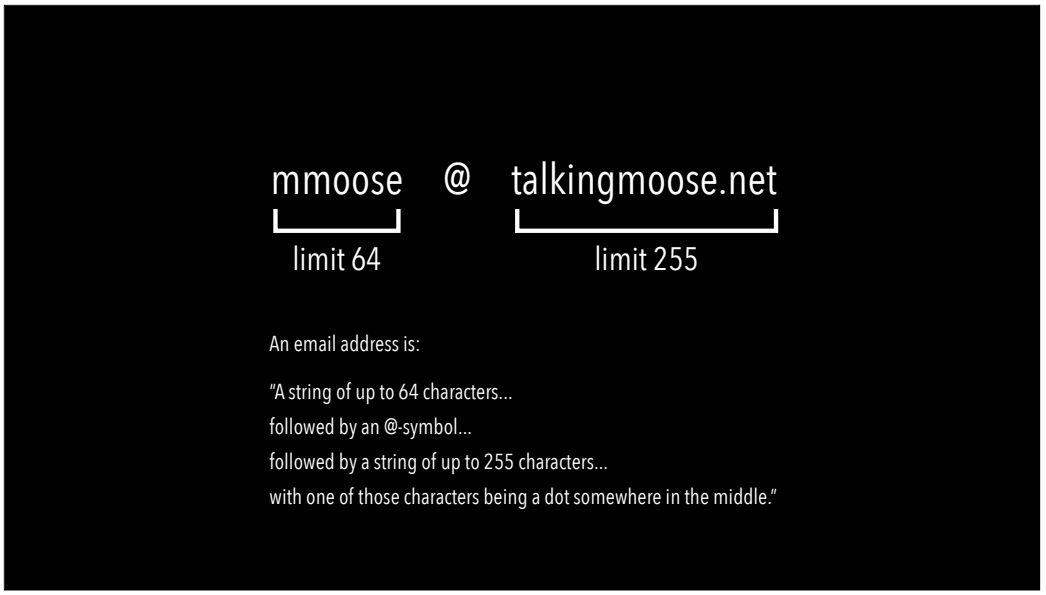
Not only is creating a database of all possible email addresses impractical, it's practically impossible!

Save the internet. Use regex.



We can all tell just by looking at an email address that it's an email address just like we tell by looking at a phone number that it's a phone number or by looking at a zip code that it's probably a zip code.

We're seeing patterns.



If we look at this email address it's easy to **say** the pattern. Tell me if you think this right or wrong.

"A **string** of up to 64 characters...
followed by an @-symbol...
followed by a string of up to 255 characters...
with at least one of those characters being a dot somewhere in the middle."

Does that sound fair?

Here's what a regex for that **might** look like.

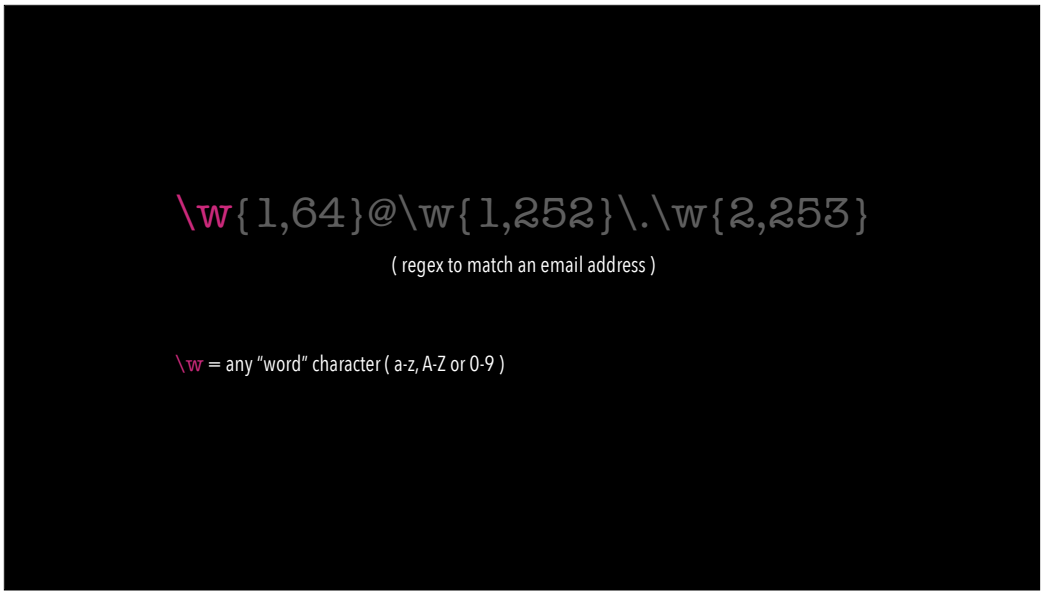
I'll be honest: it's easier to **write** a regex than it is to **read what it does**. As a favor to your future self and anyone else reading your regex, try to **include documentation** that explains what it's matching. You'll thank yourself later and so will others.

Let's break this down...

We haven't yet covered any regex symbols or shortcuts, but we'll be doing that shortly. For now, I'll step you through this one.

To read a regex, start at the left and identify what can. To make this somewhat easier to read, I'm going to color-code all the regex symbols I'm using. Anything that's white is just a literal character like "abc" or "123".

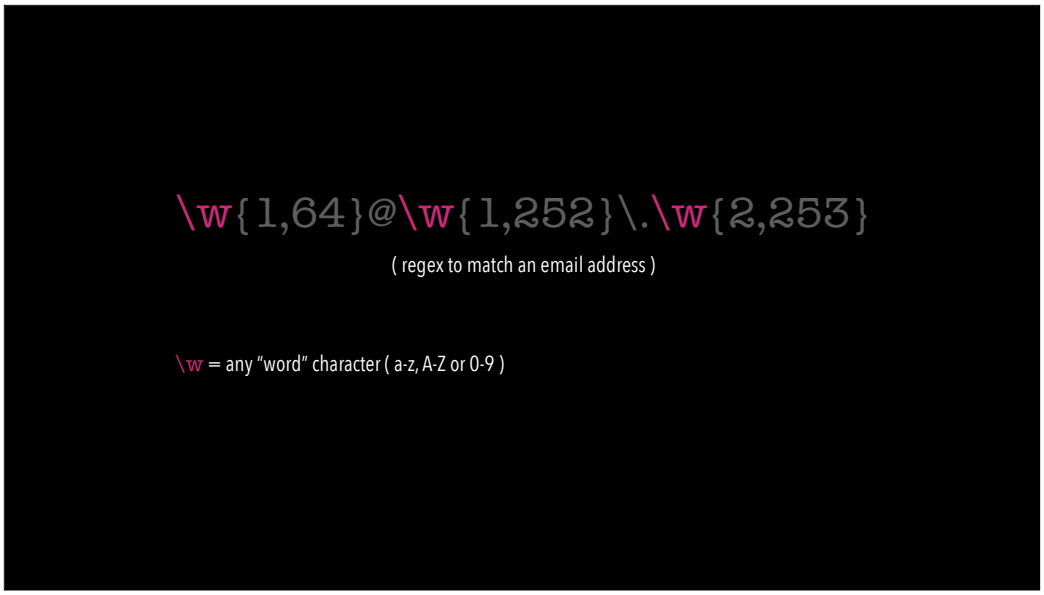
The **first item** "\w" is a regex shortcut. It means any single word character or literally any single character in the list of lowercase and uppercase letters as well as numbers.



We haven't yet covered any regex symbols or shortcuts, but we'll be doing that shortly. For now, I'll step you through this one.

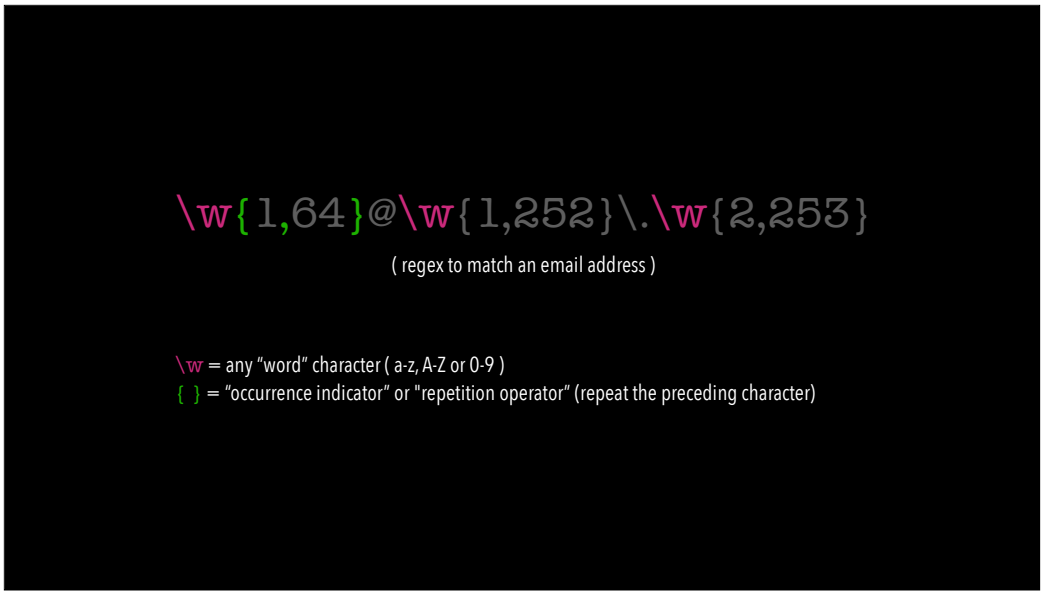
To read a regex, start at the left and identify what can. To make this somewhat easier to read, I'm going to color-code all the regex symbols I'm using. Anything that's white is just a literal character like "abc" or "123".

The **first item** "\w" is a regex shortcut. It means any single word character or literally any single character in the list of lowercase and uppercase letters as well as numbers.



You might notice this shortcut is appearing a couple more times.

So long as you see this shortcut by itself, it means **just... one... character.**

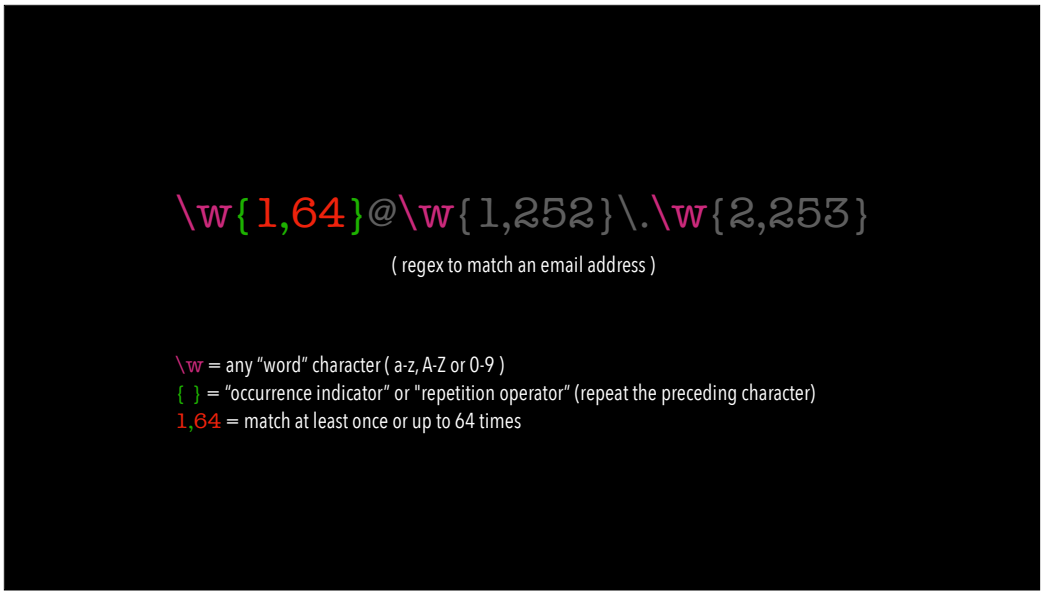


But here, it's followed by **curly braces** with numbers in them.

This signals something called an "occurrence indicator" or "repetition operator", which are just fancy names meaning that the character before it is going to repeat. **That means** I may have just one character or up to 64 characters. That's what those numbers mean.

So, think about your own email address and the part of it before the @-sign. Will **your** email address fit **this** pattern?

For some of you it will, but for others it won't.

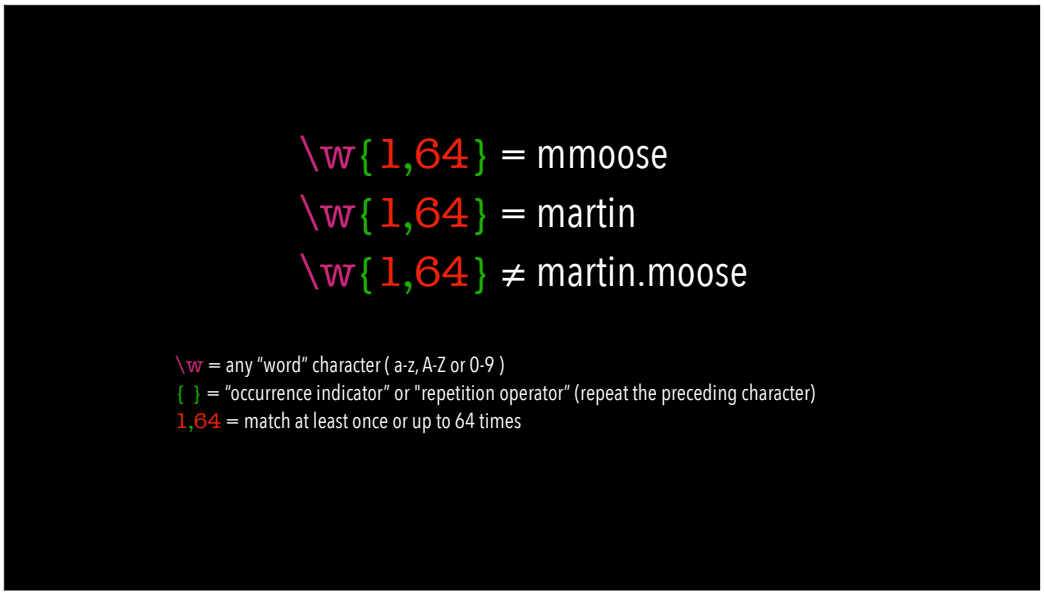


But here, it's followed by **curly braces** with numbers in them.

This signals something called an "occurrence indicator" or "repetition operator", which are just fancy names meaning that the character before it is going to repeat. **That means** I may have just one character or up to 64 characters. That's what those numbers mean.

So, think about your own email address and the part of it before the @-sign. Will **your** email address fit **this** pattern?

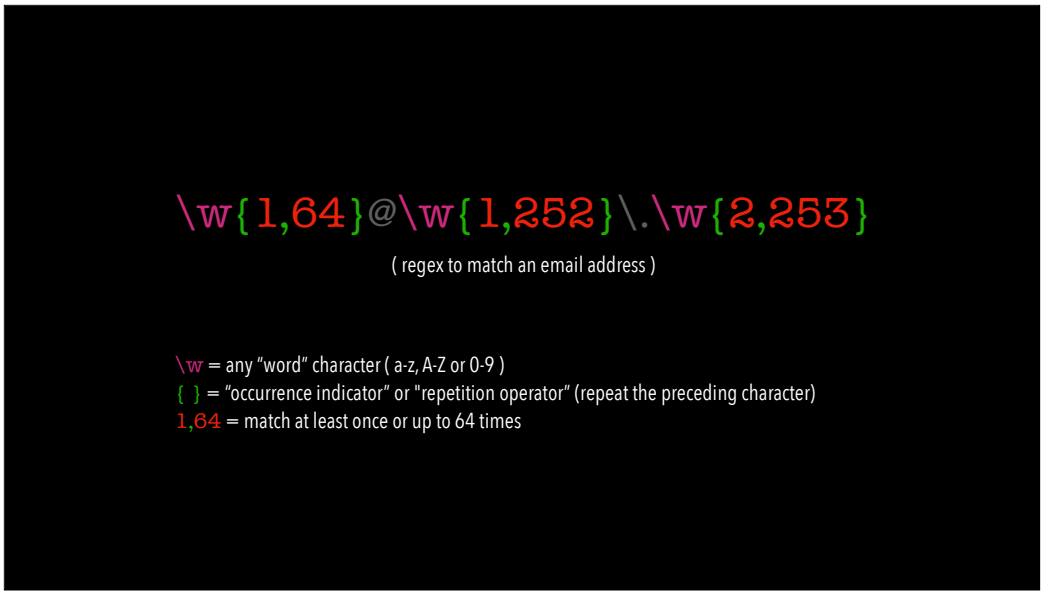
For some of you it will, but for others it won't.



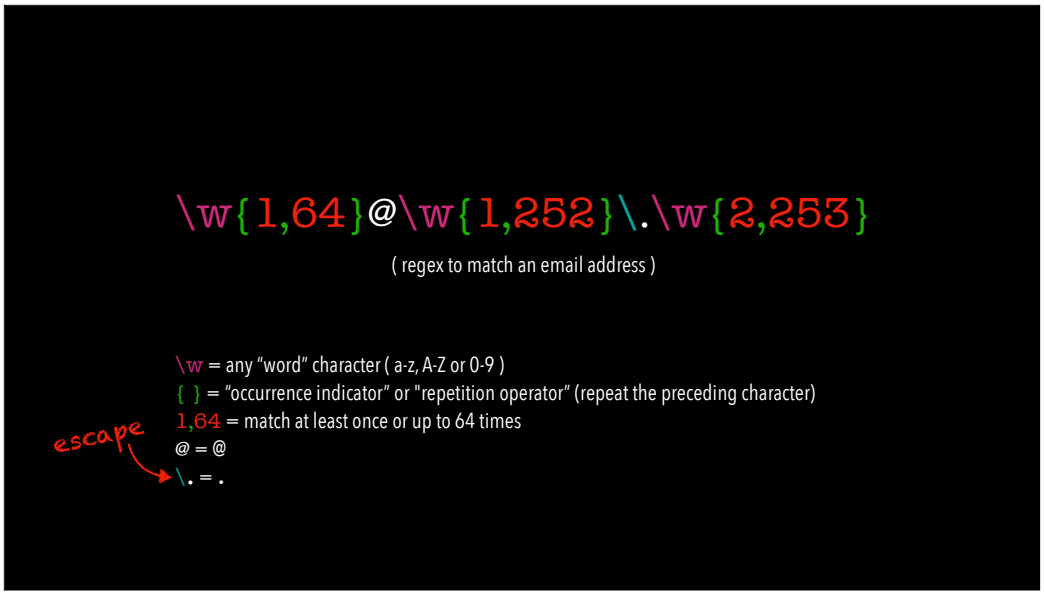
If your email address is in the form of **first.last**, you'll match the letters, but the "\w" matches **only** letters and numbers. **It doesn't match the dot.**

Keep in mind, this regex may be just fine for some of you because you may know that your company's email address **language** doesn't include dots in the username. But for the rest of you, you'll need to ensure your regex accounts not just for letters and numbers but also a dot... and maybe even other characters.

As you learn regex, you're going to create patterns that work for you but may not work for someone else. Don't be too concerned. But do be aware it may need to match more than you previously thought.



Again, you'll see there are two more sets of curly braces with a range of numbers following that "\w" character. These are defining the domain, which must have a period between the domain name and the top-level domain like ".com" or ".edu". Currently the longest top-level domain is only 24 characters, but nothing says it can't be longer. And, currently, the shortest top-level domains are just two characters.



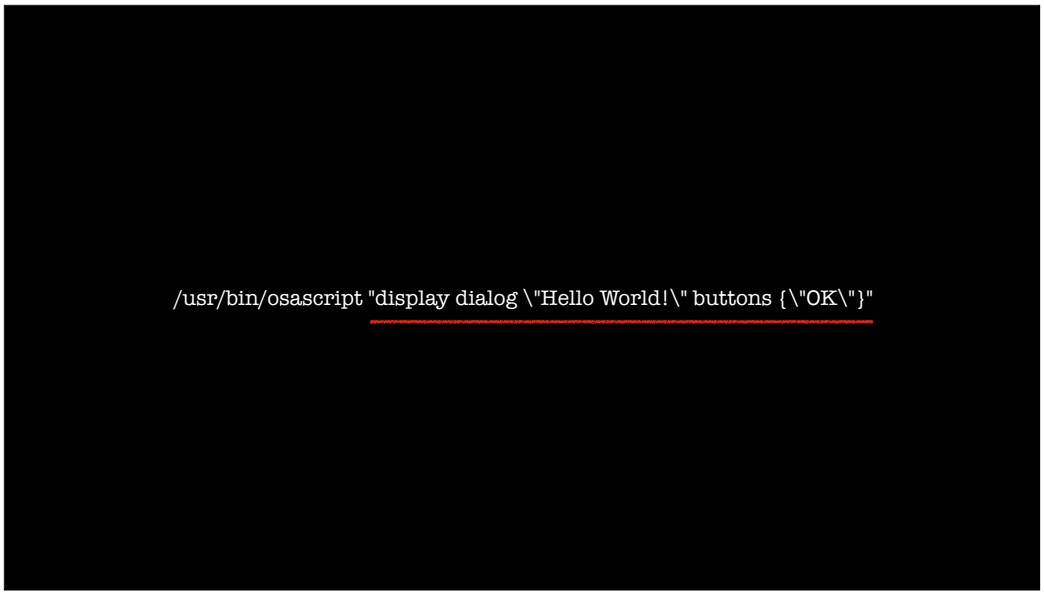
Finally, here's our @-symbol and here's our period. The @-symbol looks normal, but what's with the backslash in front of the period?

In regex, the @-symbol has no special meaning like it would in an email address. @ is just @. Nothing more exciting than that.

But a period in regex does have a special meaning. We'll talk about that in just a little bit. For now, if we want to denote a literal period, **we have to precede it with a backslash**, which is call an "escape". If you've done any scripting, you're probably already familiar with escaping certain characters.

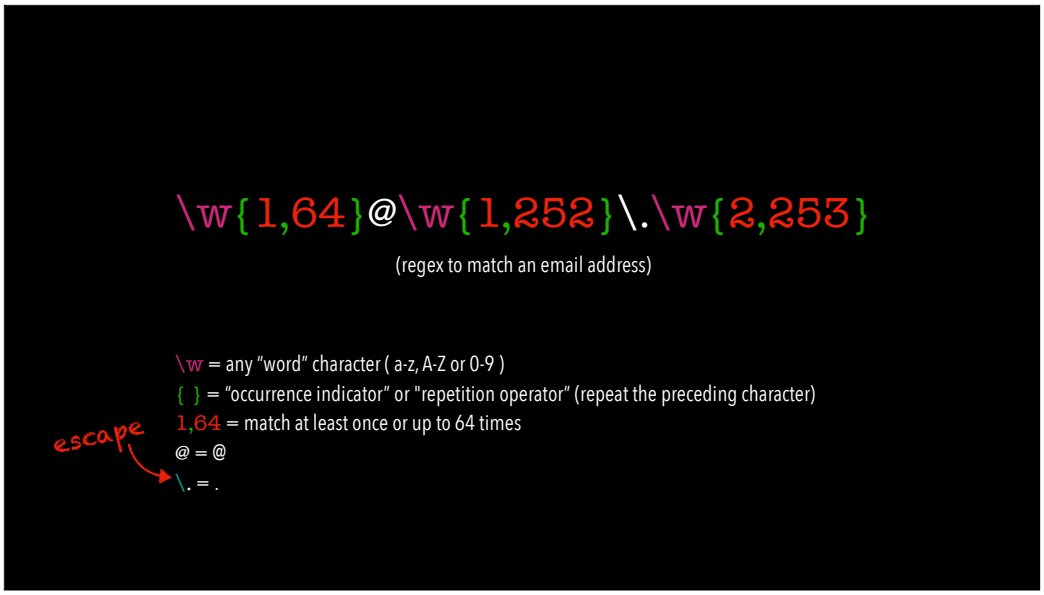
Here's an example where our script may require we put double-quotes **around an entire argument**, which is "display dialog hello world buttons OK". **But if we read that literally**, we get a fragmented set of arguments where some words or characters are in double-quotes and others aren't.

What we need is to use double-quotes at the beginning and the end to keep the **entire argument together** and to escape the inner double-quotes to indicate we literally mean just double-quotes.

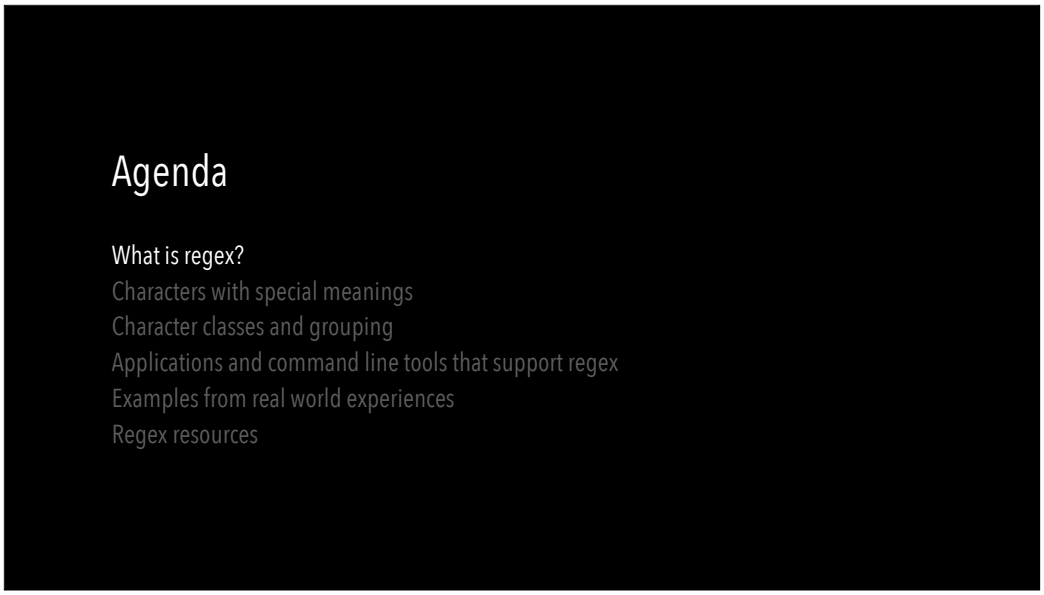


Here's an example where our script may require we put double-quotes **around an entire argument**, which is "display dialog hello world buttons OK". **But if we read that literally**, we get a fragmented set of arguments where some words or characters are in double-quotes and others aren't.

What we need is to use double-quotes at the beginning and the end to keep the **entire argument together** and to escape the inner double-quotes to indicate we literally mean just double-quotes.



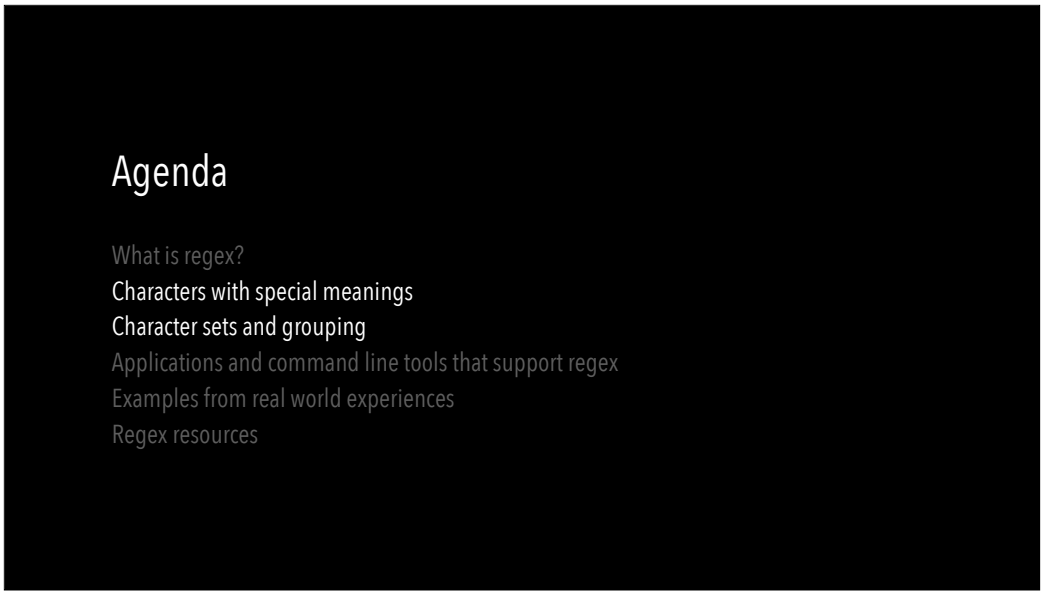
There are about about a **dozen or so** special characters like the period that'll need escaping if we want to use them literally and not as special characters.



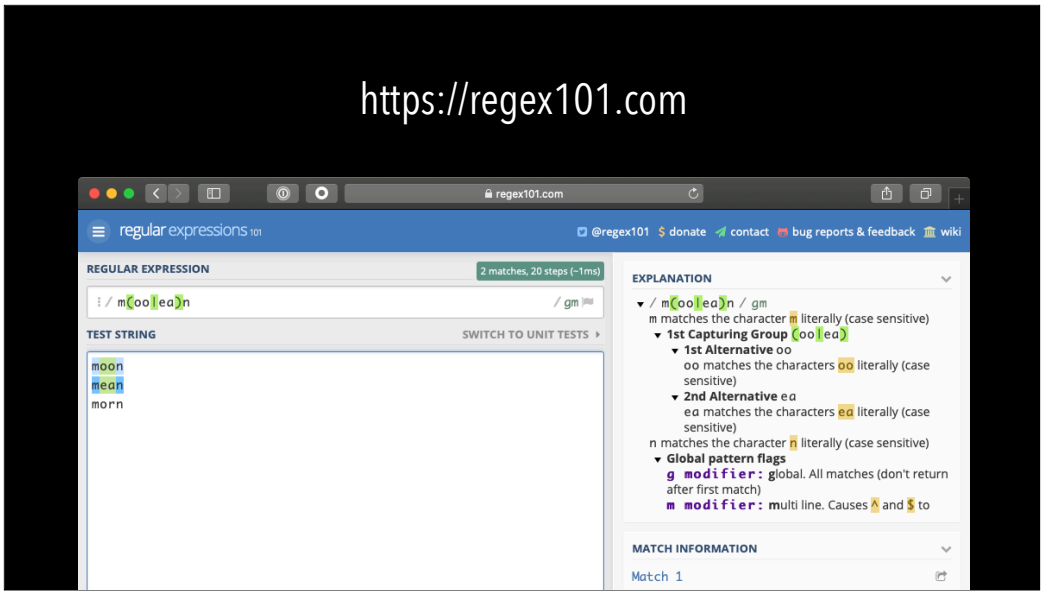
That's a short explanation of regex.

We're using it to **validate** whether a string of characters matches a pattern like an email address, a phone number or a zip code. If we validate text, we can find errors and correct them.

It's much easier look for a pattern than to try to create, understand or even store every possible permutation of a word in a language.



Next, let's look at the different types of characters that make up regex.



Before we get started, I'd like to point you to one of my favorite online tools — regex101.com.

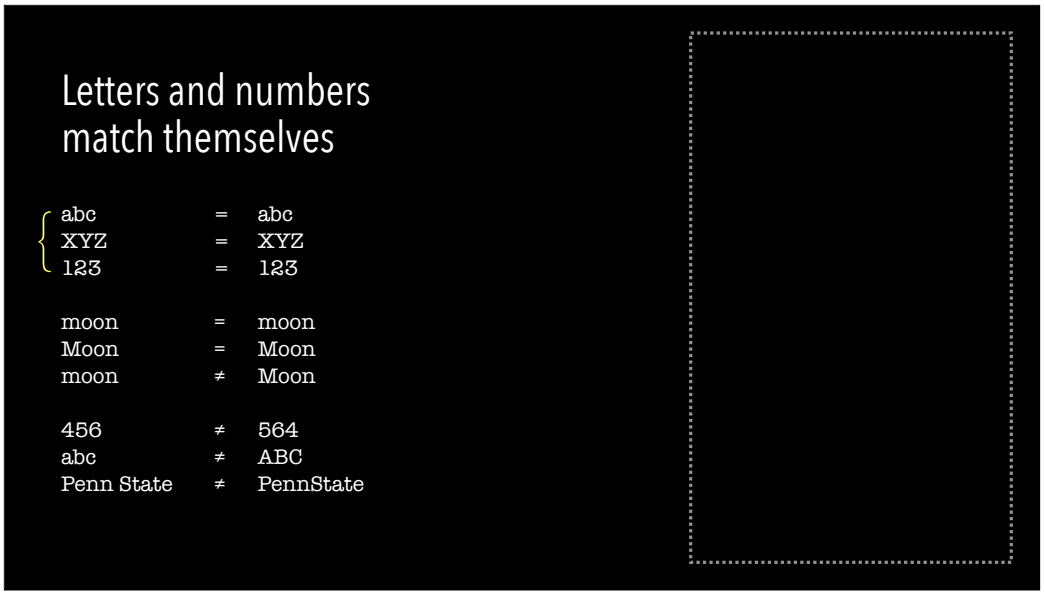
I'm getting ready to show you a lot of examples and if you find anything interesting, you can experiment with it here.

The top field is where you'll enter the regex and the bottom field is where you can type in one or several test strings to see if the regex matches them. The letters in your test strings will highlight as you complete your regexes. If the regex doesn't validate the test string, it won't highlight.

So, open your browser and play at home, if you like.

First, I'd like to say there is no shame in making a regex cheatsheet. In fact, I encourage it.

So, let's start one here on the right and we'll fill it in as we go.



In regex, letters and numbers match themselves.

And specifically lowercase letters only match lowercase letters and uppercase letters only match uppercase letters.

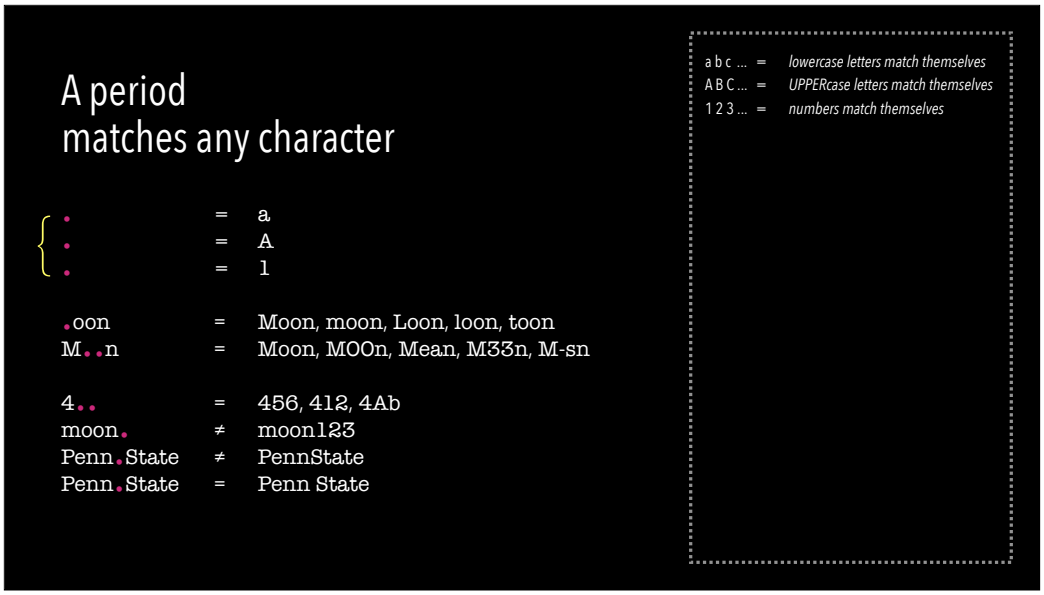
If we look at the middle section, the word "moon" all lowercase matches "moon" all lowercase and the word "Moon" with a capital M matches the word "Moon" with a capital M.

The word "moon" all lowercase, though, is not the same as the word "Moon" that begins with a capital M. This doesn't validate.

Likewise, the string 456 doesn't match the string 564 and, therefore, it doesn't validate either.

And looking at the last example, "Penn space State" doesn't match "PennState" without a space.

Simple enough?

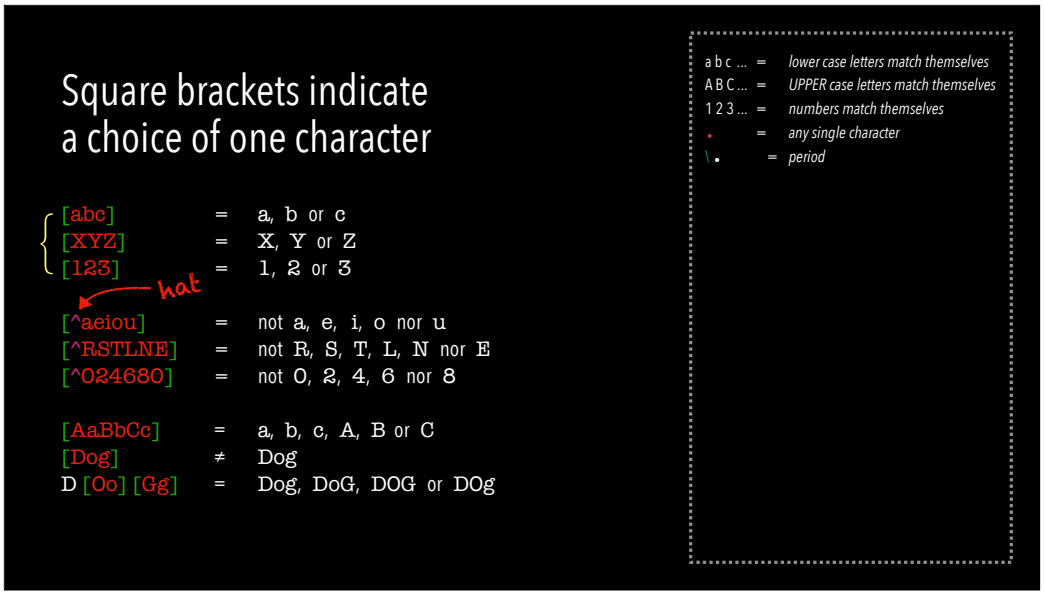


Next, **a period** matches exactly one character. *(If you hear me say dot, I still mean a period. I sometimes use those terms interchangeably...)*

It can match any lowercase letter, uppercase letter, a number, a symbol, a space... **anything**.

Note in the second "Moon" example where the regex is capital M, dot, dot, n, the two periods mean exactly two characters whether they're letters, numbers, symbols or even a space.

And for the two Penn State examples at the bottom, the regex is "Penn, dot, State". That means it'll match "Penn State" with a space in the middle but still won't match "PennState" with no space. A dot means there **must** to be a character.



Before we get into this one, I'd like to point out that in our cheatsheet on the right, I've added both the period to mean any single character and I've added the backslash-period to mean a literal period. If I were creating a regex to validate an IP address, I'd use backslash-period to ensure we're matching the dots between each octet.

Moving on to square brackets... These indicate a "character class" or better yet a **"character set"**. **When you see square brackets**, you have a choice of any one character inside — lower case a or b or c. Or upper case X or Y or Z. Or number 1 or 2 or 3.

When you add the caret symbol to the beginning of the list, that means "not". *You might also **hear the caret symbol** referred to as the "hat" symbol — h-a-t.*

- In these examples, it means not vowels or anything **but** vowels
- Not R, S, T, L, N nor E or anything **but** R, S, T, L, N or E
- And not **even** numbers or anything **but** **even** numbers

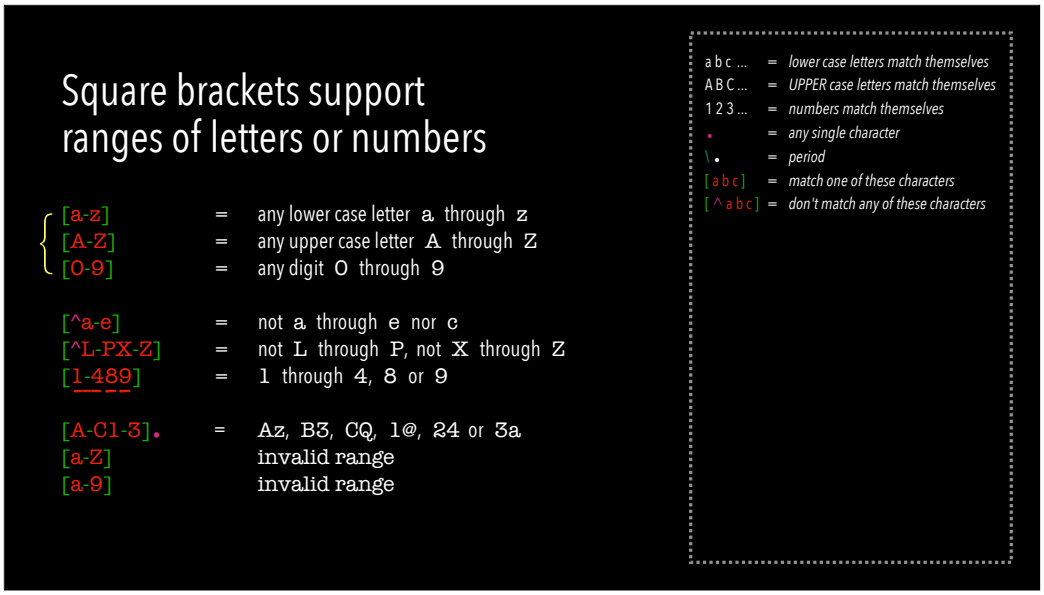
Say this to yourself in your head any way that you like to help you remember what it means.

In the first example here, your choices are upper and lower case letters a, b and c — just one those, though.

Look at the second example with the word "Dog" in brackets. **Who can tell me why it DOESN'T match "dog"?**

Because you can only match **one** character. It'll match uppercase "D" **or** lowercase "o" **or** lowercase "g" but not all three.

And in the last example, we have a regex that matches four different versions of "dog". Remember, an upper case letter by itself matches itself, so capital D followed by either a capital or lower case "o" followed by either a capital or lower case "g". It **won't match** any version of dog that begins with lower case "d".



We're not limited to individual characters within square brackets. We can specify a range of letters or numbers to keep our character sets short.

- lower case "a" through "z"
- upper case "A" through "Z"
- and numbers 0-9

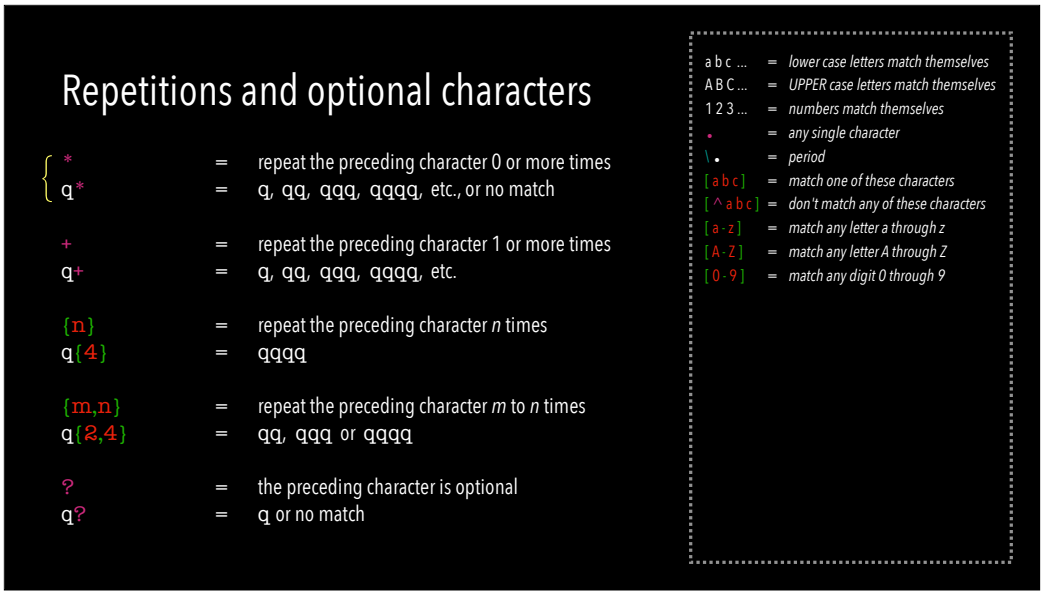
And our ranges can be just partial ranges. If we want to exclude the letters "a" through "e" using the caret symbol at the beginning of the set, we can do that.

Also notice we can specify multiple ranges like not "L" through "P" and not "X" through "Z". This regex still matches just one character, but just not those capital letters.

And don't be fooled if you see a character set that appears to have a large range like 1-489. Remember, we're matching **just one character** from inside the square brackets. This is really "1 through 4" **or** 8 **or** 9.

Last, we can mix alpha ranges with numerical ranges and single characters too as we see in the first regex here. A match must start with capital "A", "B" or "C" or number "1", "2" or "3". The period at the end matches any character.

Be sure not to mix up your ranges, though. There's no such range as lower case "a" through capital "Z" and there's no such range as a letter through a number. That makes sense if you think about it.



Like the period, the following characters are special in regex and it's easy to get them confused. I put them all together here because they all behave in the same way. They modify the preceding character in some way:

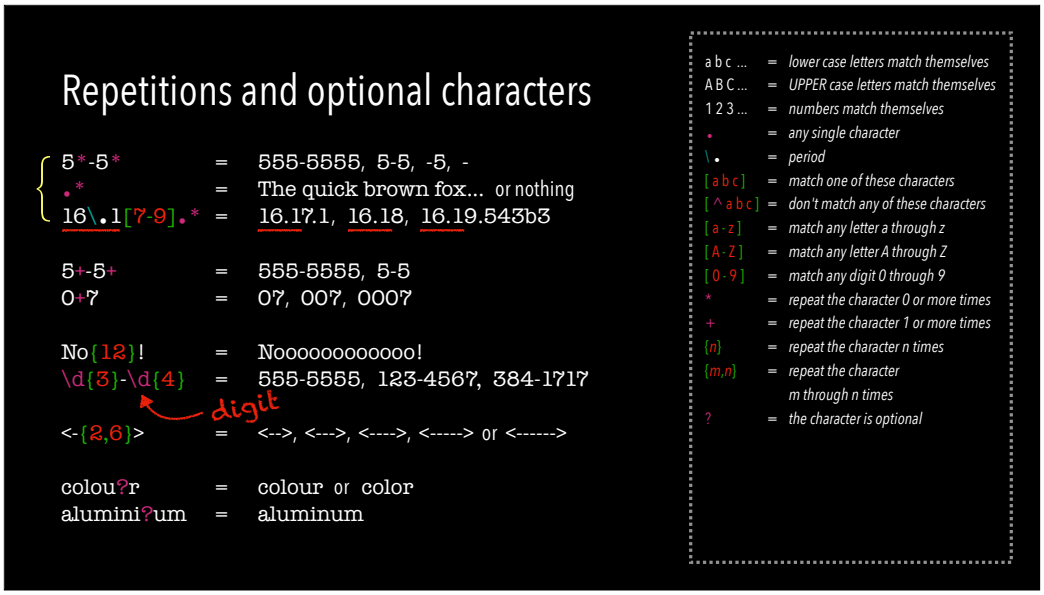
- **When an asterisk follows a character**, it repeats that character 0 or more times. For example, if you ever see the letter q followed by an asterisk, the regex will match a single letter "q", two "q"s, three "q"s, fifty "q"s or no "q"s in the pattern. An asterisk matches 0 or more times.

- **The plus symbol** does almost exactly the same thing as the asterisk, except that it must match at least one time. So, 1, 2 or fifty "q"s, but there has to be at least one.

- **If curly braces** with a number follow a character, that means match this character exactly this number of times.

- **If curly braces** with two numbers separated by a comma follow a character, the regex will match any repetitions from the smaller to the higher number. In this case, it matches two "q"s or three "q"s or four. No more. No less.

- And if a **question mark** follows a character, that means the preceding character is optional. If it's in our pattern that's great! If not, no big deal.



So, what are some examples using these characters?

- **An asterisk indicates 0 or more repetitions.** This first regex will match something that looks like a phone number, something that looks like sports score, something that looks like a negative number or just a hyphen.
- The second example is really interesting and you're going to see it used a lot in regex. Remember, the period indicates a single character. When it's followed by an asterisk, that effectively means match any length string you want from 0 characters to a sentence or something longer. Be on the lookout for this one.
- The last example might be a version number. **Every match will begin with "16.1".** We're escaping the period because we literally mean a period is the next character – not a special regex character. Then the string is followed by 7 or 8 or 9 followed by zero or more characters. Remember, those characters can be numbers, letters, symbols or spaces.

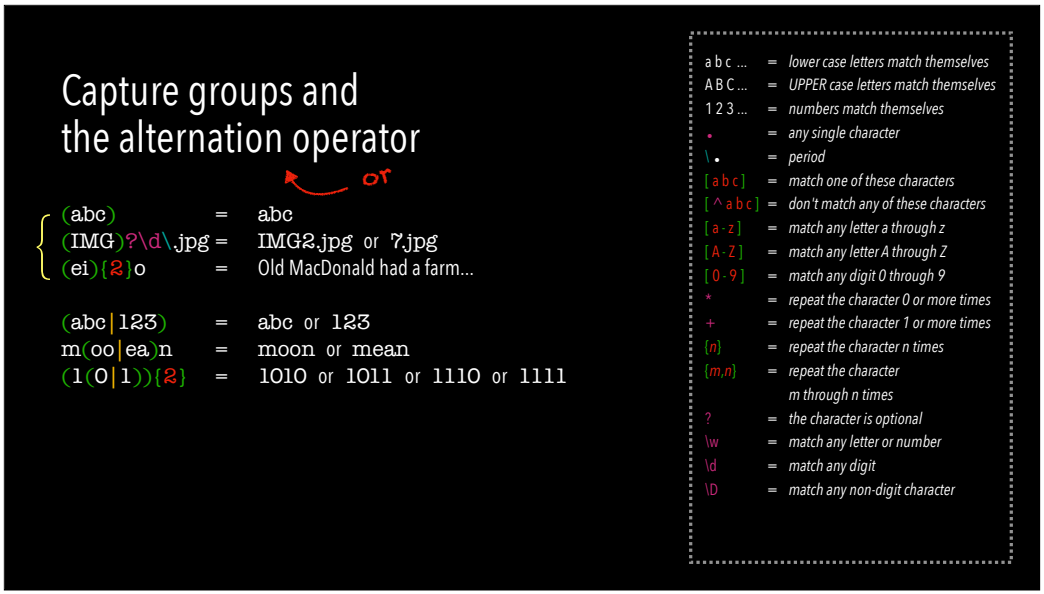
- **If we change the asterisk to a plus sign,** we're altering just slightly what we'll match. The 555 phone number still matches as well as the 5-5 sports score, but now at least one "5" on either side of the hyphen is required.
- And a 0 followed by a plus sign assures that whatever match we have, it must have a leading zero.

- **Now**, if we want to be specific about the **number** of matches, we just need to put a number in curly braces after the characters. Here, we have the Darth Vader "Noooooooooo!" regex with 12 "o"s.

- **In the second example**, I'm throwing in something new — the "\d" shortcut. Earlier, we saw the "\w" shortcut, which stood for any letter or number. "\d" stands for only a number "0" through "9". If I want to specifically match a phone number, my regex must match three digits, hyphen and four digits.

- **Here**, the curly braces follow a hyphen and possible matches will include **at least** 2 and **up to** 6 hyphens only.

- **Finally**, there can be humor in regex. *It may not be funny to everyone, but it's there.* A question mark means "optional". While some of us may disagree on the spelling of the word "color" with the optional letter "u", regex can remain neutral and accommodate **everyone**. And where some folks like to say "al-yoo-min-e-um" while others say "al-oom-i-num", regex will always say it **correctly**.



Now, that I've introduced you to the "digit" shortcut "\d", I've added it to the cheatsheet along with its counterpart "\D", which matches any non-digit character. There are a lot of these shortcuts — too many to cover today. But knowing just the handful here that I'm showing you will go long way to accomplishing what you need to validate strings.

The last two items I'll cover before we look at some examples are capture groups and the "alternation" or **"or"** operator.

- **Until now, I've been telling you that we're matching just one character at a time.** But what if you wanted to match multiple characters? Parentheses around two or more characters make a group and that group will be taken in whole. By themselves, they're not very interesting because something like "abc" in a group is the same as just "abc" without the parentheses.
- But combine capture groups with regex special characters as you see in the second example, and you can do some pretty interesting things. For example, you can find numbered files that may or may not begin with "IMG". Here, the entire IMG string is optional not just each letter.
- Or in the third example, we can use regex to complete "Old MacDonald had a farm..."

- **Now**, let's talk about the "alternation" operator — the vertical bar or "pipe" symbol, **which we all know** as just "or". It's really

only useful in capture groups and it lets us choose one string from the group and treat it as a unit. In the first group we have a choice of "abc" **or** "123".

- When we put it in context with other characters, we can match multiple strings like moon or mean without having to explicitly write out "moon" or "mean".
- And we can even nest groups within groups. In the third example, we have a capture group of "0" or "1" inside another capture group that begins with "1" and then we say repeat it two times. That gives us four possible matches — "1010", "1011", "1110" and "1111". If this seems confusing, it can be. As I said earlier, it's easier to write a regex for your own needs rather than read someone else's.

But let's take a stab at it.

We have a full cheatsheet on the right, but it's by no means exhaustive of the characters you'll find in regex. However, these few conventions and shortcuts will take you a long way.

The best way to learn regex is to practice, so let's build a few. If you're using the regex101.com site I showed you earlier, enter the examples I give you here for testing and then start building your regex in the top field.

Here's a list of model identifiers for MacBook Airs that support macOS Catalina. What would a regex for this look like? I'll give you a few seconds to think about it.

...

The first thing I would do is start at the left and try to find all the leading characters that each string has in common. **That's pretty easy.** Each string begins with "MacBookAir".

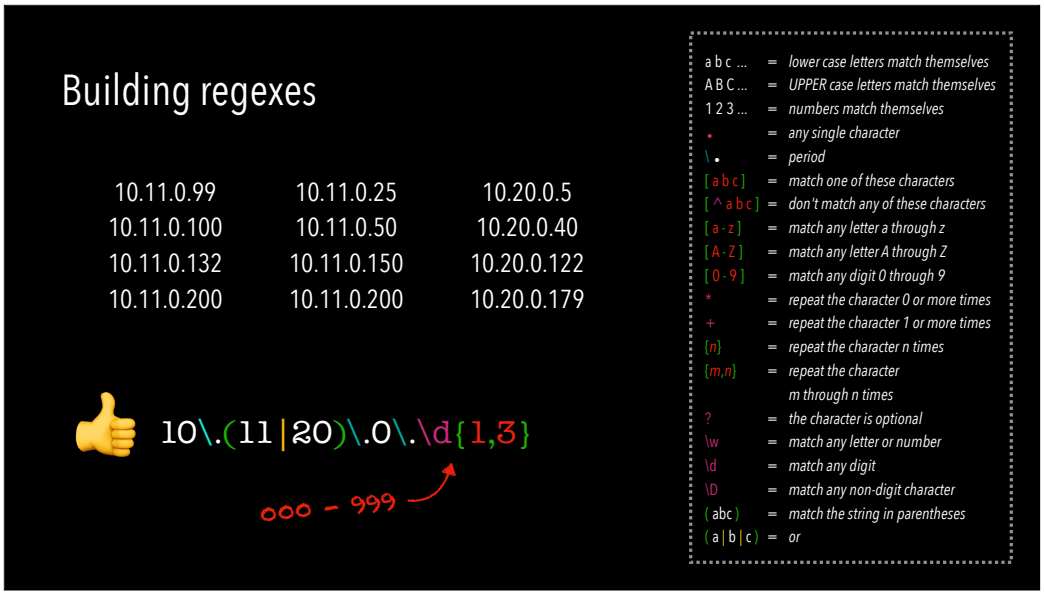
The next character in each string is a single number, but they're not all the same. However, they are all in a **range of "5" to "8"**.

So, I'd use square brackets to define that range.

Next in all the strings is a comma. When it's by itself and **not in any brackets or braces**, it's just a comma.

And each string ends with a number — either "1" or "2". I could **write it this way** using square brackets. **Or I could write it this way** using parentheses as a capture group. Both work. There's really no right or wrong other than I'm trying to write as efficient a regex as possible and that usually means shorter. I like the top one better.

We'll take a look at more model identifiers in just a little bit.



Here's a list of Class A IP addresses and I need to match any IP address on these networks. And let's assume we have a simple subnet mask of 255.255.255.0. What would a regex for this look like? I'll give you a few seconds to think about it.

...

Starting at the left to find all the leading characters that each string has in common, **we'll start our regex with 10\.** I want a literal period not a special regex "match-any-character" period.

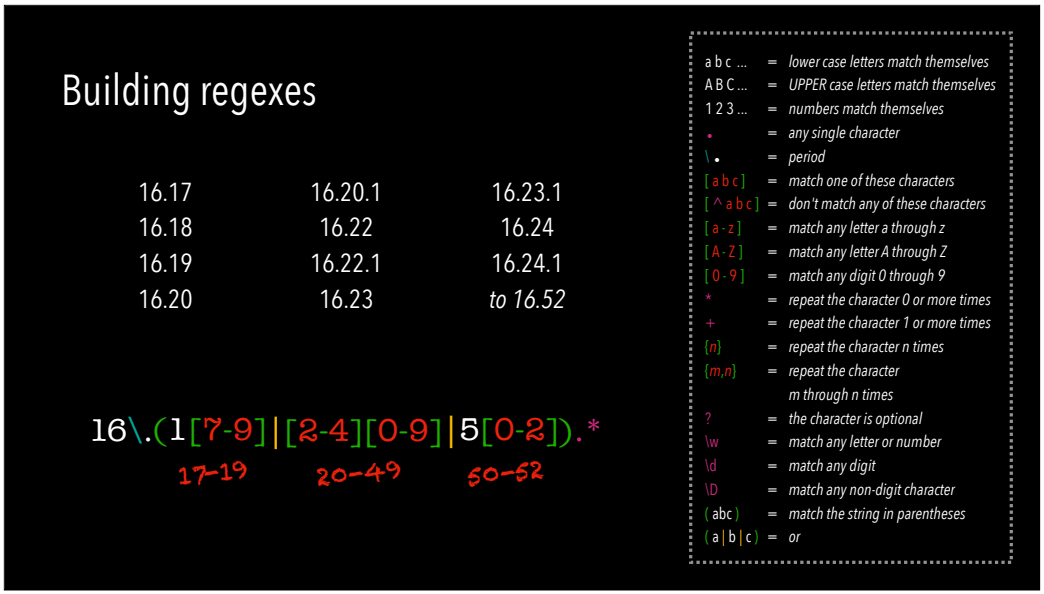
The second octet is either "11" or "20", so I'll use a capturing group for that.

After that, all the addresses continue with ".0.". Again, I'm escaping the period because I want a literal period.

And then I need to **match any IP address** on these networks, so the **digit** character with a range of 1 to 3 digits works just fine.

Now, think about the regex for that last octet for a moment. **What will it really match?** It'll match "000" through "999" but octets

in IP address only go to 255. Is this a problem? Well, it depends on what you're trying to accomplish. If I'm using this regex to validate a form field that someone's filling out online, I probably need to be a lot more careful that the last octet doesn't go past "255". But if another system is simply feeding me a list of IP addresses that it generated and I'm simply matching those, then I don't need to worry. It's the other system's responsibility to make sure it's feeding me valid IP addresses. I'm simply making sure they're on the right network. This is one of those situations **where sometimes** a simpler regex is simply good enough.



Here are some Microsoft Office **2019** version numbers starting at "16.17". For the next 2-3 years, these will count up to about 16.52. Microsoft Office **2022**'s starting version number will start at 16.53. So, my goal is to identify any Office 2019 version. Also notice a few of them have ".1" updates too.

What would the regex for that range look like? I'll give you a few seconds to think about it.

...

It starts off easy with matching "16." But now we have to match "17", "18" and "19" and they all begin with "1". We also have to consider, though, the numbers in the "20"s, "30"s, "40"s and a few of the "50"s **and then stop**. So, it's a range of "17" through "52". How do we do that?

We break up those numbers into groups. **So, to start a group**, we add an open parenthesis.

To start the group, I'm interested only in those minor version numbers that won't use a full range of numbers from "0" to "9". So,

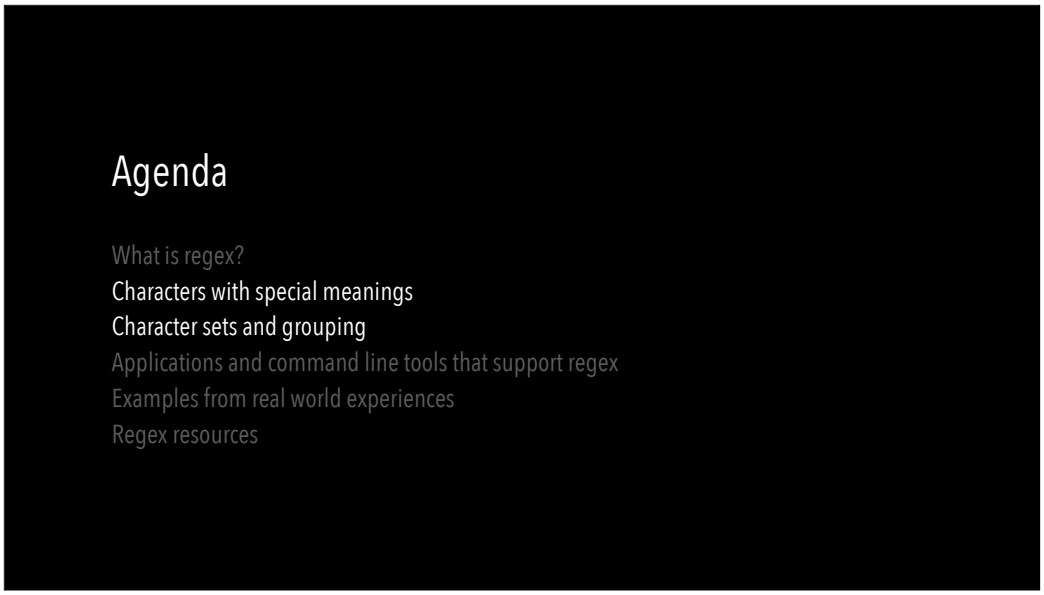
I'm looking very specifically at "17", "18" and "19". **I can match the number "1"** in all of those and use a range for "7" through "9". That let's me match versions "16.17", "16.18" and "16.19".

Now, I'm thinking about the next range for the "16.20"s, "16.30"s and the "16.40"s. **I can handle those using two sets of square brackets.** The first brackets match the "20"s, "30"s and "40"s. The second brackets match "0" through "9". Now, I have a range of "20" through "49".

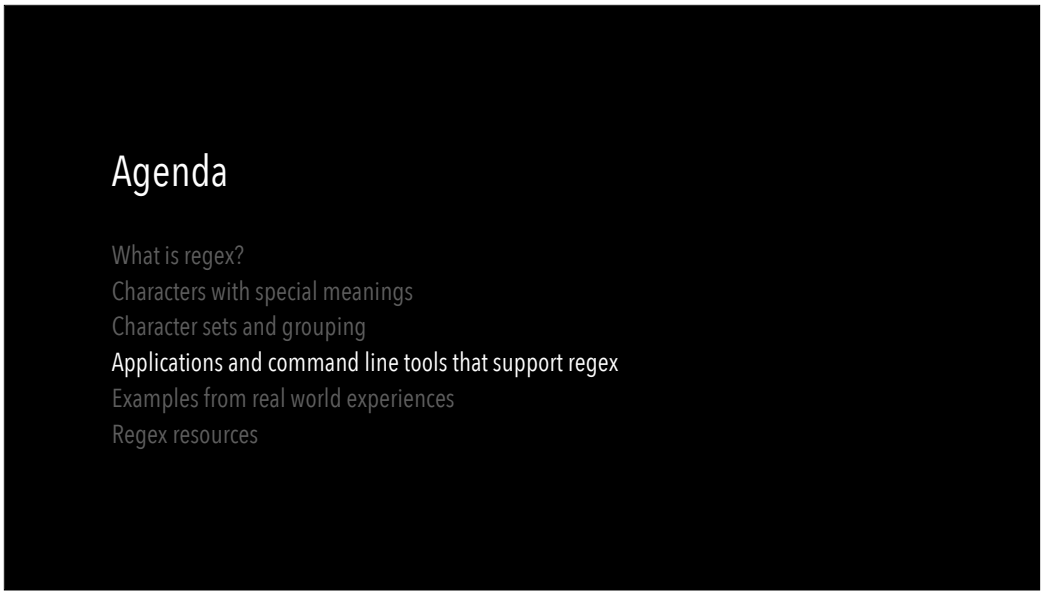
Lastly, I need to match "50", "51" and "52" and I'll do it the same way I matched "17", "18" and "19". **I can match the number "5" in "16.52" literally** and then use square brackets to create a range of "0" to "2".

Now, I close my capturing group and this gives me three possible matches: **"17" through "19" or "20" through "49" or "50" through "52"**.

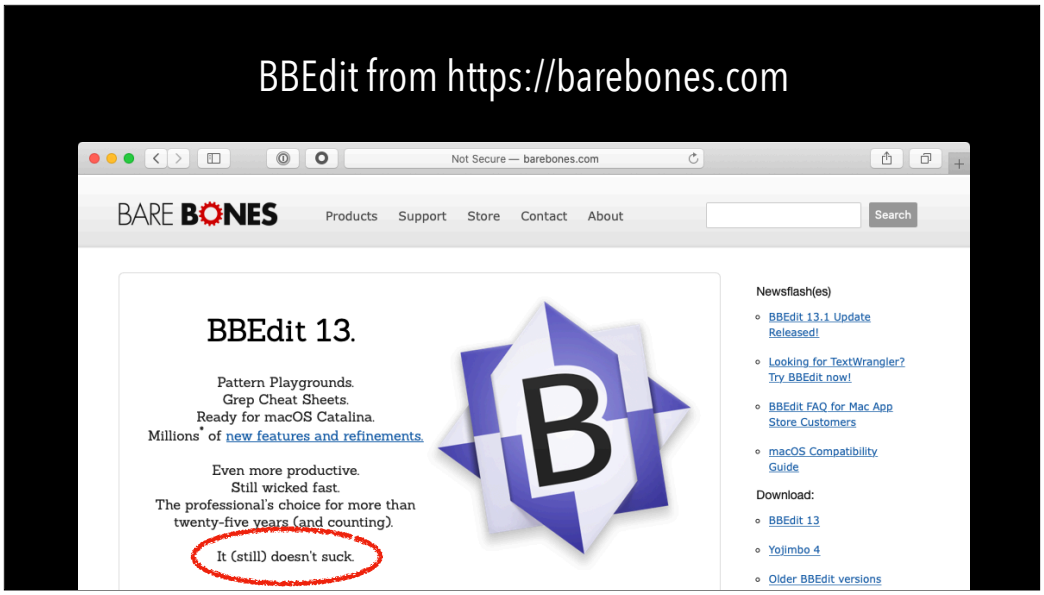
And as far as some of those version numbers having a ".1" update, **I can handle that** with a ".*" to match "0" or more characters at the end. If there's a ".1" or ".2" or whatever, great! If not, no big deal.



That was a lot of detailed information thrown at you very quickly, but just like we had to learn the letters of the alphabet before we could make words, we need to learn at least a **few** regex symbols and shortcuts to make words in regex. The difference, though, between learning our ABCs and regex, is that learning more symbols and shortcuts will actually allow us to write **shorter** regex strings not longer. Anything that's shorter is going to be easier to read and understand.

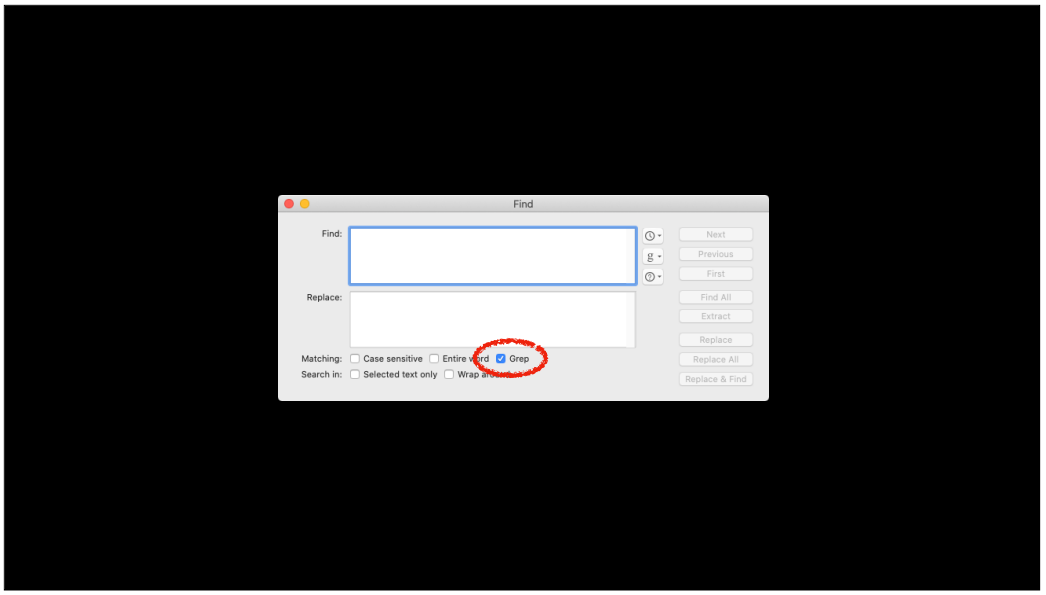


Now, let's take a quick look at where you can use regex. I'll show you a few applications and command line tools that support it.



I give BBEdit from the folks at Bare Bones Software a lot of love because I love their product. It's a very powerful plain text editing application with a very simple interface.

I use to write and test all my scripts, edit XML and JSON files and sort and massage all kinds of information. And even though it's been around for nearly 30 years — version 1.0 was released in 1992 — and it's now up to version 13, **it (still) doesn't suck.**

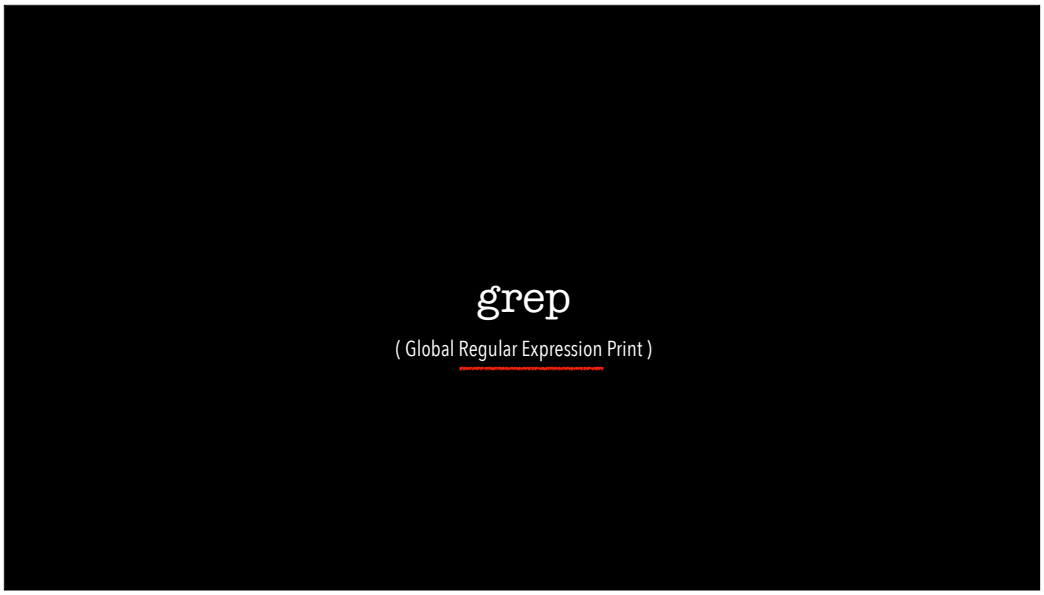


Its Find/Replace feature includes an **option to use regex**. It's called grep.

Now, some of you who have done some scripting may be thinking, "Where have I heard that before?"

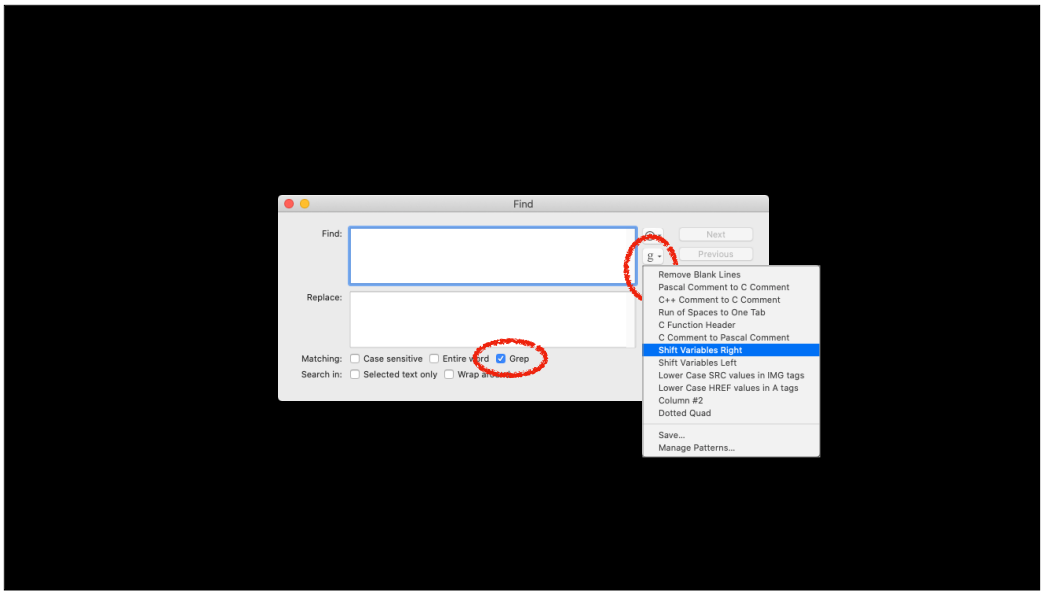
You've likely used it in many of your scripts to "grep" for something. Maybe you've used cat to read a file and then piped that into grep to find all the lines that include "dog".

(By the way, for those of you who are familiar with the ***useless use of "cat"*** *here in this command, I did that on purpose. For the rest of you, I'll explain what I mean a little later.)*



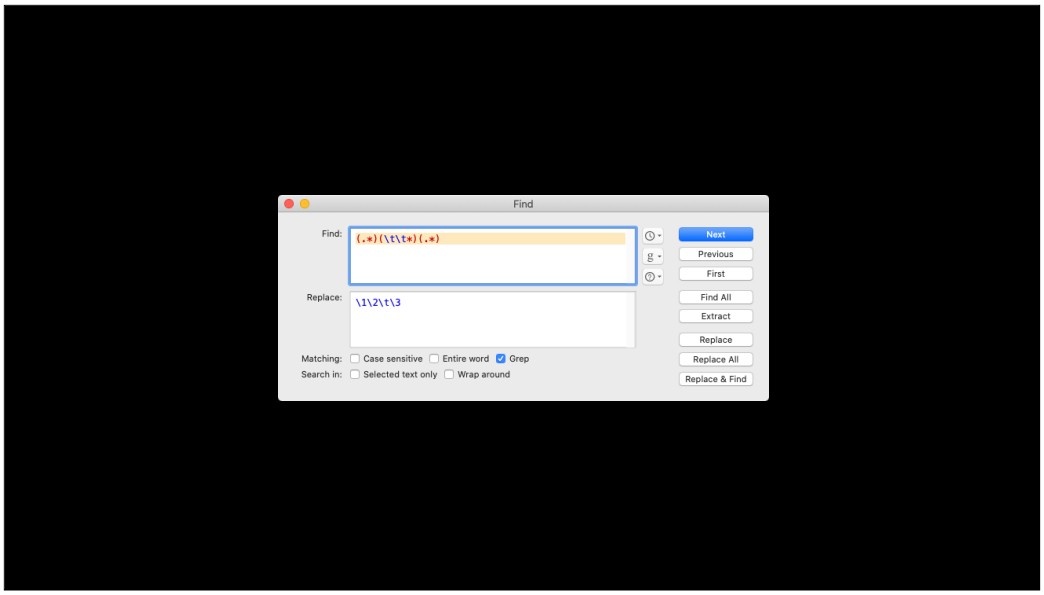
The name **"grep"** comes from "Global Regular Expression Print" or more understandably "globally search for a regular expression and print matching lines". What you may not have **realized** is that the "re" in grep stands for "regular expression".

There's a high likelihood that you've been using regular expressions all this time without realizing it.



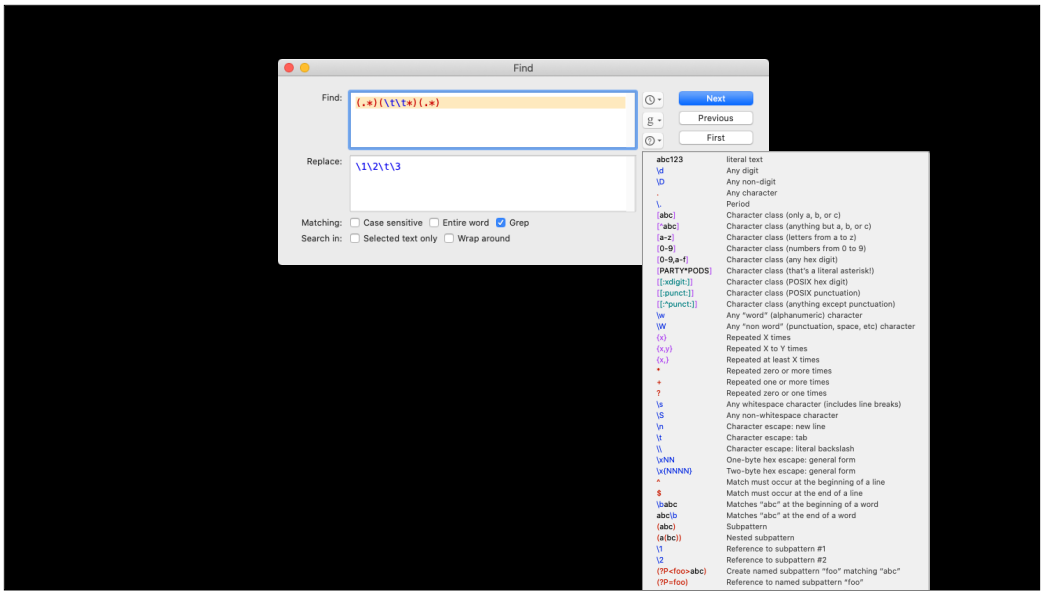
Don't worry if you're new to regex or just now digging into regex **because BBEdit includes** these two handy buttons.

Clicking the first one reveals a menu of built-in patterns, which you can edit to add your own.



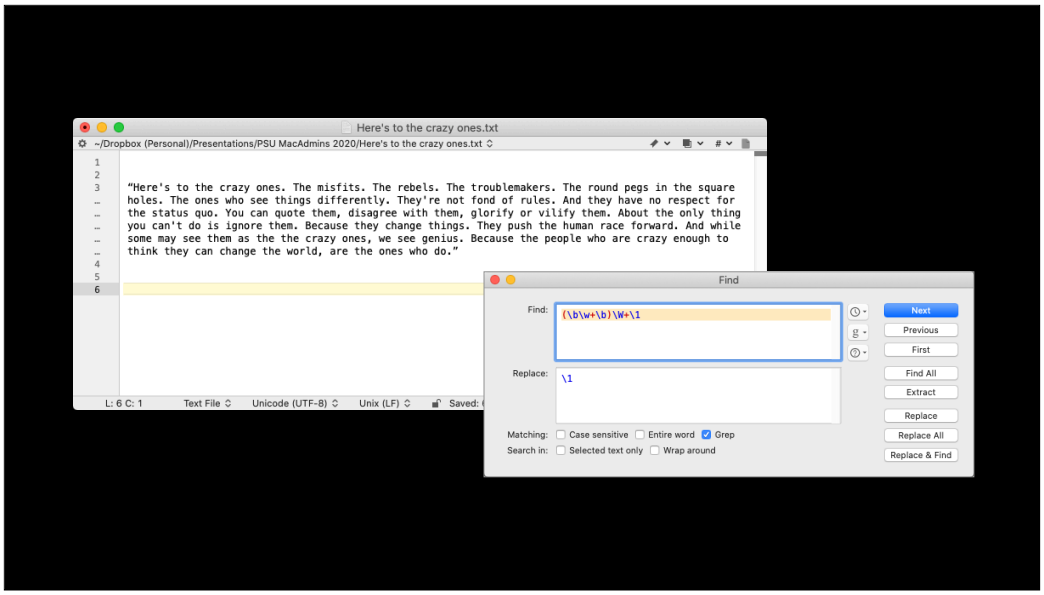
It'll fill in the regexes for both the find and the replace strings. All you have to do is supply the document.

And if you need help with some of the characters when making your own regexes, **clicking the help button** reveals BBEdit's own cheatsheet.

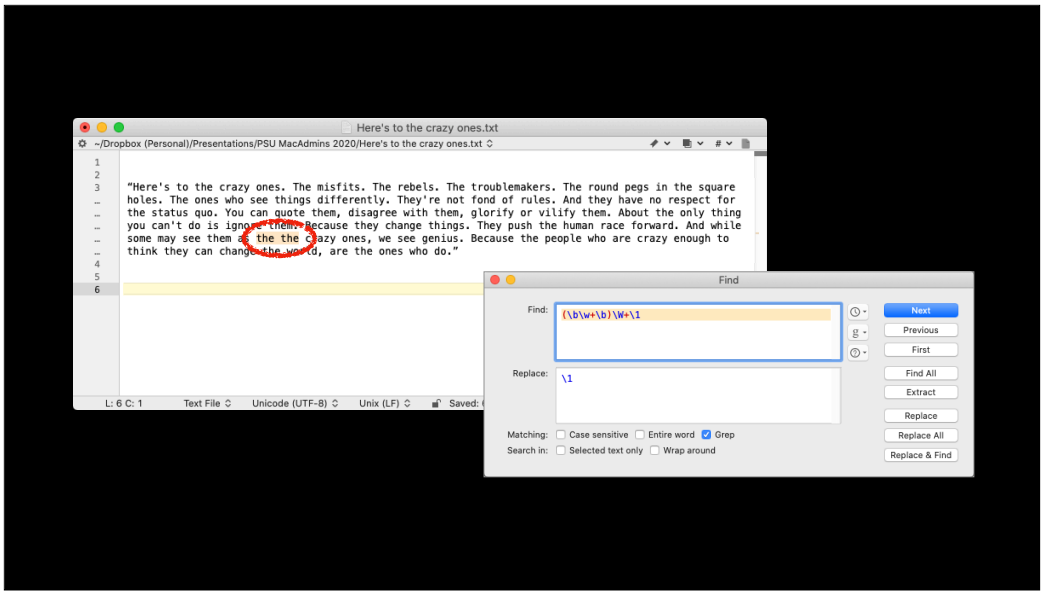


It'll fill in the regexes for both the find and the replace strings. All you have to do is supply the document.

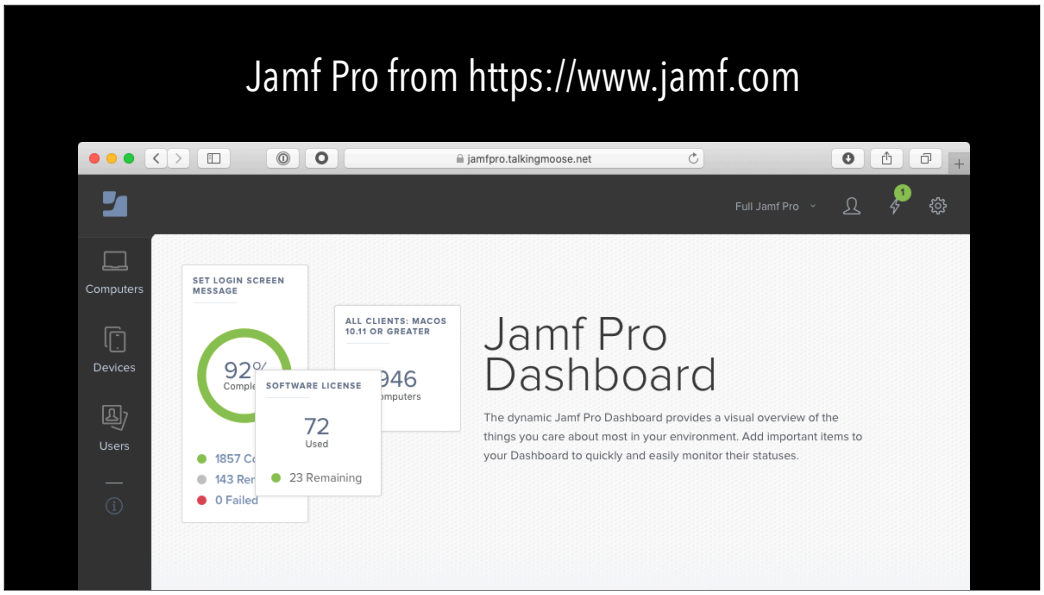
And if you need help with some of the characters when making your own regexes, **clicking the help button** reveals BBEdit's own cheatsheet with far more examples than I included in mine.



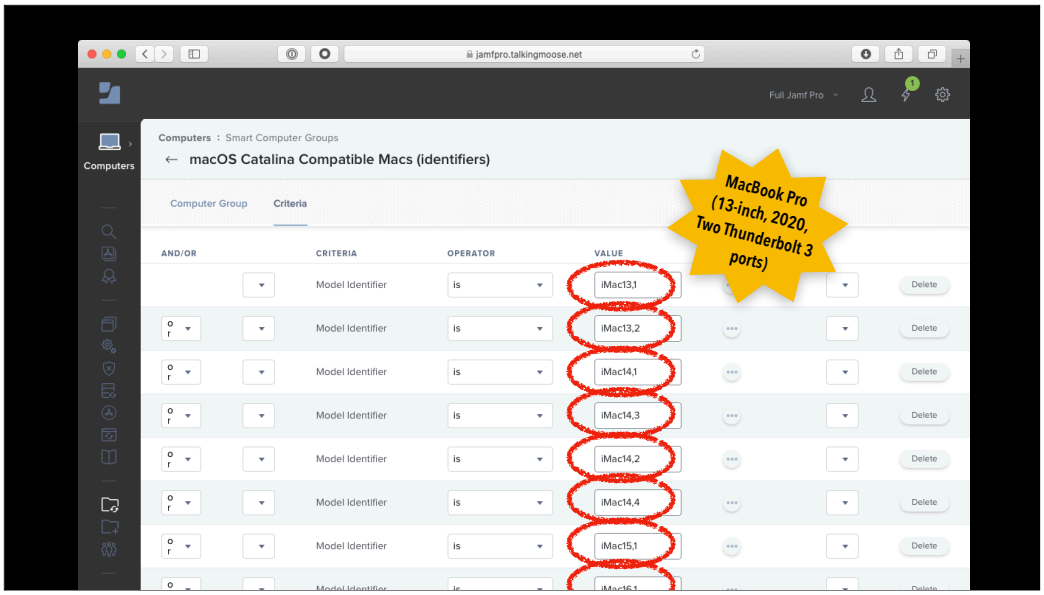
One of the interesting things you can do with regex in BBEdit or most any application that supports regex is take a pattern like this and run it against a bunch of text.



And when you do, **it'll help you find** doubled words that you **can't easily see** when reading through the text. The backslash-one you see in the Replace field is actually a shortcut referencing back to the capture group in parentheses in the Find field. This search is effectively saying, "find all doubled words and replace them with single words".



Another application that many of you know is Jamf Pro, which is a management system for Apple Devices.



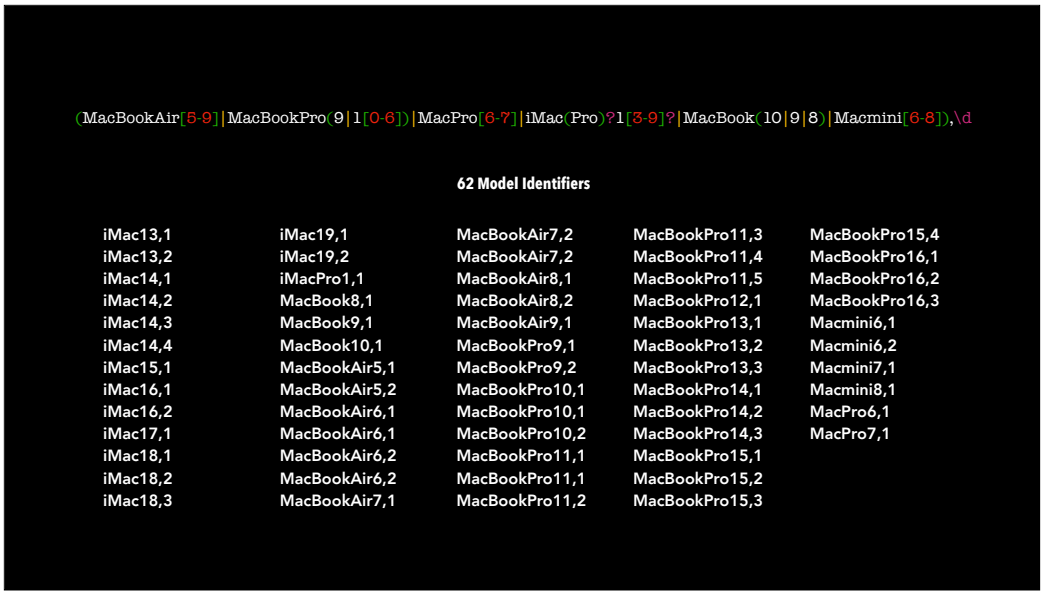
It has two features — smart groups and advanced searches — that allow an administrator to search inventory using any of about 200 pieces of information collected about a device.

Here, I'm using criteria to match the model identifiers of all Macs that support macOS Catalina to create a smart group. **Does anyone know how many specific models of Mac support Catalina?**

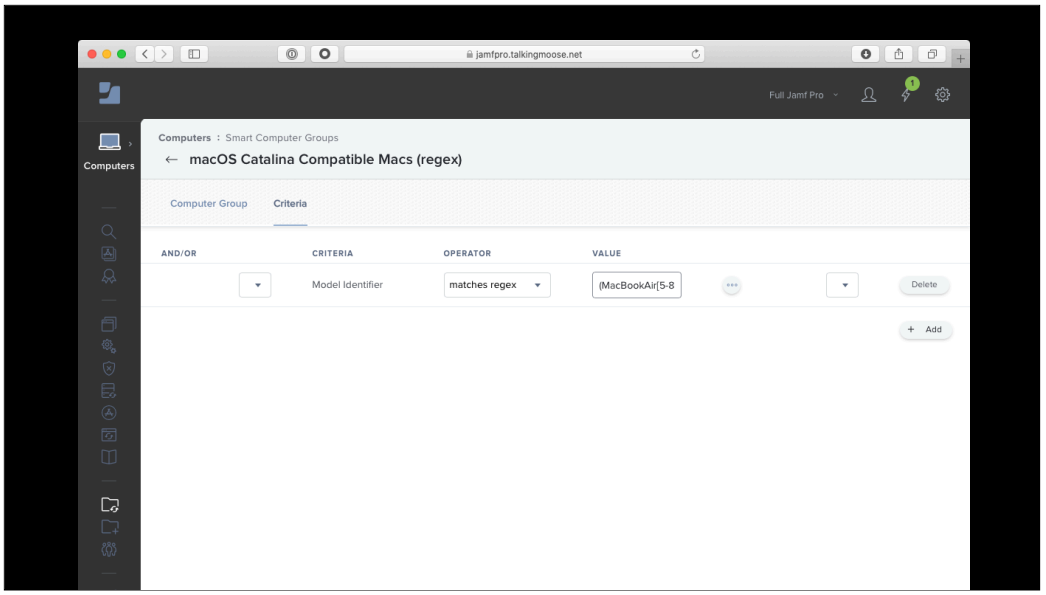
...

As of May 2020, **Catalina will run on 62 models of Macs.** And the only way we have to identify them in inventory is using their model identifiers. **We can't take Apple's term** of "MacBook Pro (13-inch, 2020, Two Thunderbolt 3 ports)" because that name doesn't exist anywhere on the Mac where we can read it from the operating system. Even if it did, we'd still have 61 more to collect.

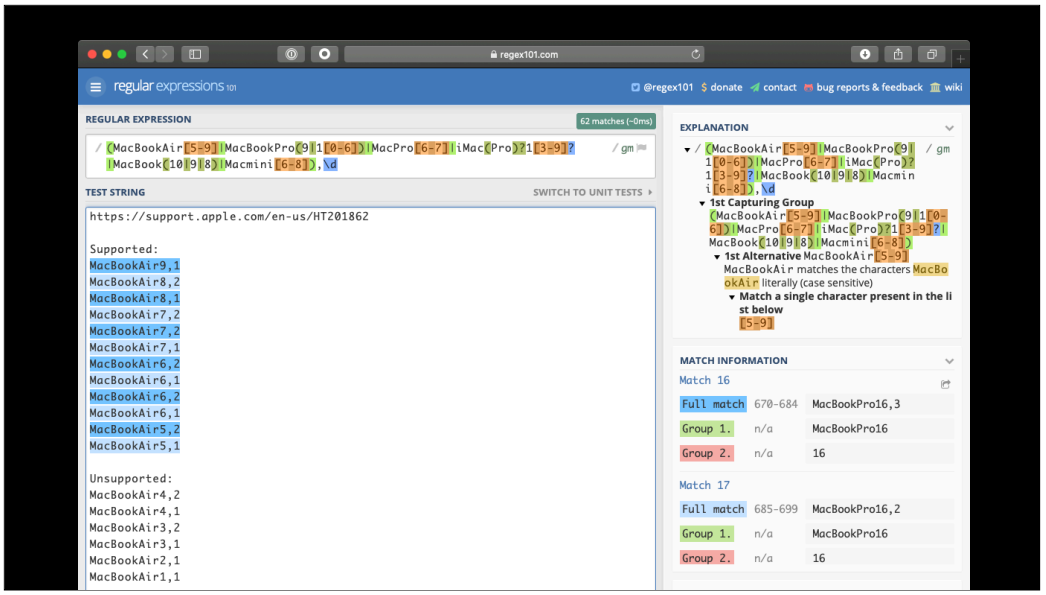
More importantly, model identifiers follow a pattern.



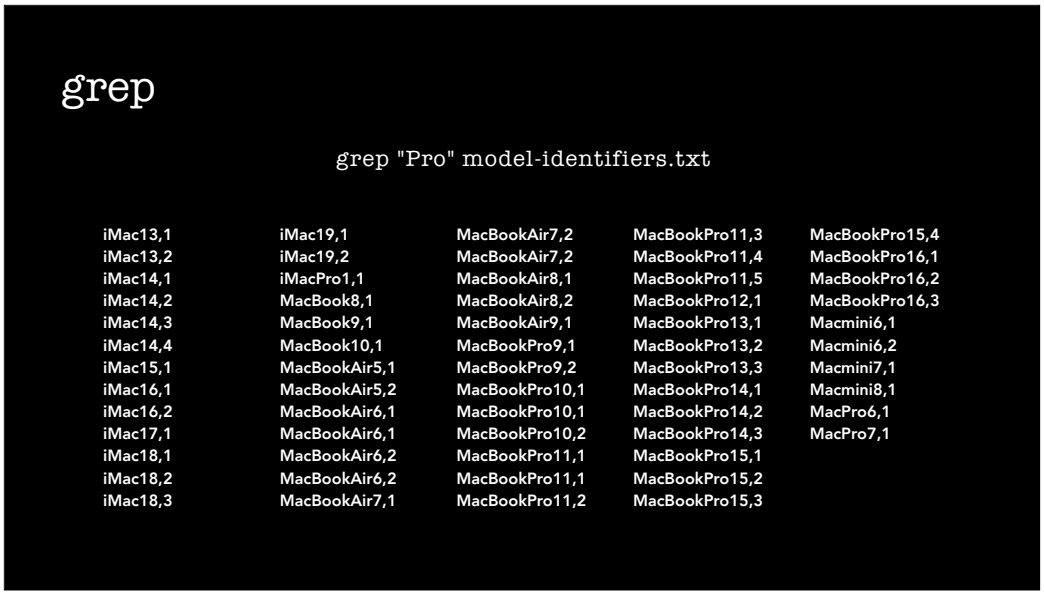
It's longer than any regex we've seen so far and I've made liberal use of ranges and capture groups, **but this one regular expression** will match these 62 Mac model identifiers.



And in Jamf Pro, I get to simplify 62 different criteria to just one regex. And **processing** this one regex, even with eight "or" operators in the string, is going to be a lot more efficient than processing 62 possible matches.



If you'd like to experiment yourself with this and deconstruct the regex, I'll have it along with the list of model identifiers listed in the resources at the end of the presentation.

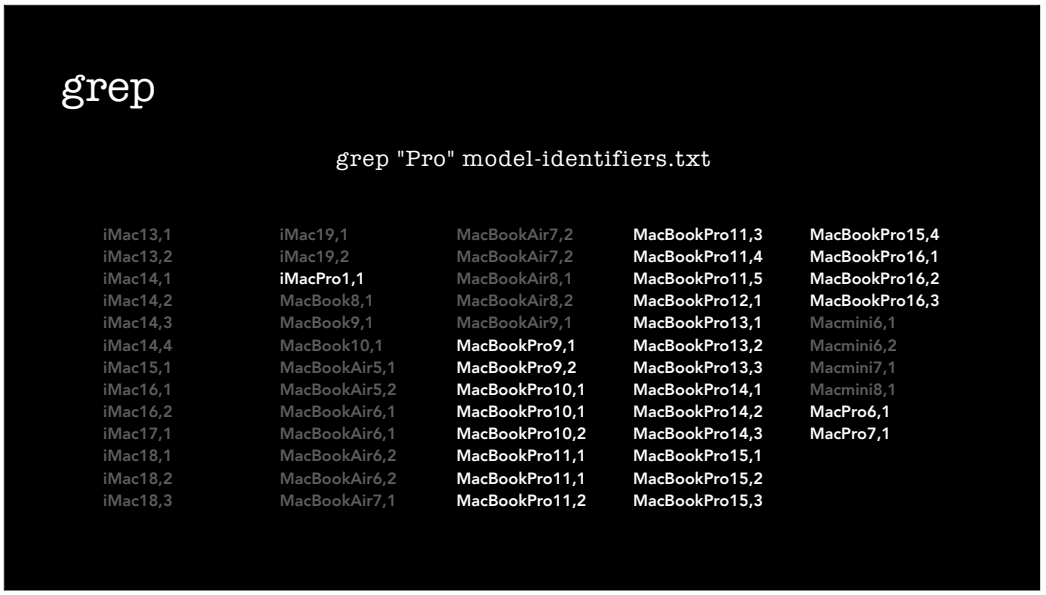


If we have a file that lists all Catalina-compatible Macs by their model identifiers, **grep can read** that file and **identify** every line that contains the pattern "Pro".

I have them listed in five columns here, but pretend they're each written to a single line in the file.

And notice, this time, I'm not using "cat" to read the file and piping that into grep. I don't need to. Grep is perfectly capable of reading files on its own. That's what I meant earlier about the "useless use of cat".

When I run the command on the file, it prints only those lines that match my pattern. And from there, I can take the new list and process it however I need.

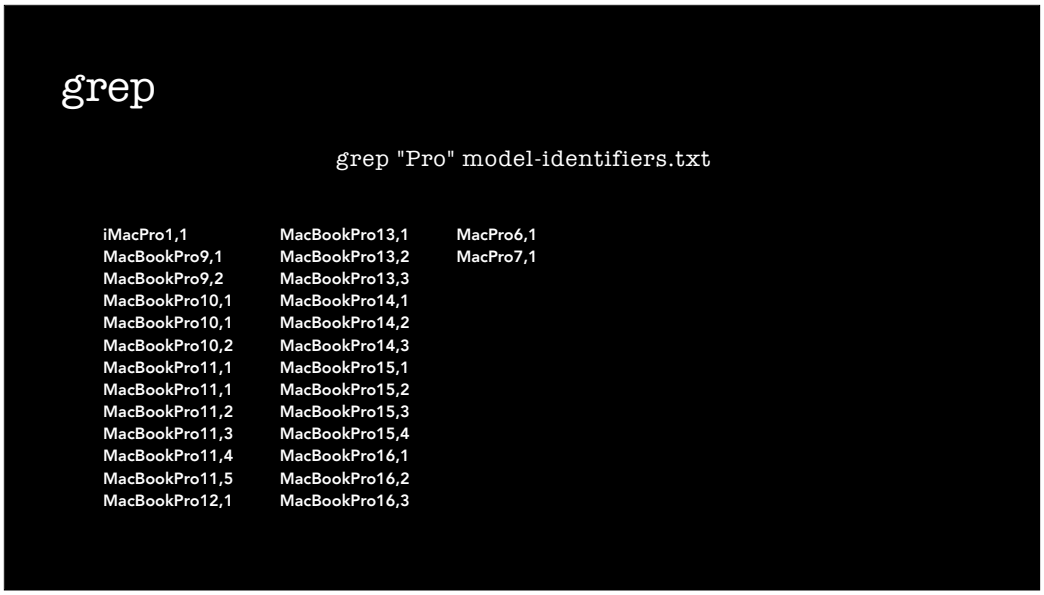


If we have a file that lists all Catalina-compatible Macs by their model identifiers, **grep can read** that file and **identify** every line that contains the pattern "Pro".

I have them listed in five columns here, but pretend they're each written to a single line in the file.

And notice, this time, I'm not using "cat" to read the file and piping that into grep. I don't need to. Grep is perfectly capable of reading files on its own. That's what I meant earlier about the "useless use of cat".

When I run the command on the file, it prints only those lines that match my pattern. And from there, I can take the new list and process it however I need.



If we have a file that lists all Catalina-compatible Macs by their model identifiers, **grep can read** that file and **identify** every line that contains the pattern "Pro".

I have them listed in five columns here, but pretend they're each written to a single line in the file.

And notice, this time, I'm not using "cat" to read the file and piping that into grep. I don't need to. Grep is perfectly capable of reading files on its own. That's what I meant earlier about the "useless use of cat".

When I run the command on the file, it prints only those lines that match my pattern. And from there, I can take the new list and process it however I need.

I can further narrow my list **using brackets** in the command to find just newer Pro models like those 12-16.

But what else just happened?

I lost all my Mac Pros and my iMac Pro because their model identifiers weren't in that range. All I'm left with are just MacBook Pros.

That should be easy to fix, right? I can adjust my regex to add a capture group in parentheses to find numbers in the range of 12-16 **or** just a single digit.

I can further narrow my list **using brackets** in the command to find just newer Pro models like those 12-16.

But what else just happened?

I lost all my Mac Pros and my iMac Pro because their model identifiers weren't in that range. All I'm left with are just MacBook Pros.

That should be easy to fix, right? I can adjust my regex to add a capture group in parentheses to find numbers in the range of 12-16 **or** just a single digit.

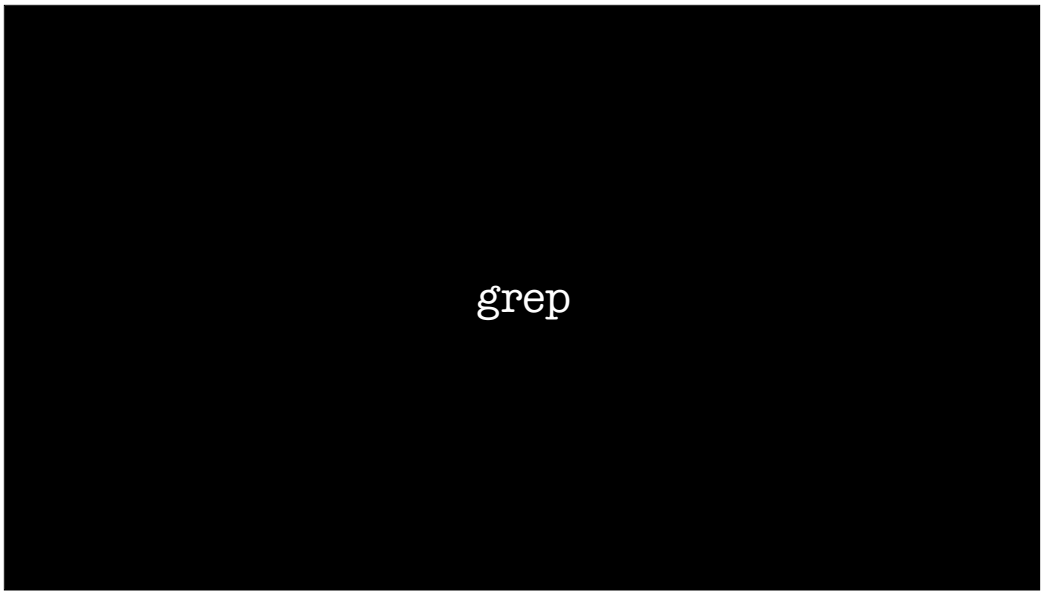
grep

```
grep "Pro(l[2-6]|\d)," model-identifiers.txt
```

And it'll look like this.

But for some reason when I run the command, it doesn't work. I get nothing matched.

Why is that?



If we need to get more complex with our search, **we need to add the "-E" option**, which is the shorter form of "--extended-regexp".

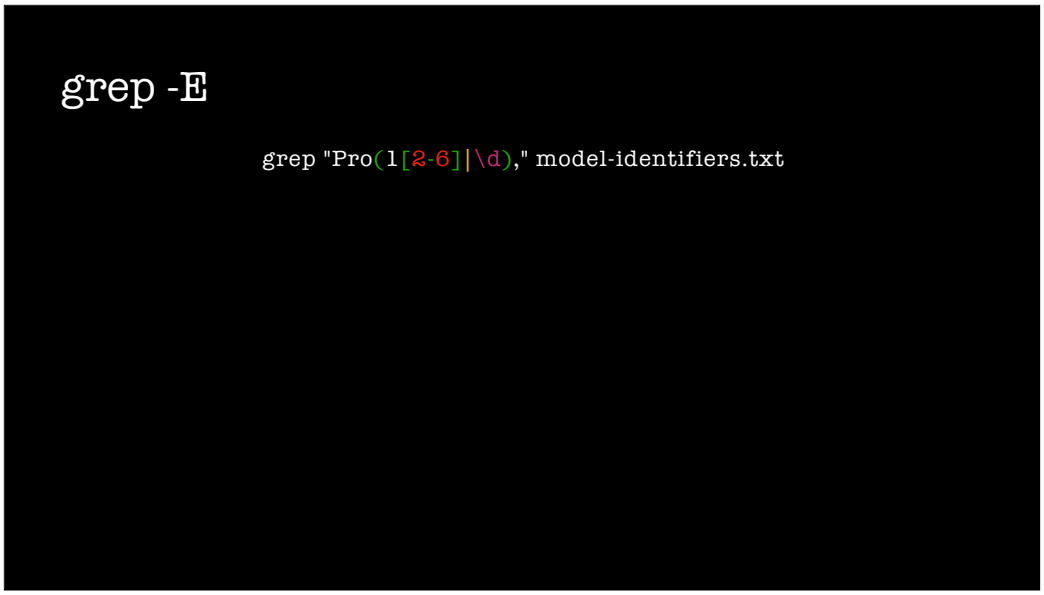
Both of these commands are the same. One is just a shorter way of writing it. I encourage you to use the longer form in your scripts because it better explains to you later or to someone else reading your script what you're doing.

So, what's an **extended** regular expression? Simply put, **any time that we need to use certain regex characters** like the plus symbol, the question mark, the pipe symbol or parentheses, we're no longer performing a basic pattern match where our regex characters match just one character like the number "1". We're now performing a more complex match where our regex characters may match more than one character — 1 **or** 11 **or** 111 **or** 1111, etc.

If we need to get more complex with our search, **we need to add the "-E" option**, which is the shorter form of "--extended-regexp".

Both of these commands are the same. One is just a shorter way of writing it. I encourage you to use the longer form in your scripts because it better explains to you later or to someone else reading your script what you're doing.

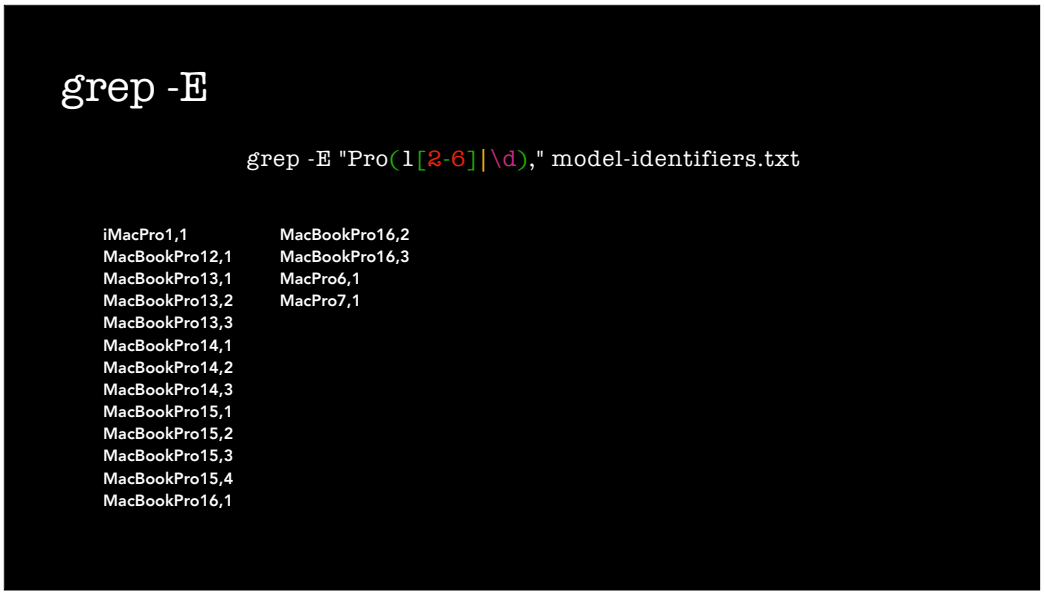
So, what's an **extended** regular expression? Simply put, **any time that we need to use certain regex characters** like the plus symbol, the question mark, the pipe symbol or parentheses, we're no longer performing a basic pattern match where our regex characters match just one character like the number "1". We're now performing a more complex match where our regex characters may match more than one character — 1 **or** 11 **or** 111 **or** 1111, etc.



Here's our grep command that didn't return anything.

We'll add the -E.

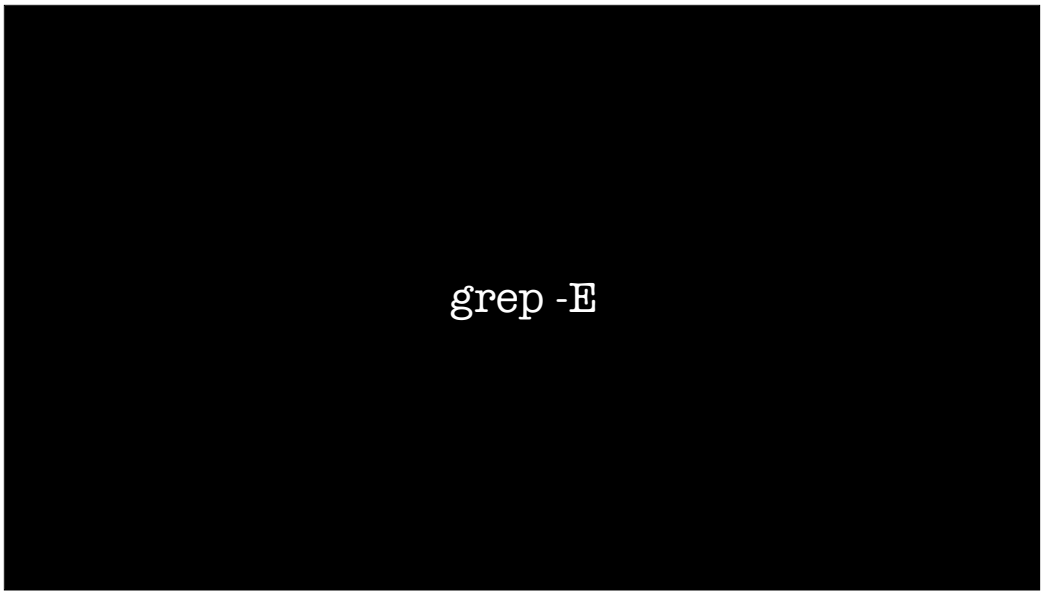
And now when we run it, we get all the Pro Macs including the iMac Pro, MacBooks Pro and Macs Pro.



Here's our grep command that didn't return anything.

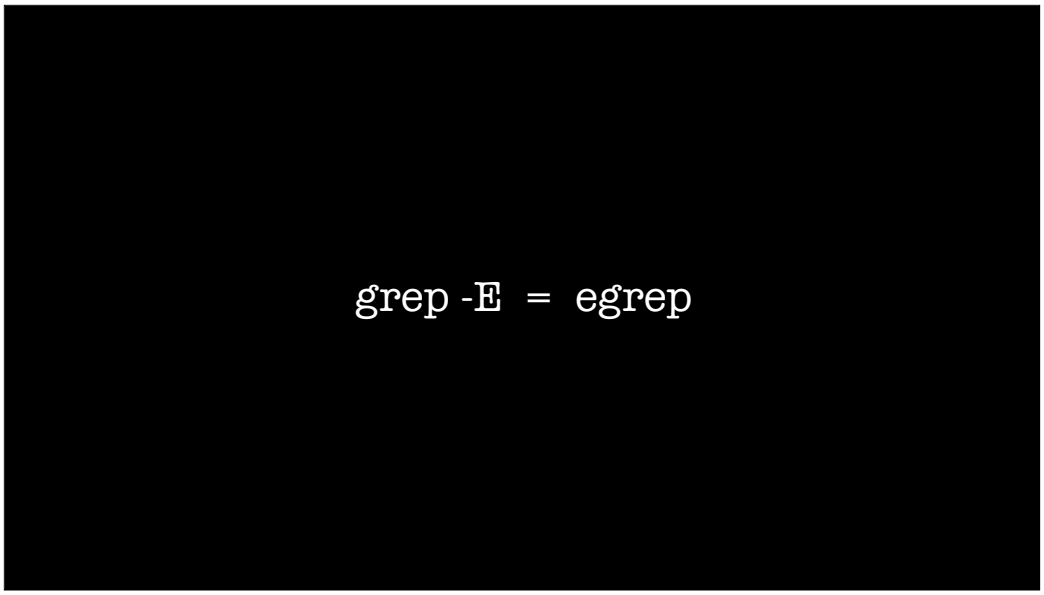
We'll add the -E.

And now when we run it, we get all the Pro Macs including the iMac Pro, MacBooks Pro and Macs Pro.



One last note for you about grep.

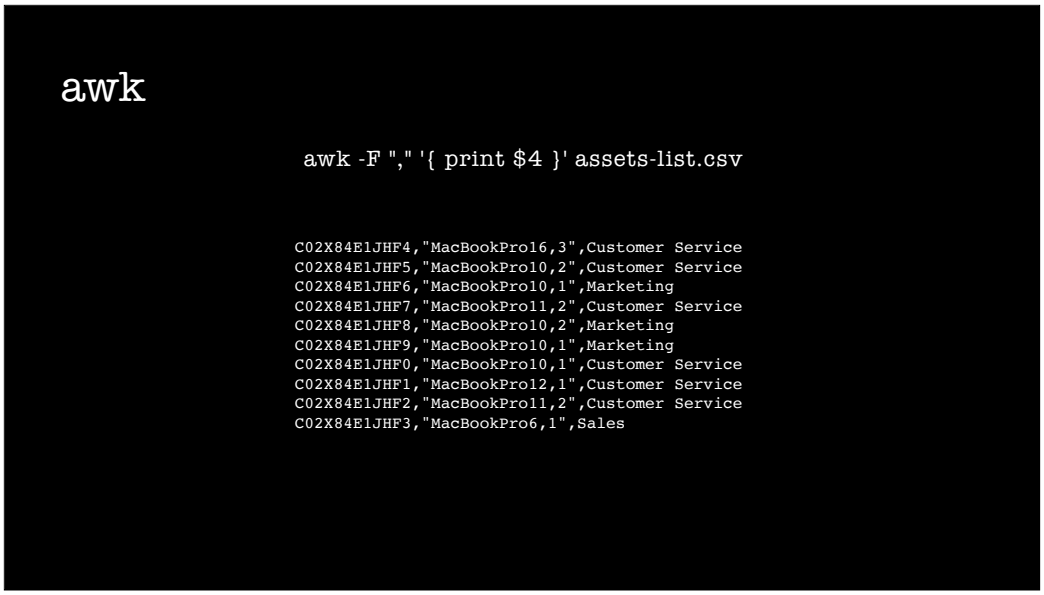
"grep -E" **is the same as** the egrep command. The "e" in egrep stands for "extended".



One last note for you about grep.

"grep -E" **is the same as** the egrep command. The "e" in egrep stands for "extended".

Another common tool we'll use in our scripting is awk and it too supports regex in several different ways. I'll show you two.



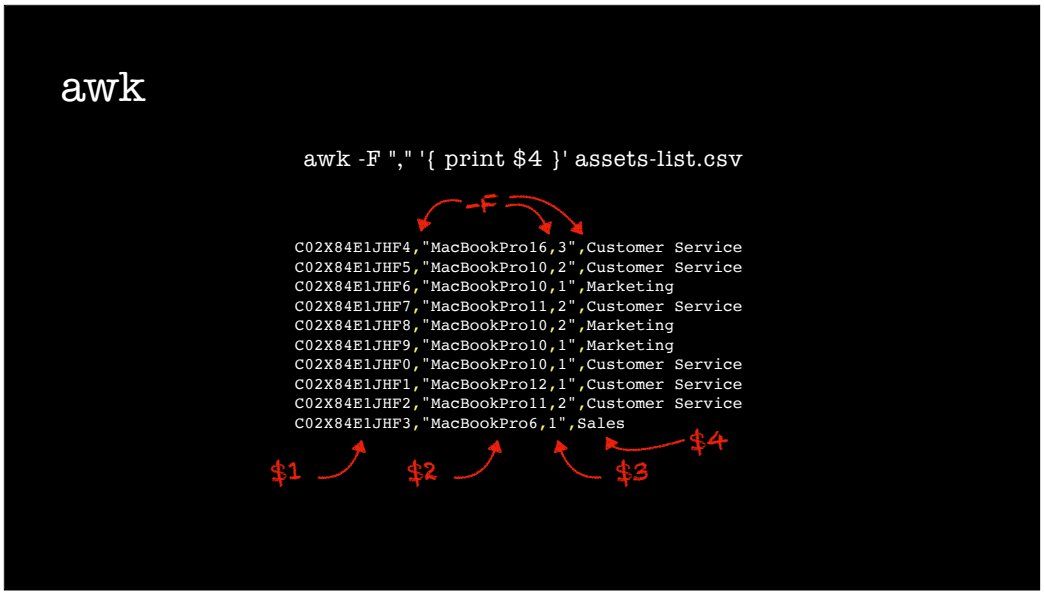
One of the most common uses for awk is to evaluate text files with multiple columns of text.

Here I have the contents of a comma-separated-values list — a CSV file — with Mac serial numbers in the first column, model identifiers in the second column and departments in the third column. I'm only showing 10 rows, but imagine this could be a list of hundreds or thousands of computer records.

My goal using awk is to identify all the older MacBook Pro models my company no longer supports and then contact the managers of each department to schedule refreshes. Specifically, the MacBook Pro models I want to refresh are those with an identifier of 10-something or less.

I'll start with a simple awk statement that gets the last column of data. **The "-F"** lets me specify a character that separates each column — **in this case it's a comma**. And the "print $4" means print the fourth column. Notice the model identifiers themselves add an extra comma. That's why I can't use "$3" to get the department names — I have to shift over to "$4".

When I run my command on the CSV file, it returns just a list of my departments.



One of the most common uses for awk is to evaluate text files with multiple columns of text.

Here I have the contents of a comma-separated-values list — a CSV file — with Mac serial numbers in the first column, model identifiers in the second column and departments in the third column. I'm only showing 10 rows, but imagine this could be a list of hundreds or thousands of computer records.

My goal using awk is to identify all the older MacBook Pro models my company no longer supports and then contact the managers of each department to schedule refreshes. Specifically, the MacBook Pro models I want to refresh are those with an identifier of 10-something or less.

I'll start with a simple awk statement that gets the last column of data. **The "-F"** lets me specify a character that separates each column — **in this case it's a comma**. And the "print $4" means print the fourth column. Notice the model identifiers themselves add an extra comma. That's why I can't use "$3" to get the department names — I have to shift over to "$4".

When I run my command on the CSV file, it returns just a list of my departments.

One of the most common uses for awk is to evaluate text files with multiple columns of text.

Here I have the contents of a comma-separated-values list — a CSV file — with Mac serial numbers in the first column, model identifiers in the second column and departments in the third column. I'm only showing 10 rows, but imagine this could be a list of hundreds or thousands of computer records.

My goal using awk is to identify all the older MacBook Pro models my company no longer supports and then contact the managers of each department to schedule refreshes. Specifically, the MacBook Pro models I want to refresh are those with an identifier of 10-something or less.

I'll start with a simple awk statement that gets the last column of data. **The "-F"** lets me specify a character that separates each column — **in this case it's a comma**. And the "print $4" means print the fourth column. Notice the model identifiers themselves add an extra comma. That's why I can't use "$3" to get the department names — I have to shift over to "$4".

When I run my command on the CSV file, it returns just a list of my departments.

Now, I need to add some more awk syntax to handle the regex.

I'm doing multiple things here.

First, I'm using "$2" to tell awk to search in the second column.

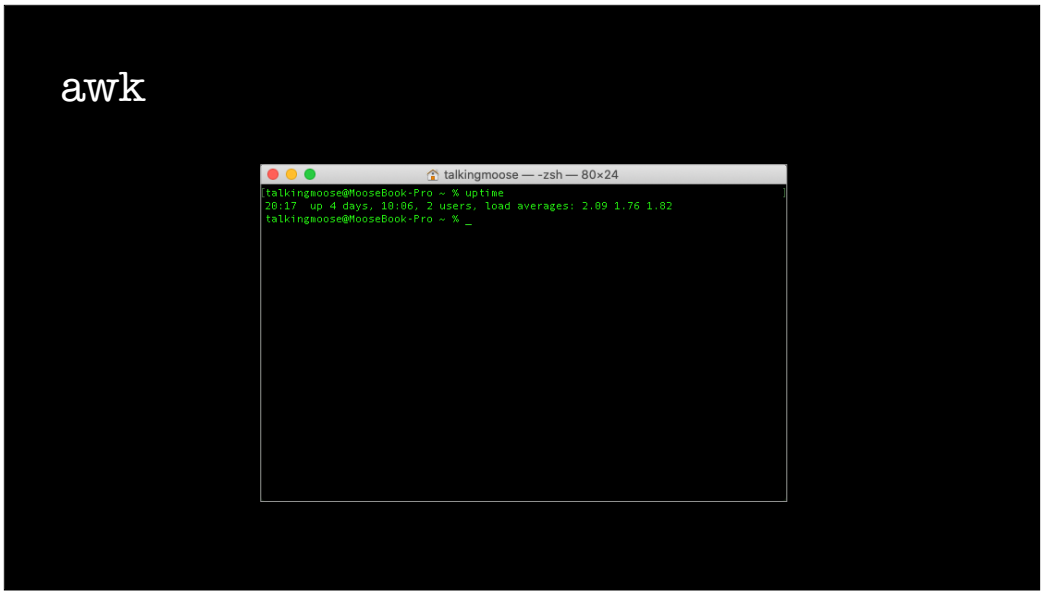
The regex is between the two slashes. And it's looking for the string "MacBookPro1" followed by a range of "1" to "9". So, it matches all MacBook Pro model identifiers 11-19. The highest number to date is on the first line, which is 16, so this regex will be a little future proof probably for the next couple of years at least.

Finally, I don't want to match identifiers "11" through "19". I actually want everything "10" and below. When using regex in awk, the tilde means "matches". **By putting a "bang"** or "exclamation point" in front of it, that mean "doesn't match".

Altogether, my awk command is saying:

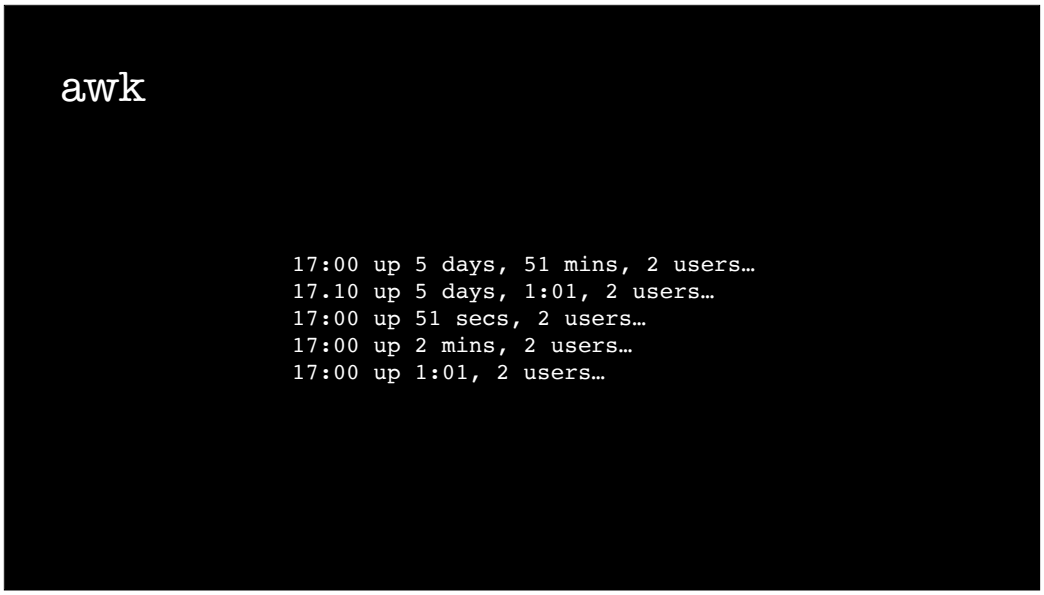
- Read the file at the end of the command — assets-list.csv
- Divide the text of the file into columns at every comma
- Read the second column of each line and look for anything that doesn't match MacBookPro "11-19"
- And then print out the fourth column

What I'm left with are these departments. If I wanted, I could add a sort command to the end to alphabetize and remove duplicates so that all I get is just "Customer Service", "Marketing" and "Sales".



Here's another way to use regex in awk.

There's a command you can run in Terminal called "uptime" and it does pretty much what it says — it tells you how long your computer has been up. And if you look at the result here, it's pretty easy to see my Mac has been up 4 days, 10 hours. That could be handy for reporting purposes and troubleshooting.



But there's a catch to the response you'll get. When working with one of my customers, we created a command to get the uptime and then trimmed out the parts before and after it that weren't really important and just give us the uptime. But we kept getting really inconsistent results.

Eventually, we found out why. The result of uptime could be could be one of five different strings like those you see here.

Looking from top to bottom in this list:

- The computer could be up a certain number of days and minutes
- Or it could be up a certain number days and less than two minutes
- Or it could be up a certain number of seconds
- Or less than 60 minutes
- Or up less than two minutes

So, then we had to think about how we divide up this information. Commas are usually a good bet, but in this case the first two

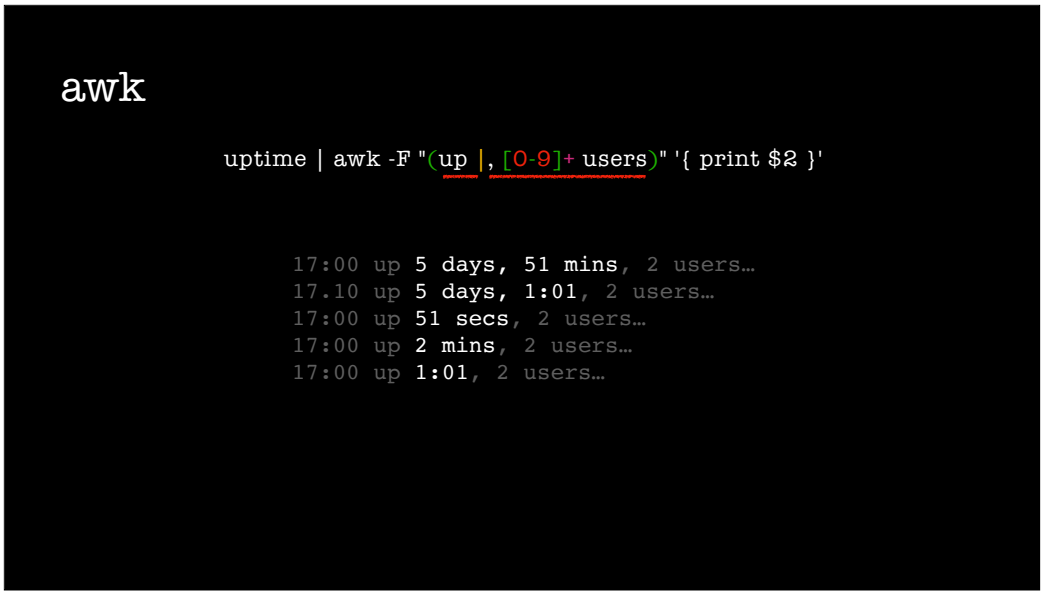
results actually include commas as part of the uptime. The rest don't, though.

We also considered spaces, but again the first two results contain spaces too.

Then we thought about using strings of characters before and after the uptime and that partially worked. But we couldn't always guarantee two users would be logged in. There could be more.

Regex helped us with this.

Remember, the "-F" specifies a field separator that lets us divide rows of text into columns. **In our solution**, the word "up" with a space after it defined our first field separator and then we used the "or" operator to specify a second separator that began with a comma followed by a number range followed by the word users. This let us capture uptimes whether they contained commas, spaces or not.



But there's a catch to the response you'll get. When working with one of my customers, we created a command to get the uptime and then trimmed out the parts before and after it that weren't really important and just give us the uptime. But we kept getting really inconsistent results.

Eventually, we found out why. The result of uptime could be could be one of five different strings like those you see here.

Looking from top to bottom in this list:

- The computer could be up a certain number of days and minutes
- Or it could be up a certain number days and less than two minutes
- Or it could be up a certain number of seconds
- Or less than 60 minutes
- Or up less than two minutes

So, then we had to think about how we divide up this information. Commas are usually a good bet, but in this case the first two

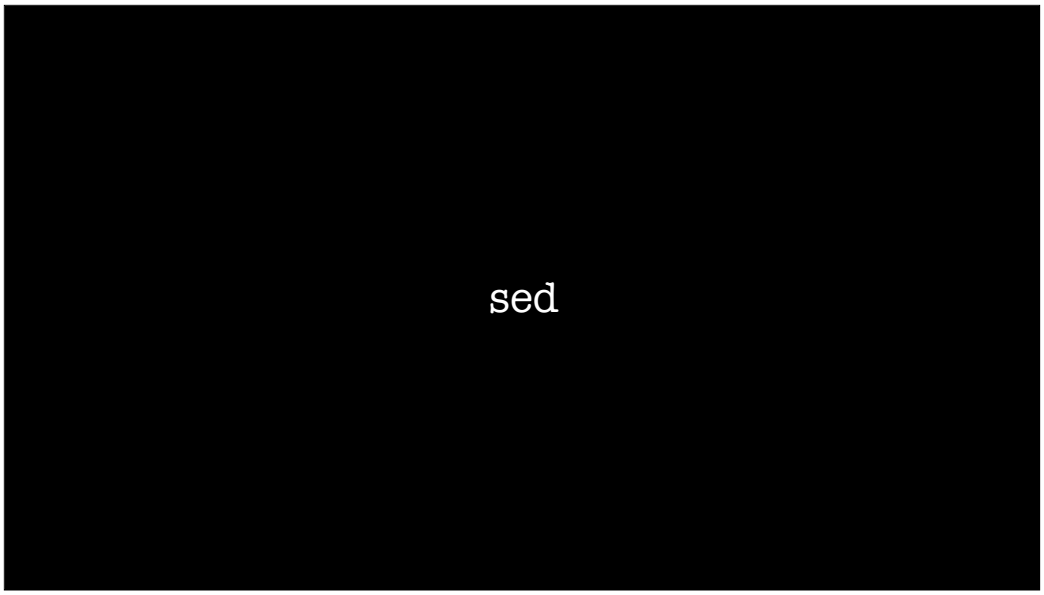
results actually include commas as part of the uptime. The rest don't, though.

We also considered spaces, but again the first two results contain spaces too.

Then we thought about using strings of characters before and after the uptime and that partially worked. But we couldn't always guarantee two users would be logged in. There could be more.

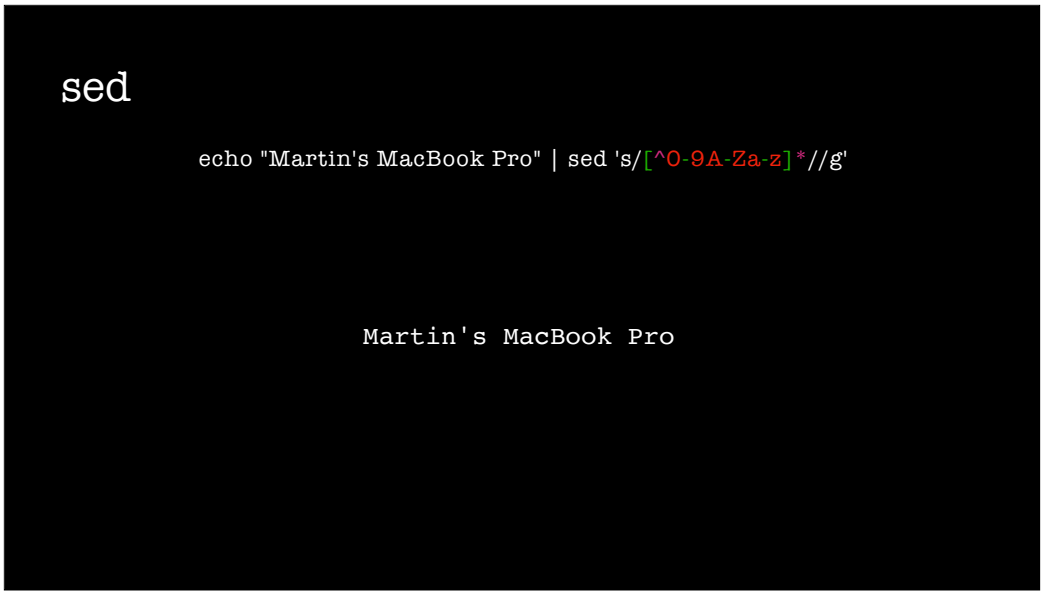
Regex helped us with this.

Remember, the "-F" specifies a field separator that lets us divide rows of text into columns. **In our solution**, the word "up" with a space after it defined our first field separator and then we used the "or" operator to specify a second separator that began with a comma followed by a number range followed by the word "users". This let us capture uptimes whether they contained commas, spaces or not.



One more simple example.

If you're familiar with grep and awk then the sed command line tool isn't far behind.



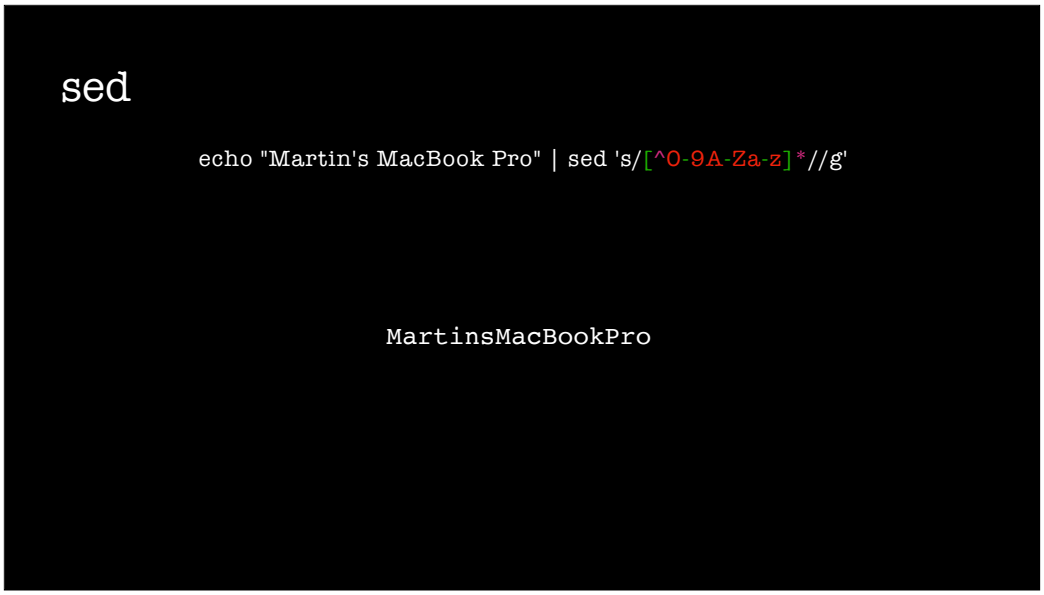
The vast majority of the time that we're using sed, we're trying to remove something or change something.

A lot of us like to follow a computer naming convention of one form or another. One of the reasons is for compatibility with an external system. Sometimes, those systems don't like special characters. But the macOS Setup Assistant frequently names our computers something like this.

We can use regex in a sed command to remove all non-alphanumeric characters.

Here, the sed statement is substituting any character that's not a number, upper case or lower case character with nothing.

The result is pretty straight-forward. All symbols and spaces are removed.



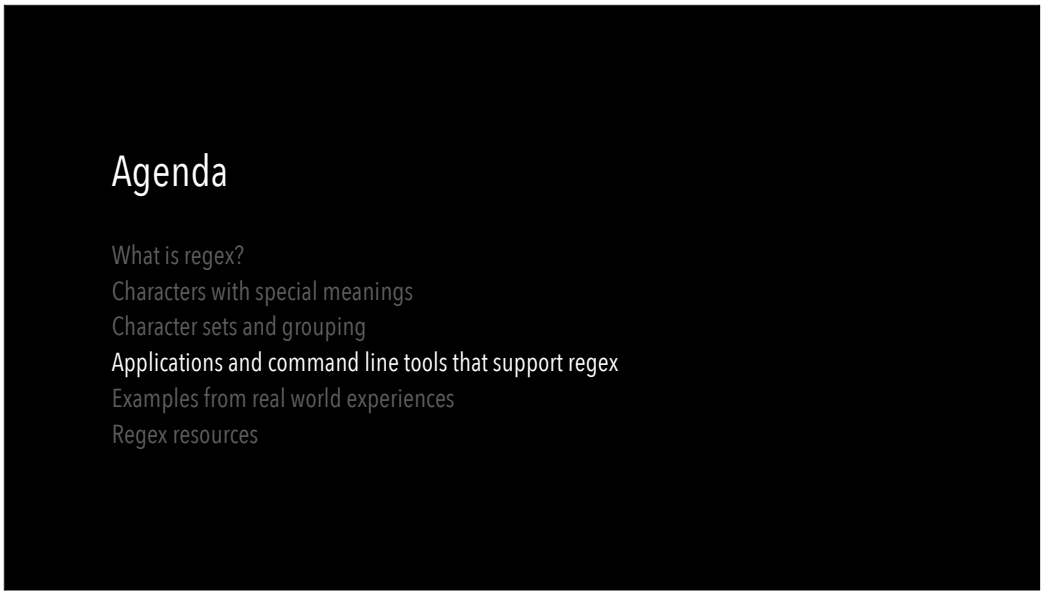
The vast majority of the time that we're using sed, we're trying to remove something or change something.

A lot of us like to follow a computer naming convention of one form or another. One of the reasons is for compatibility with an external system. Sometimes, those systems don't like special characters. But the macOS Setup Assistant frequently names our computers something like this.

We can use regex in a sed command to remove all non-alphanumeric characters.

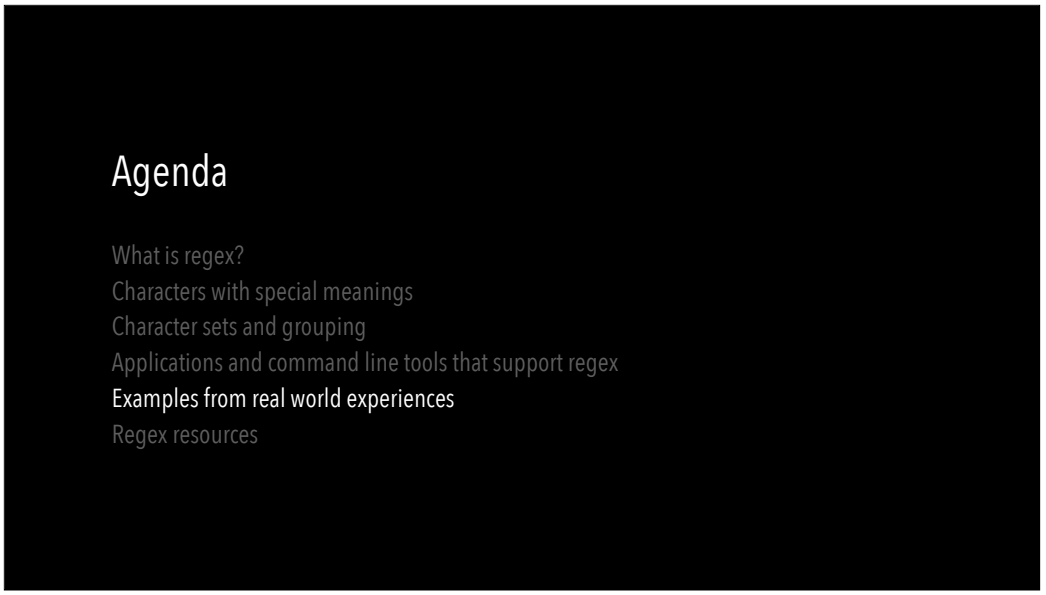
Here, the sed statement is substituting any character that's not a number, upper case or lower case character with nothing.

The result is pretty straight-forward. All symbols and spaces are removed.

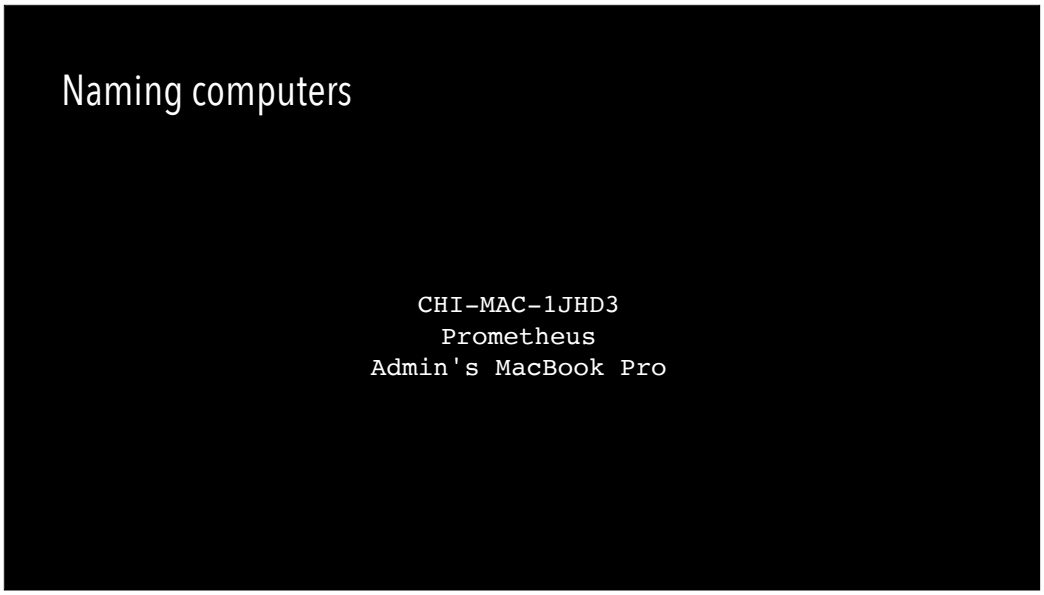


I hope by now you realize that regex has probably been a part of your scripting and command line work whether you knew it or not.

If you weren't aware of it, I hope you take away today that you've got a powerful search and editing tool baked in to a lot of your daily tools.



What I'd like to do now is show you a couple of the more interesting uses I've helped put together for my customers. Maybe these will give **you** some ideas for where you can start using regex.

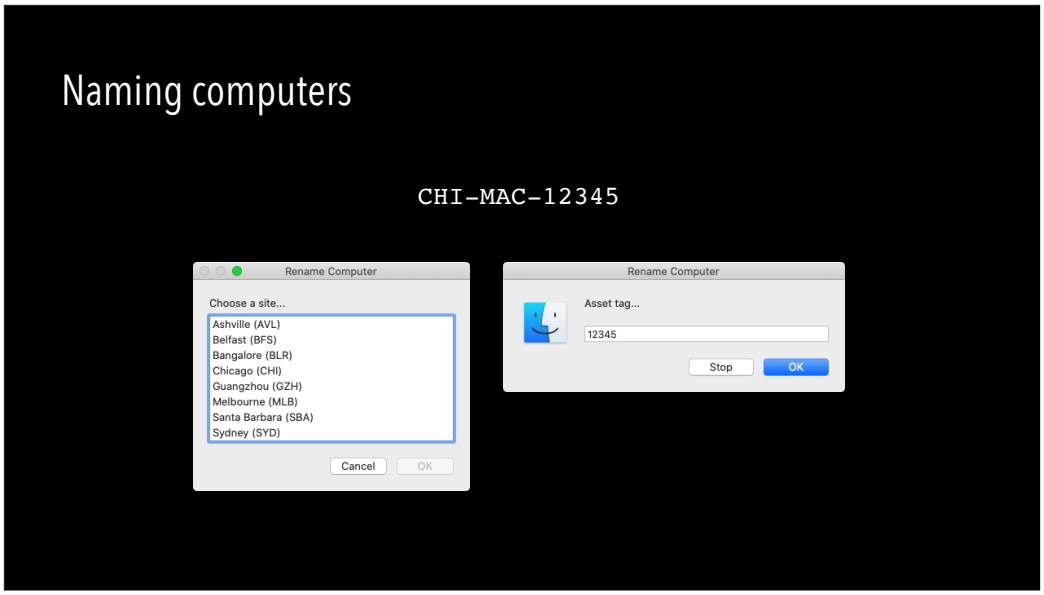


I had a customer with 472 Macs enrolled into Jamf Pro. This was a system he largely inherited from the former administrator who left the company.

He wanted to change the computer naming scheme to something slightly different from his predecessor. On top of that, he had some developers with admin rights who liked to change the computers names to something else and sometimes the technicians preparing the Macs forgot to rename them.

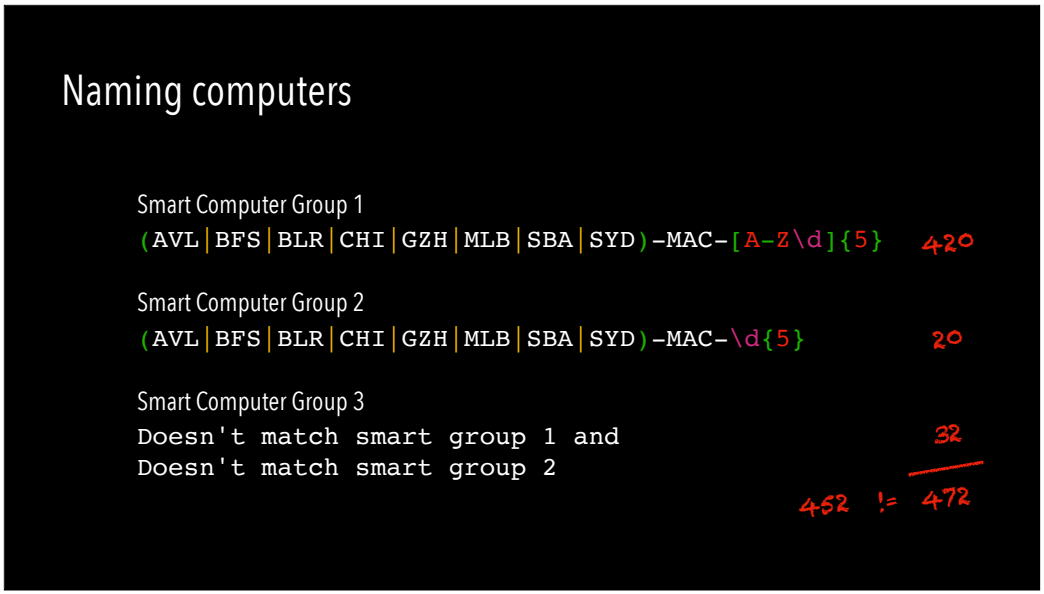
The old naming scheme followed a simple convention of three-character site code — dash — MAC — dash — and the last five characters of the computer serial number. If you know anything about the structure of Mac serial numbers, you know the last four characters identify the model of the Mac. That means the last four characters of a lot of his computer names were going to be the same. In fact, my customer had several computers with the exact same name.

These are some of the names you could expect to find on his network.



What he wanted to do was name everything similar to the established naming convention, but instead of using part of the serial number, he'd run a script at enrollment that prompted technicians to choose a site and enter the asset tag from the bottom of the computer.

The script was easy to put together, but what we had to do next was identify Macs that **didn't** follow the new naming convention.



So, to identify Macs that were named correctly as well as incorrectly, we created three smart groups and put them on our dashboard to track progress.

The first smart group used regex to identify Macs with the older naming convention where the site code was one of eight possible three-letter sites — dash — MAC — dash — and the last five characters of the computer serial number. Knowing Apple's format for serial numbers, we knew the last five characters would contain a combination of letters **and** numbers (or digits).

The second smart group's regex identified Macs with the **new** naming convention we wanted. Instead of five alphanumeric characters at the end, we wanted the asset tag of only five numbers (or digits).

And then we added a third smart group to identify Macs with names that didn't match either of these patterns.

When we went to look at the numbers, we found something odd. **Our first group matched 420 Macs.** We knew this would be the biggest group, so it made sense.

Our second group matched 20 Macs. That made sense too because the technicians had been following the new naming convention for a few weeks.

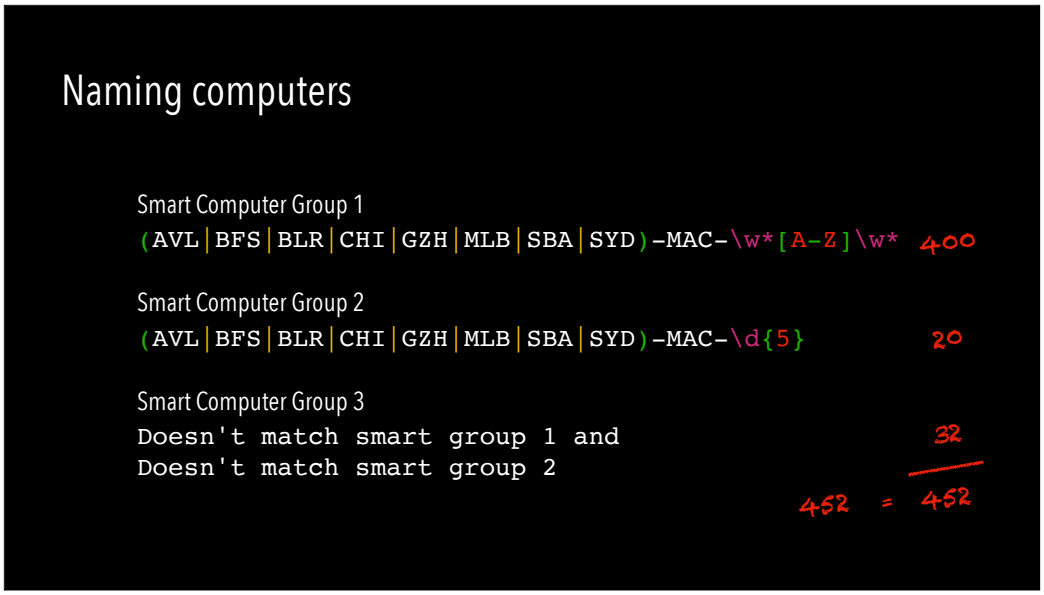
And our third group matched 32 Macs where the names had either been changed by developers or the technicians had simply forgotten to rename then.

All of these numbers made sense until we looked at our **total Mac count**, which was 452. We had a discrepancy of 20 Macs.

Can anyone see what we did wrong? I'll give you a few seconds to look things over and if you see the problem, feel free to shout it out.

...

If you compare the regexes at the ends of both patterns, you'll see the first matches any **combination** of letters and numbers and the second only matches numbers. The same Macs that matched only numbers at the end were being matched by **both** regexes not just the second one.

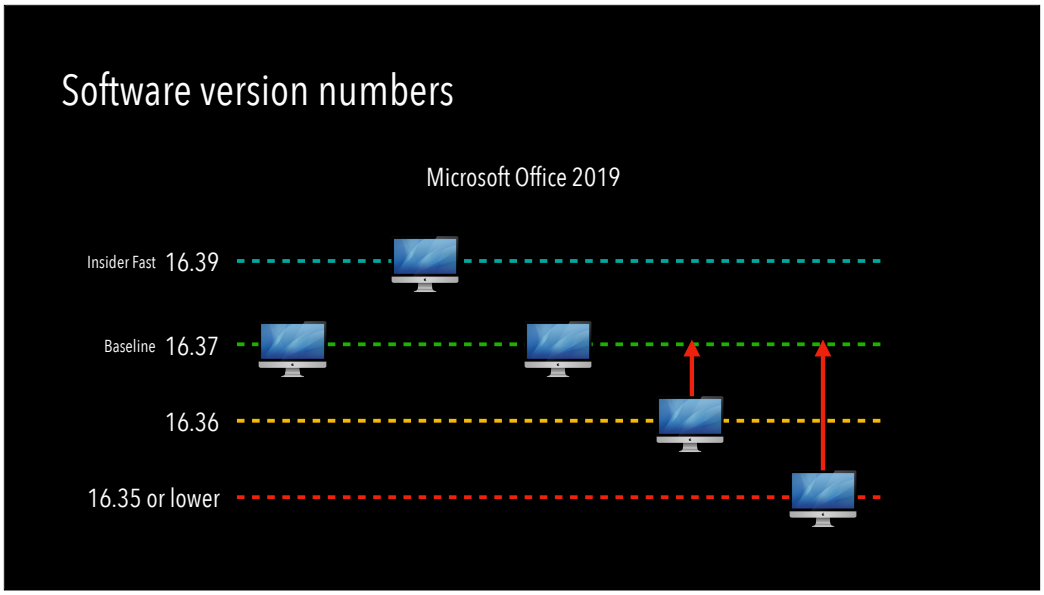


What we had to do was modify the first regex to match only if there were both letters and numbers.

The change we made was a little tricky. If you recall from our cheatsheet, the "\w" equals any letter or number and the asterisk makes that "0" or more letters or numbers. So with the "**\w***" at both the beginning and end of the regex, we guarantee the first and last characters will be either letters or numbers, and then with the "A-Z" range in the middle, we **guarantee** at least one letter.

When we viewed our totals again, everything came out correctly.

This was a good example to illustrate that if you're creating regexes to identify two completely different groups, be sure there's no potential overlap.



Finally, here's one last project in regex that took me a while to complete.

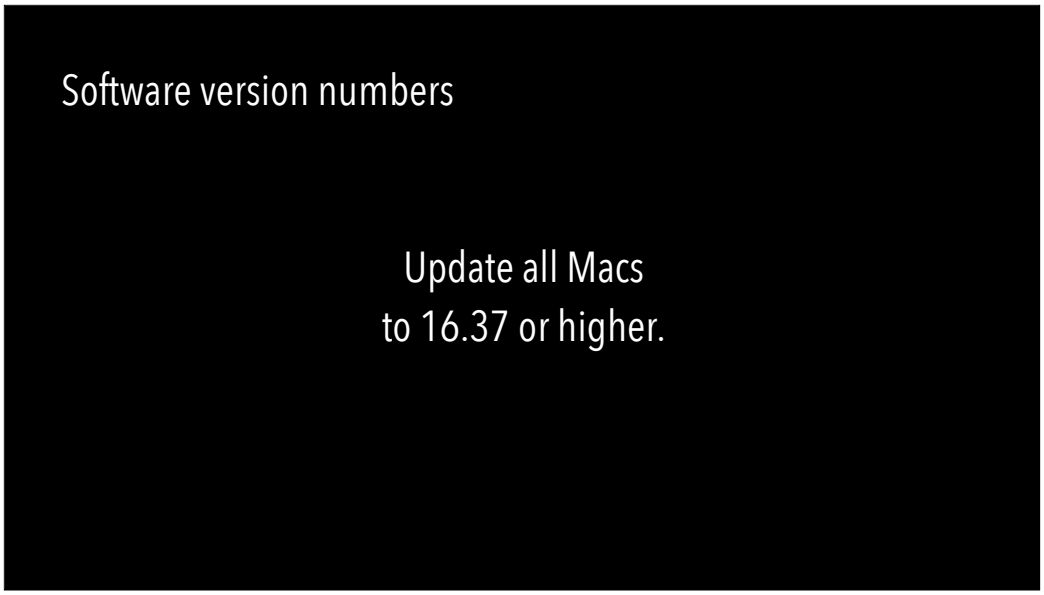
I've had more than one customer with a very specific need. They want to control the versions of software installed on their Macs, but they only want to enforce a minimum version. Here's what I mean by that.

Let's assume an organization deploys Microsoft Office and the **version IT deploys** to production is "16.37". This is their baseline and **some Macs are at baseline**. That's good.

They know **some of their users are on Insider Fast** and may be on a later version like "16.39" and they want to leave these folks alone. They don't want to downgrade them.

However, they have some users who are **on older versions** and falling below baseline. **They need to upgrade these** without touching those that are already at or above baseline.

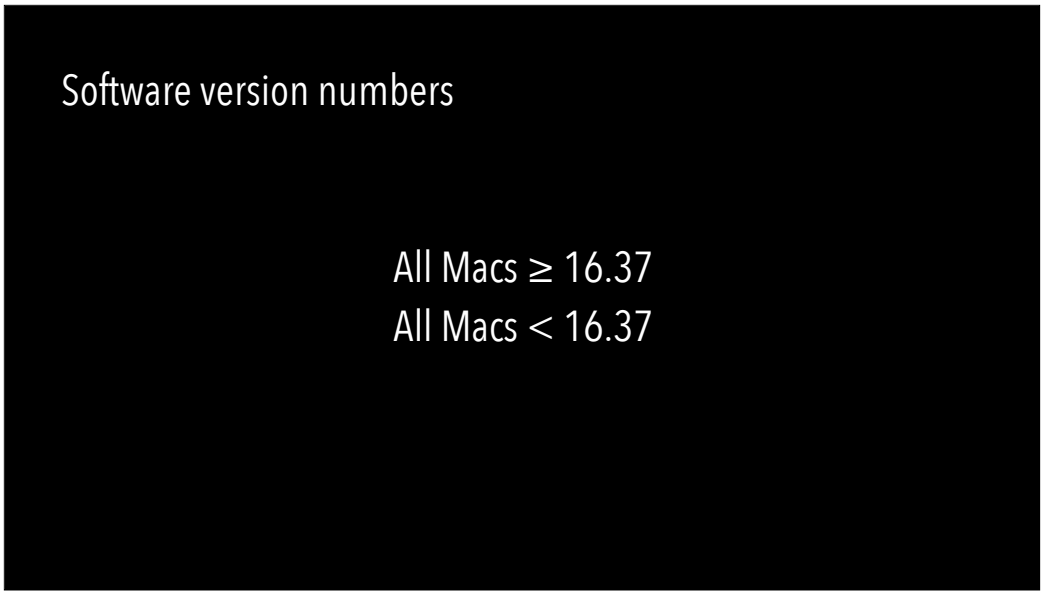
Sounds simple, right?



It's as easy as saying, "Update all Macs to 16.37 or higher."

Such. Simple. Words. I thought it would be easy too.

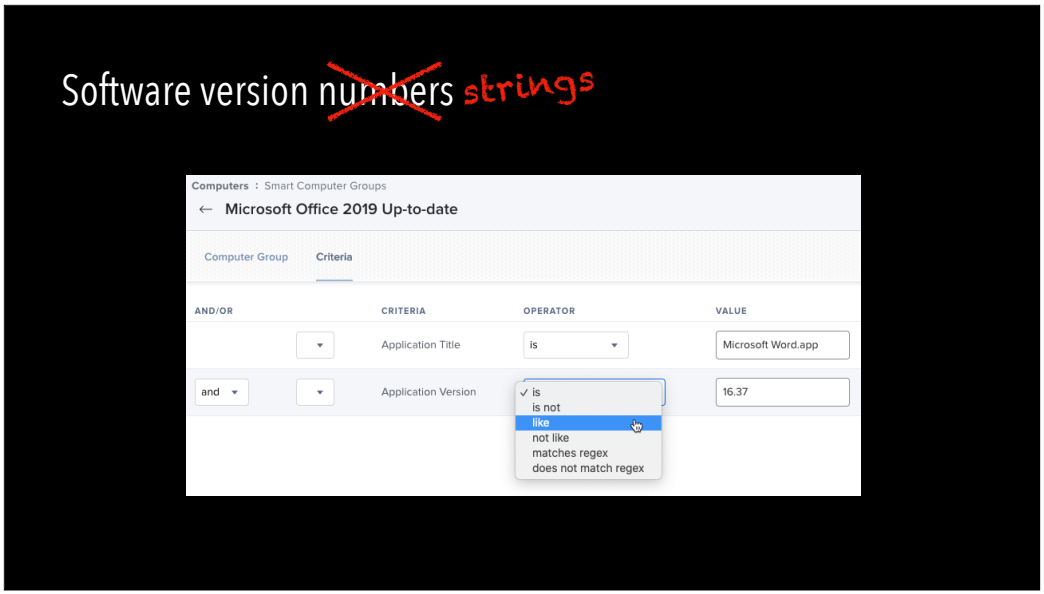
We could look at this two ways:



I could either identify Macs where the Microsoft Office version was greater than or equal to "16.37". Or I could identify Macs where Microsoft Office was less than "16.37".

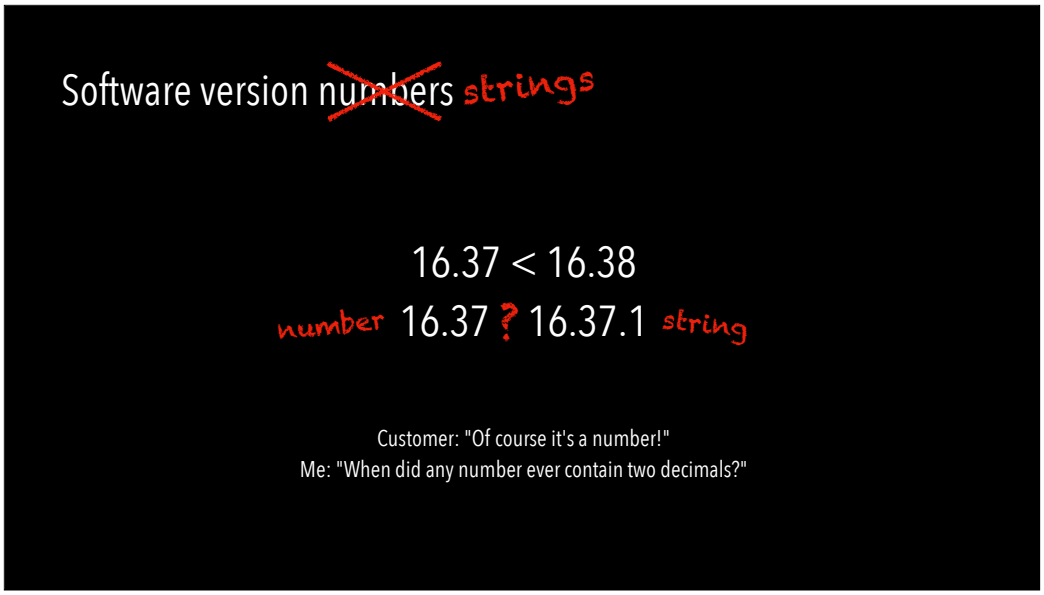
I like to identify the state of Macs as they exist not as they don't exist, so I chose to find all Macs greater than or equal to "16.37".

Now, I'm working in Jamf Pro and it's easy to see the version number for any application. We collect that information natively.



However, there's a gotcha. Jamf Pro's ability to match a version number is there using the **"is"** operator. And it can do something like a wildcard with the **"like"** operator. But there's no ability to say "greater than". And there's a reason why.

Software version numbers aren't really numbers, they're strings and strings of characters don't have a greater than / less than equivalent. **At least not in the computer world.**

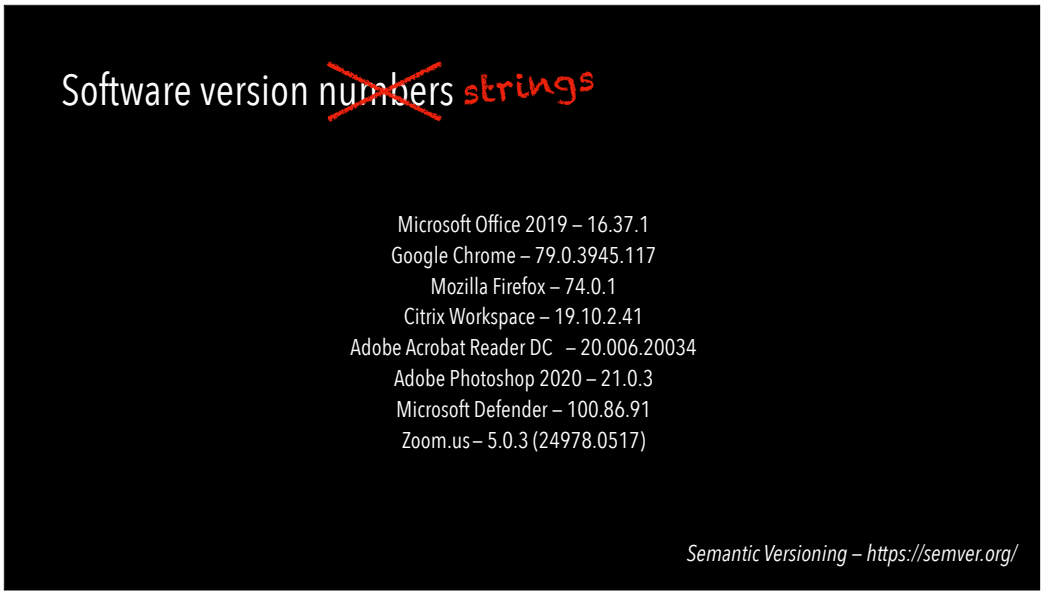


You and I and any computer can can tell that "16.37" is less than "16.38" because they're both legitimate numbers.

But a computer can't do a greater than / less than comparison **between an actual number** and something that **looks like a number** but is really a string.

When I tell that to my customers, they look at me weird and say: **"Of course it's a number!"**

And I reply: **"When did any number ever contain two decimals?"**



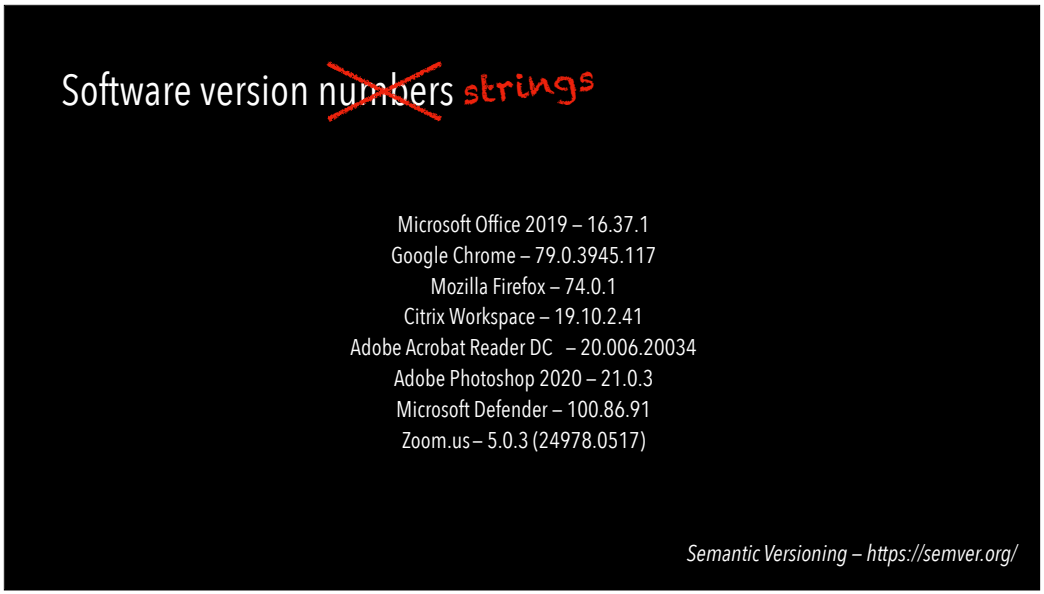
And then I start showing them other applications they're using where the version strings get really wild.

For the most part, Microsoft Office's version string contains one period, but there are hotfix versions where they'll tack on a ".1" and turn it into a string.

And look at Google Chrome and Adobe Acrobat! They each have three decimals!

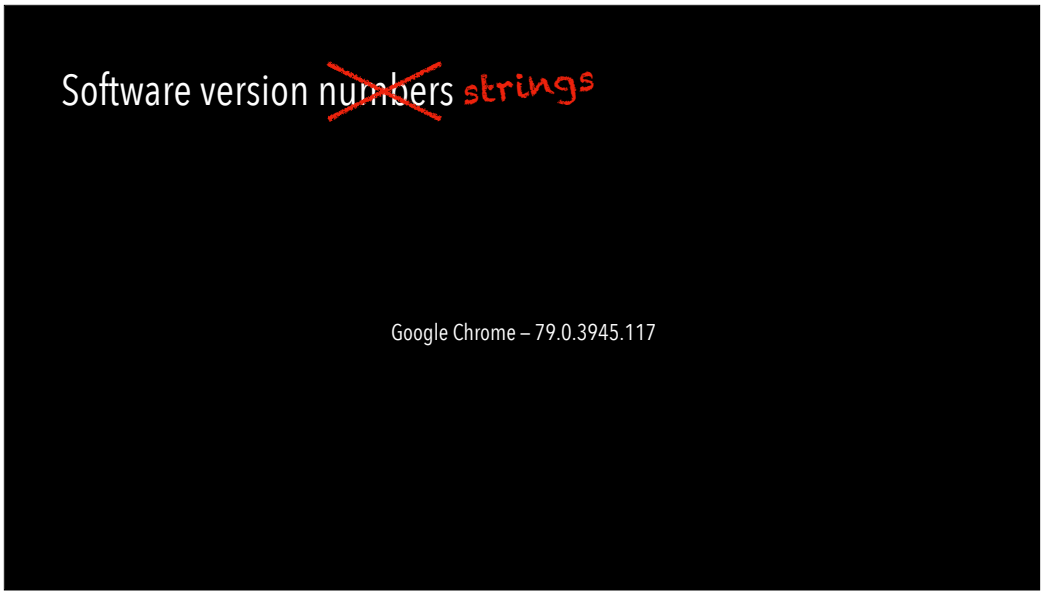
And Zoom at the bottom... **Oh. Zoom.**

Needless to say, most developers don't follow a standard with regard to their version numbers. Office, Firefox, Photoshop and Defender in this list appear to be following **something called Semantic Versioning**, which is three sequences of numbers separated by periods to denote "major", "minor" and "patch" numbers. Acrobat at first glance looks like it follows Semantic Versioning too until you see it includes leading zeroes in its middle sequence in the version string. **Is "006" the same thing as just "6"? Who's to say?**



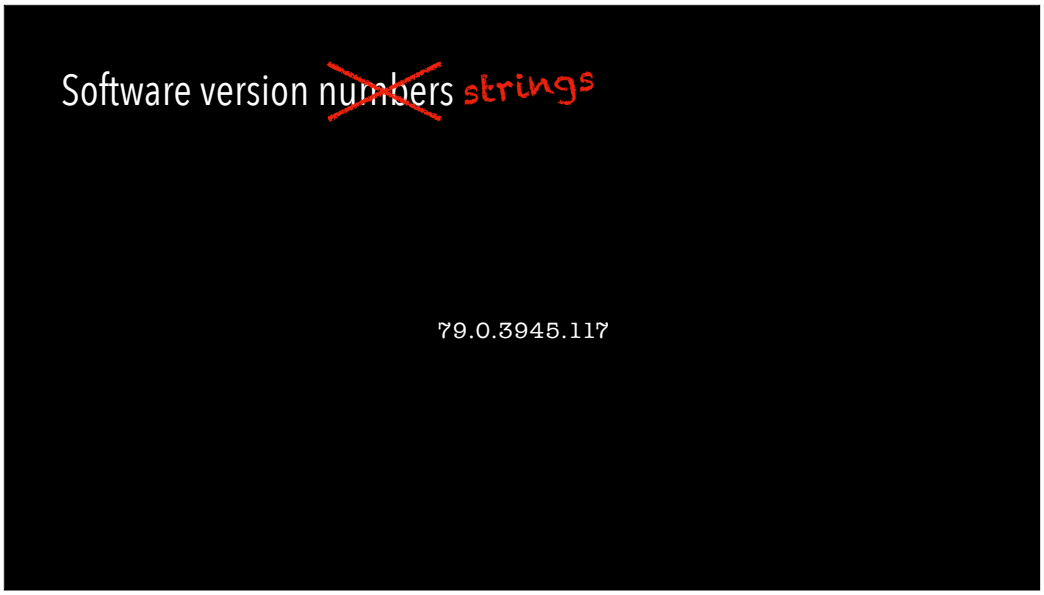
So regex to the rescue, right?

Very much so, but when you try to do something that sounds as simple as **"equal to or greater than a version string"**, it actually gets a lot hairier.



Let's take Google Chrome's version string.

Where do we start — at the beginning or the end?



At first, I started at the end. It's easy enough to **change the "7"** to a range of "7" to "9". That's already two possible higher versions right there.

But then what happens if the "1" before the "7" increases?

That means I'd **change it a range of "2" through "9"**. Right? Well, not so fast because now this won't match "117", which was the original number, or "118" or "119".

So **I'll solve that with a capture group** and an "or" operator.

But if I do that, the "2" through "9" and the "7-9" only match one character each. I've lost a number.

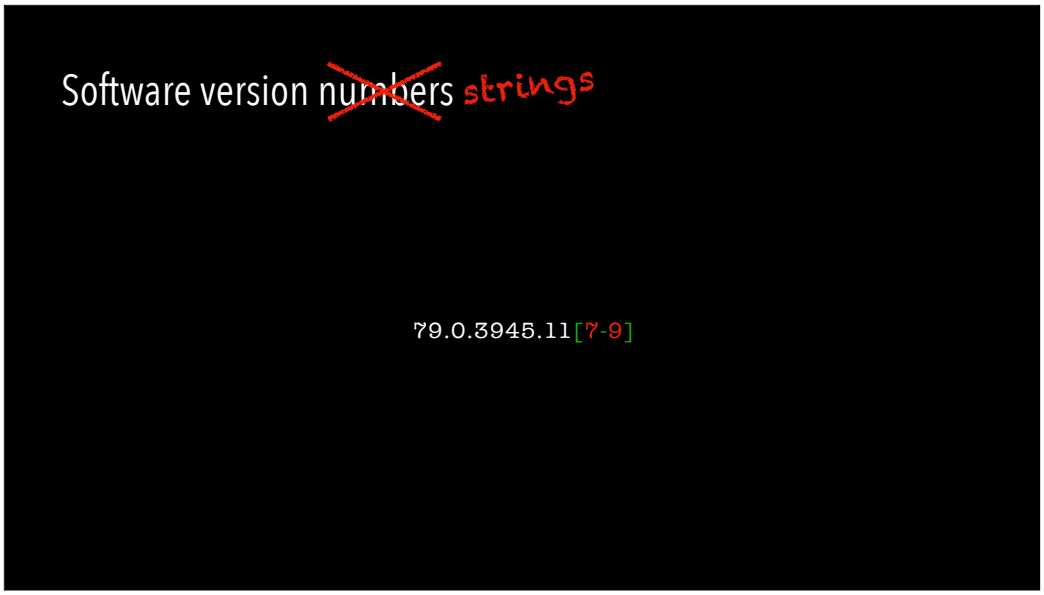
So, I can fix that by adding a **"\d"** to the end of the "2" through "9" range to indicate one more digit.

But that's still not right. I'm still missing a number if I'm going to match just the "7" through "9" at the end. **All I need to do here**

is just add the "1" back in front of it.

OK, that seems to take care of the last two numbers. We're accounting for whether the ending number "7" increases and we're accounting for the "1" in front of the "7" if it increases.

I went with that for a while and it was **tedious**! I also realized something else.



At first, I started at the end. It's easy enough to **change the "7"** to a range of "7" to "9". That's already two possible higher versions right there.

But then what happens if the "1" before the "7" increases?

That means I'd **change it a range of "2" through "9"**. Right? Well, not so fast because now this won't match "117", which was the original number, or "118" or "119".

So **I'll solve that with a capture group** and an "or" operator.

But if I do that, the "2" through "9" and the "7-9" only match one character each. I've lost a number.

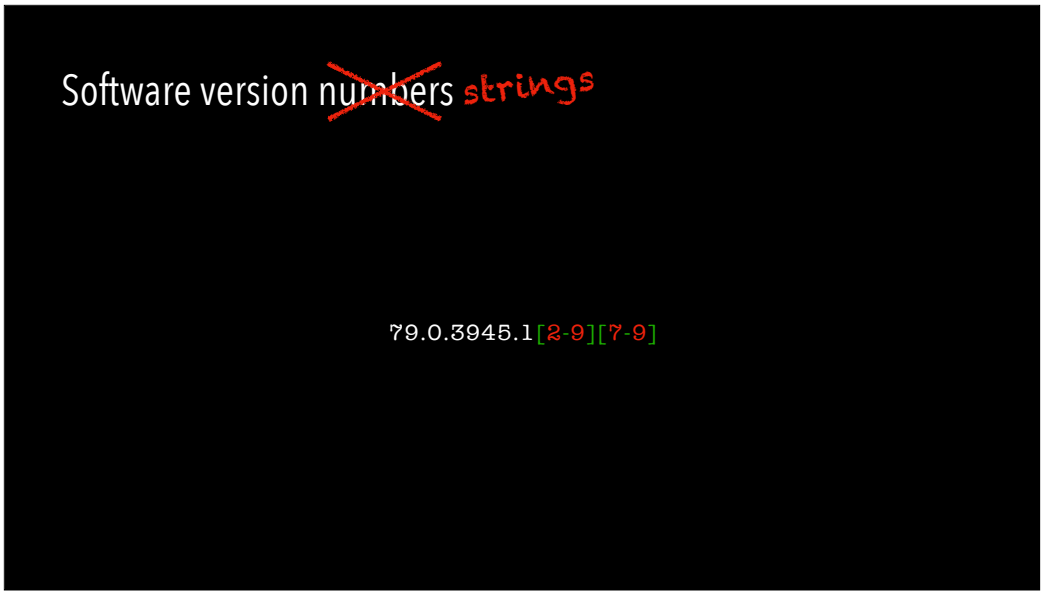
So, I can fix that by adding a **"\d"** to the end of the "2" through "9" range to indicate one more digit.

But that's still not right. I'm still missing a number if I'm going to match just the "7" through "9" at the end. **All I need to do here**

is just add the "1" back in front of it.

OK, that seems to take care of the last two numbers. We're accounting for whether the ending number "7" increases and we're accounting for the "1" in front of the "7" if it increases.

I went with that for a while and it was **tedious**! I also realized something else.



At first, I started at the end. It's easy enough to **change the "7"** to a range of "7" to "9". That's already two possible higher versions right there.

But then what happens if the "1" before the "7" increases?

That means I'd **change it a range of "2" through "9"**. Right? Well, not so fast because now this won't match "117", which was the original number, or "118" or "119".

So **I'll solve that with a capture group** and an "or" operator.

But if I do that, the "2" through "9" and the "7-9" only match one character each. I've lost a number.

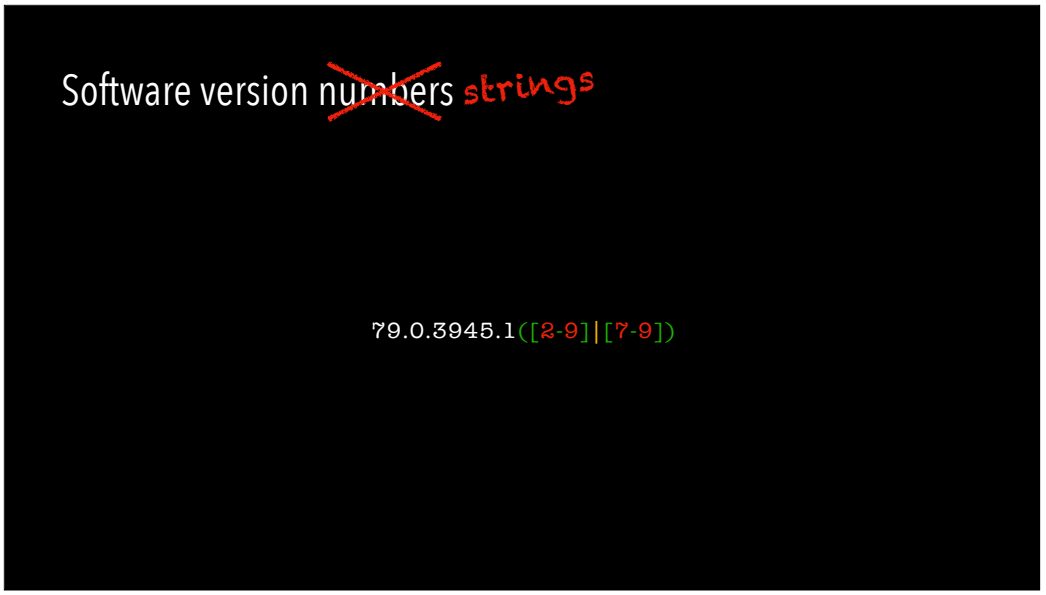
So, I can fix that by adding a **"\d"** to the end of the "2" through "9" range to indicate one more digit.

But that's still not right. I'm still missing a number if I'm going to match just the "7" through "9" at the end. **All I need to do here**

is just add the "1" back in front of it.

OK, that seems to take care of the last two numbers. We're accounting for whether the ending number "7" increases and we're accounting for the "1" in front of the "7" if it increases.

I went with that for a while and it was **tedious**! I also realized something else.



At first, I started at the end. It's easy enough to **change the "7"** to a range of "7" to "9". That's already two possible higher versions right there.

But then what happens if the "1" before the "7" increases?

That means I'd **change it a range of "2" through "9"**. Right? Well, not so fast because now this won't match "117", which was the original number, or "118" or "119".

So **I'll solve that with a capture group** and an "or" operator.

But if I do that, the "2" through "9" and the "7-9" only match one character each. I've lost a number.

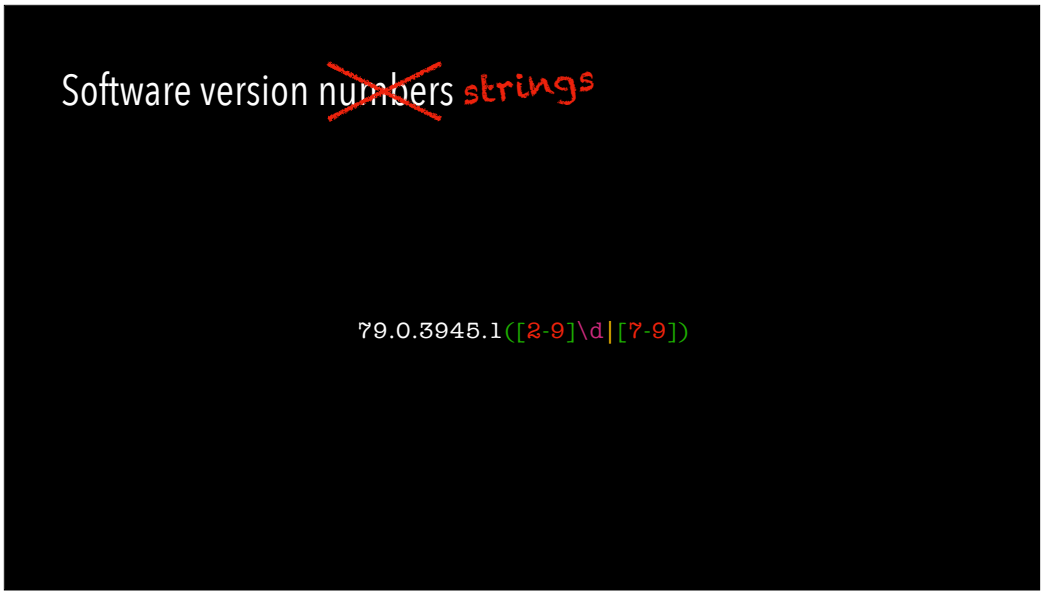
So, I can fix that by adding a **"\d"** to the end of the "2" through "9" range to indicate one more digit.

But that's still not right. I'm still missing a number if I'm going to match just the "7" through "9" at the end. **All I need to do here**

is just add the "1" back in front of it.

OK, that seems to take care of the last two numbers. We're accounting for whether the ending number "7" increases and we're accounting for the "1" in front of the "7" if it increases.

I went with that for a while and it was **tedious**! I also realized something else.



At first, I started at the end. It's easy enough to **change the "7"** to a range of "7" to "9". That's already two possible higher versions right there.

But then what happens if the "1" before the "7" increases?

That means I'd **change it a range of "2" through "9"**. Right? Well, not so fast because now this won't match "117", which was the original number, or "118" or "119".

So **I'll solve that with a capture group** and an "or" operator.

But if I do that, the "2" through "9" and the "7-9" only match one character each. I've lost a number.

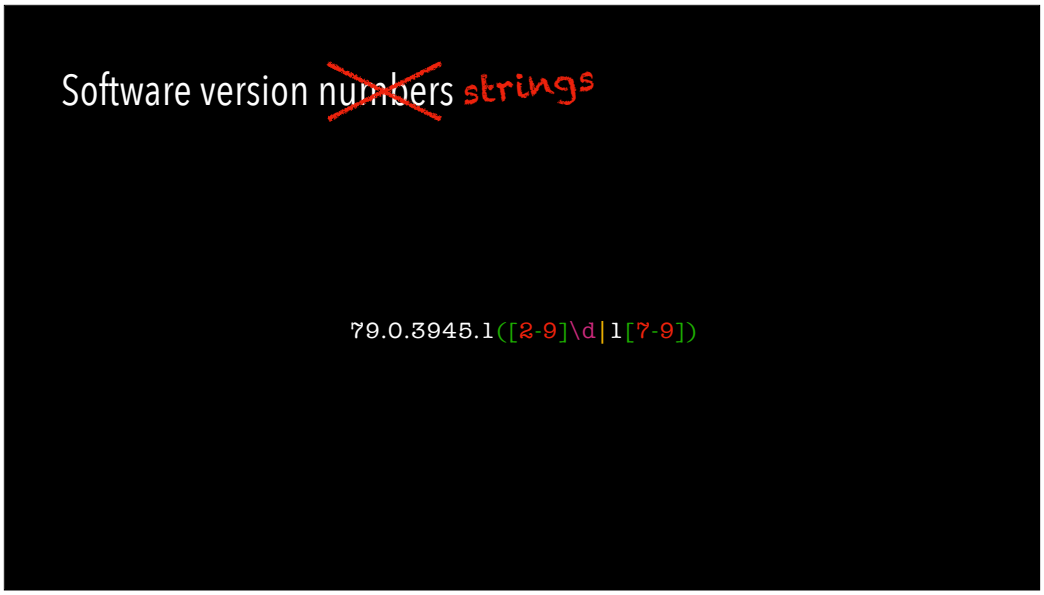
So, I can fix that by adding a **"\d"** to the end of the "2" through "9" range to indicate one more digit.

But that's still not right. I'm still missing a number if I'm going to match just the "7" through "9" at the end. **All I need to do here**

is just add the "1" back in front of it.

OK, that seems to take care of the last two numbers. We're accounting for whether the ending number "7" increases and we're accounting for the "1" in front of the "7" if it increases.

I went with that for a while and it was **tedious**! I also realized something else.



At first, I started at the end. It's easy enough to **change the "7"** to a range of "7" to "9". That's already two possible higher versions right there.

But then what happens if the "1" before the "7" increases?

That means I'd **change it a range of "2" through "9"**. Right? Well, not so fast because now this won't match "117", which was the original number, or "118" or "119".

So **I'll solve that with a capture group** and an "or" operator.

But if I do that, the "2" through "9" and the "7-9" only match one character each. I've lost a number.

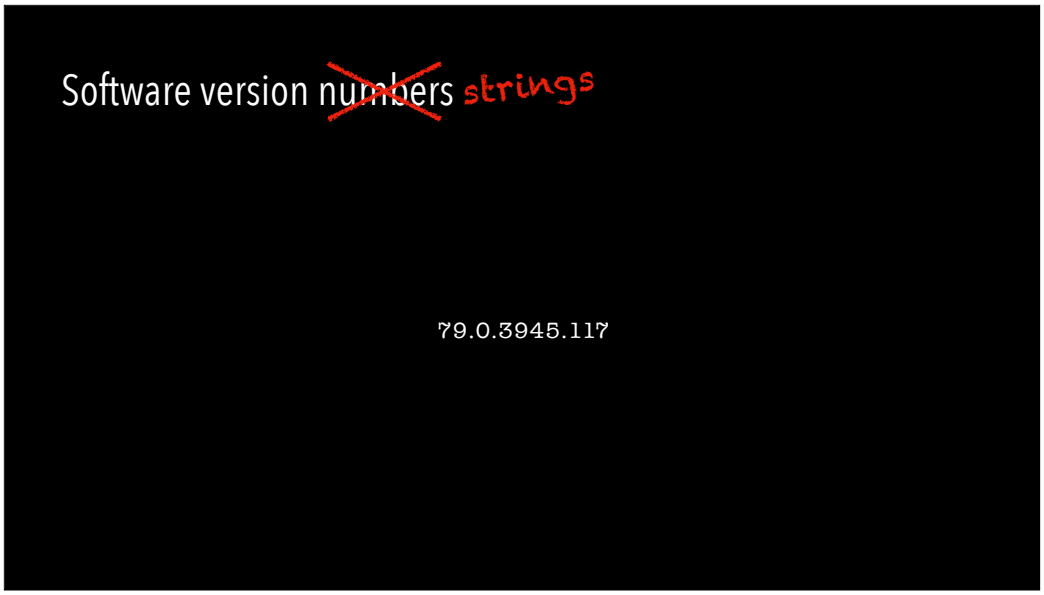
So, I can fix that by adding a **"\d"** to the end of the "2" through "9" range to indicate one more digit.

But that's still not right. I'm still missing a number if I'm going to match just the "7" through "9" at the end. **All I need to do here**

is just add the "1" back in front of it.

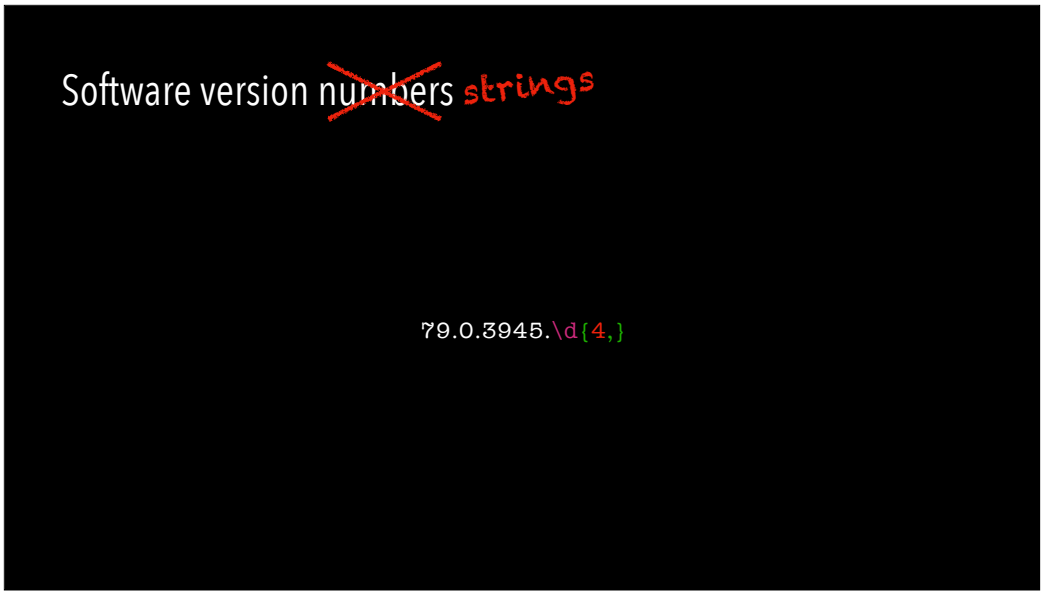
OK, that seems to take care of the last two numbers. We're accounting for whether the ending number "7" increases and we're accounting for the "1" in front of the "7" if it increases.

I went with that for a while and it was **tedious**! I also realized something else.



If I went back to the original string that ended in 117, what's to say Google will stick with just three characters at the end? What if that "117" were "999"? It could roll over to "1000" or "10,000" or "100,000".

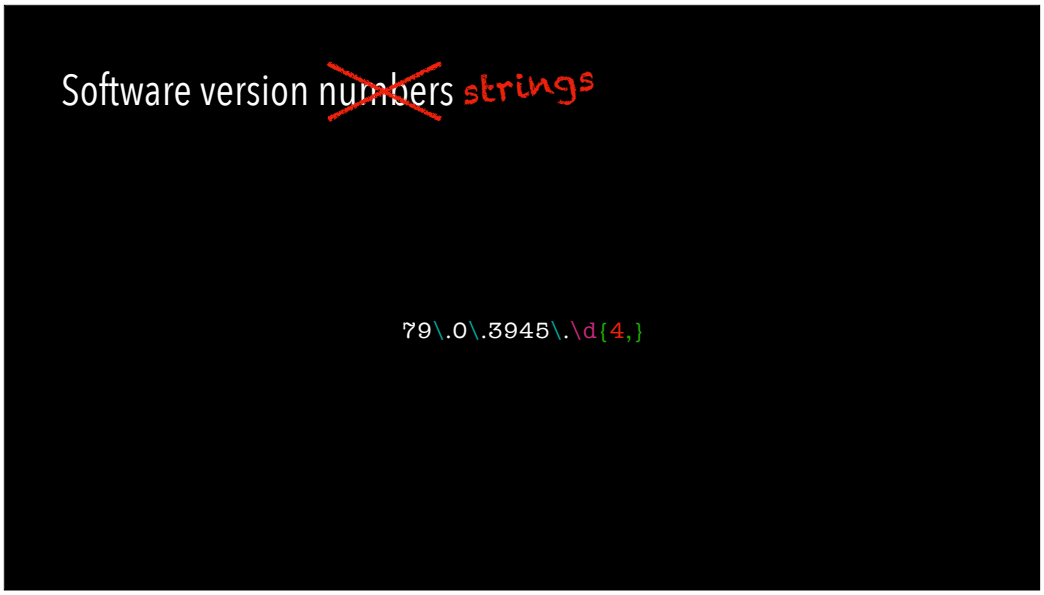
Another part of my regex would have to account for that.



I had to include the possibility the last sequence would be four or more digits.

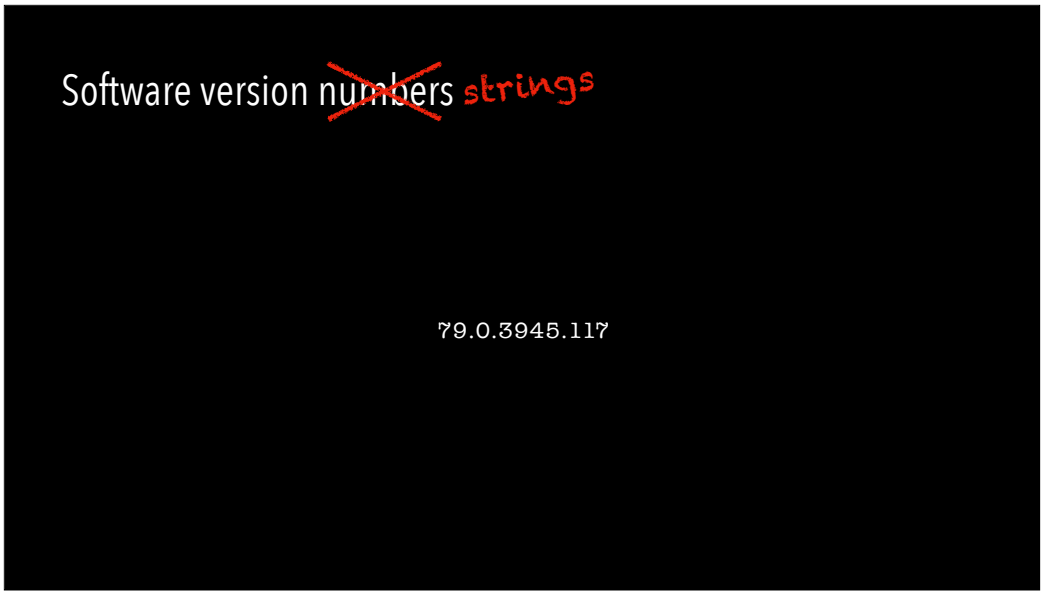
Here's something new with those repetition braces. If I specify a minimum number of characters but don't specify a maximum number, that's the equivalent to four characters or higher. So, I have to make sure I include something like this for each sequence of numbers between the periods.

And what about the periods?

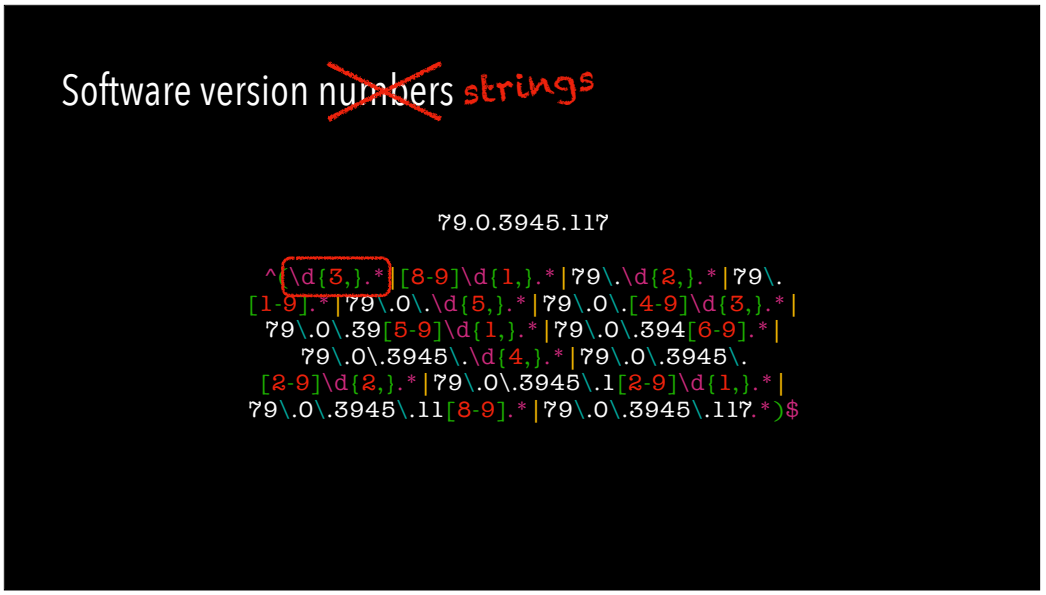


I need to make sure I escape each of those because periods are special regex characters that match any single character. I want them to just be periods.

I hope you get the point that not only was this tedious, there were a lot of gotchas that I had to account for.



The regex to match this version string or higher turned out to be a lot more than I'd anticipated and I eventually changed by my tactic to start at the beginning of the version string instead of the end. That made this a lot easier.



Here it is.

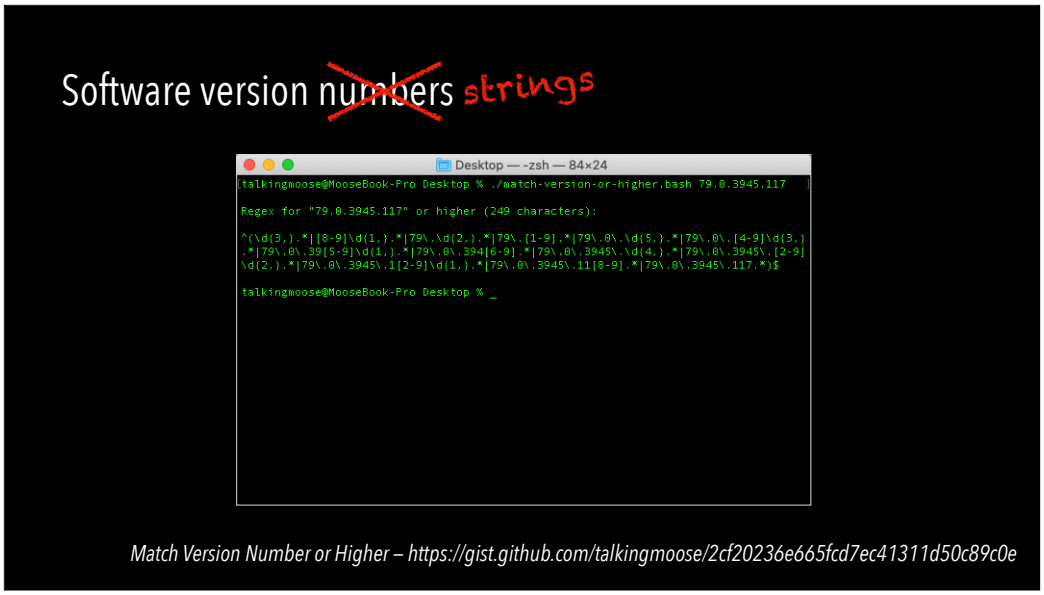
...

This is a very **complete** regex, but it's not really that **complex**. You may notice there are a lot of **"or"** separators in this pattern.

That's because this is just a bunch of shorter regex patterns strung together and each one accounts for a possibility that I mentioned earlier, such as the number of characters in a sequence changing from "2" to "3" or "3" to "4". **If you look at this very first pattern**, you'll see it accounts for the "79" at the beginning of the version string growing from two digits to three or more digits and then I just added a **".*"** to the end to match everything else.

It also accounts for each number in each sequence incrementing, such as from "6" to "7" or higher or "7" to "8" and higher.

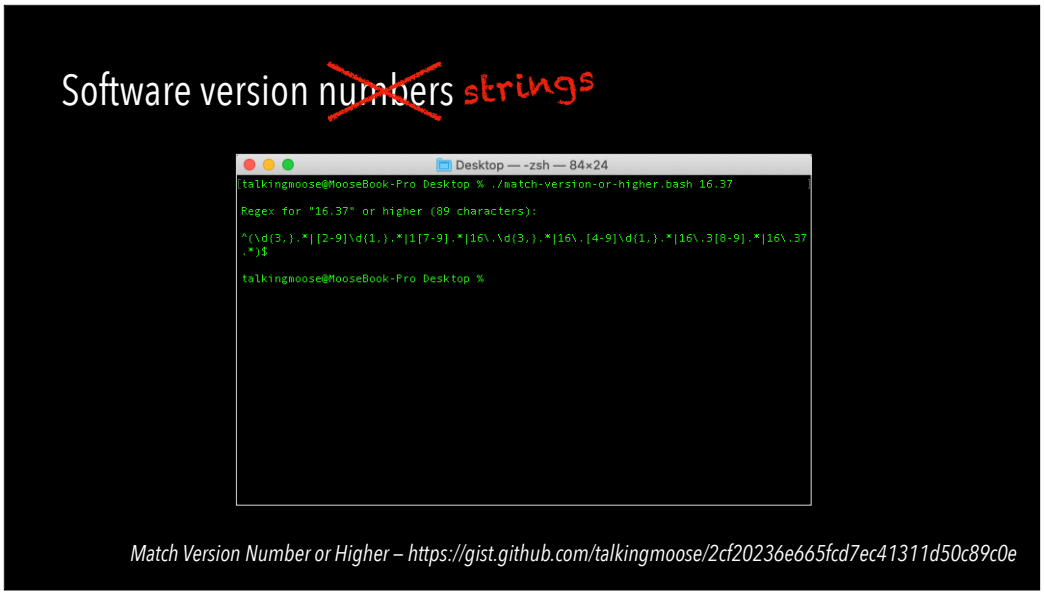
All I had to do was work my way from left to right and I found that for each sequence of numbers or each number, there was only one of four possibilities. I had found a **formula** for creating the patterns.



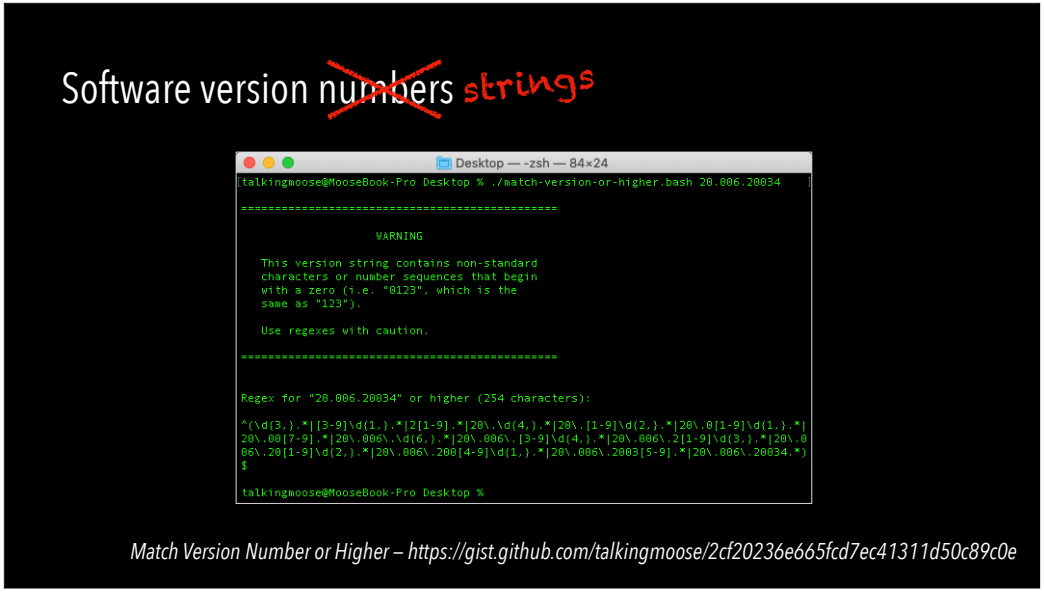
And once you know a formula, you can automate it.

So, I spent some time and **wrote a script** to create a regex for this very specific purpose of matching a specific version number or higher. It'll take most any version string you give it.

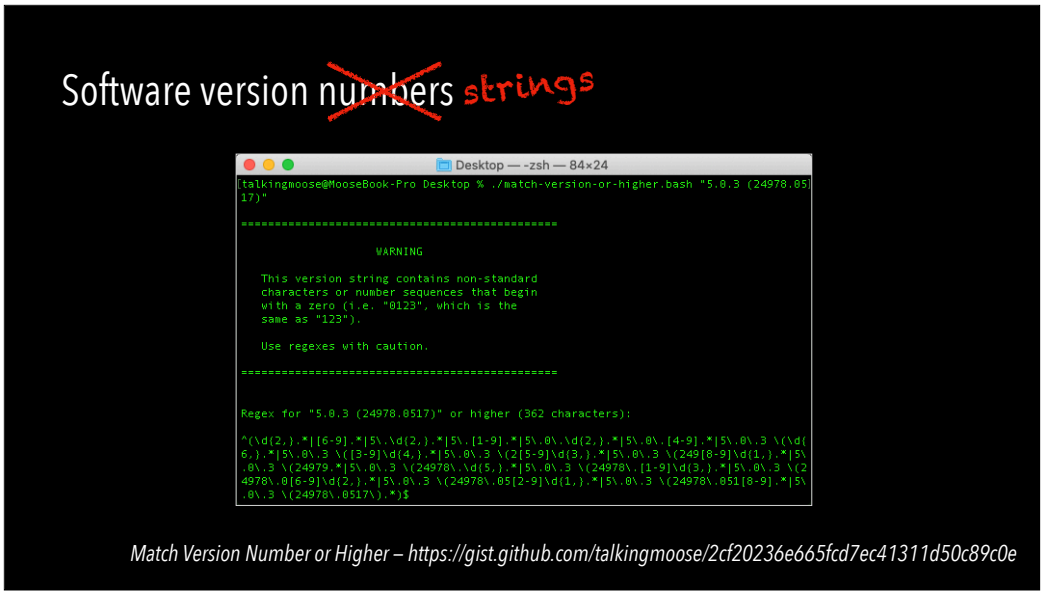
This one's for Google Chrome.



This one's for Microsoft Office. You'll see it's a **lot** shorter.



This one's for Adobe Acrobat Reader DC. You'll notice that I'm giving you a warning about this regex because the middle sequence in the version string contains leading zeroes. Is this "6" or is it ".006"? We have no idea what this developer is doing.

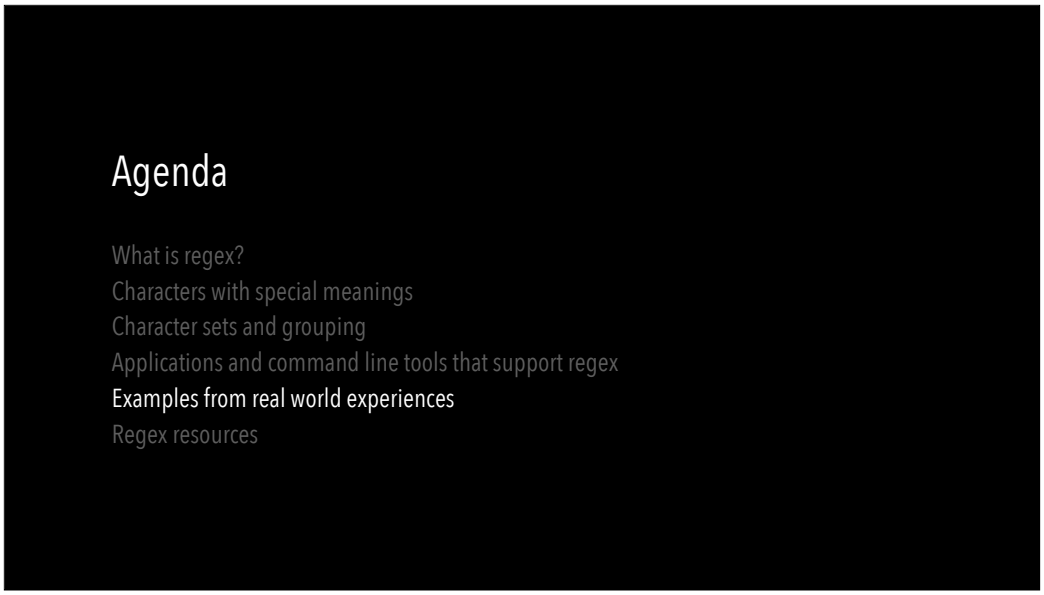


And finally, Zoom. With its spaces and parentheses.

Oh, Zoom, what are you thinking? The world may never know.

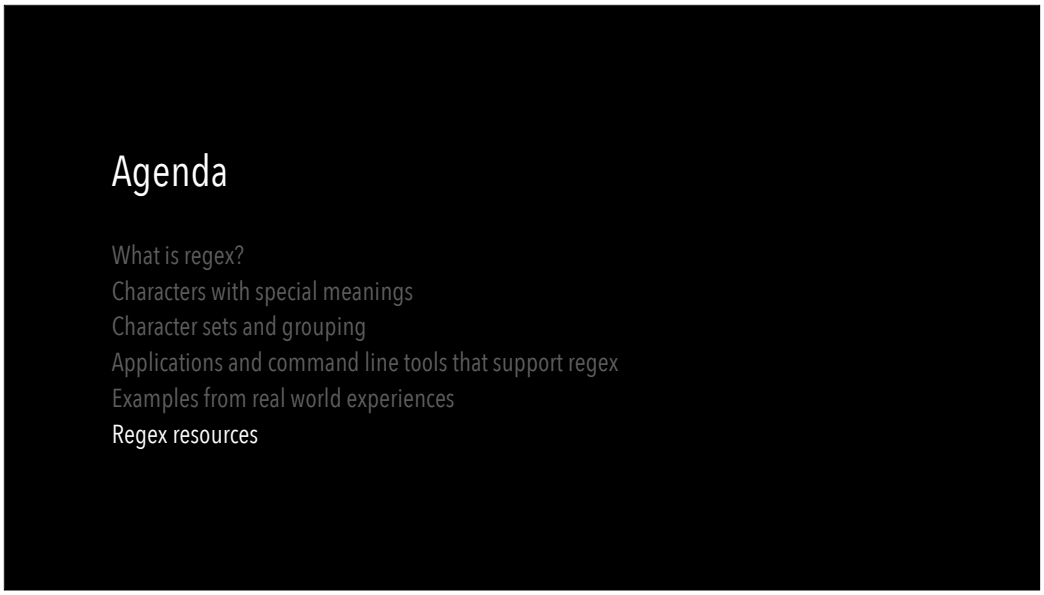
I've updated my script to account for version strings with non-standard characters, but again I show a warning because when developers don't follow standards, we can't design patterns or scripts that will make them standard. Use with caution.

The link to the script is at the bottom, but I'll also include it in my resources at the end .

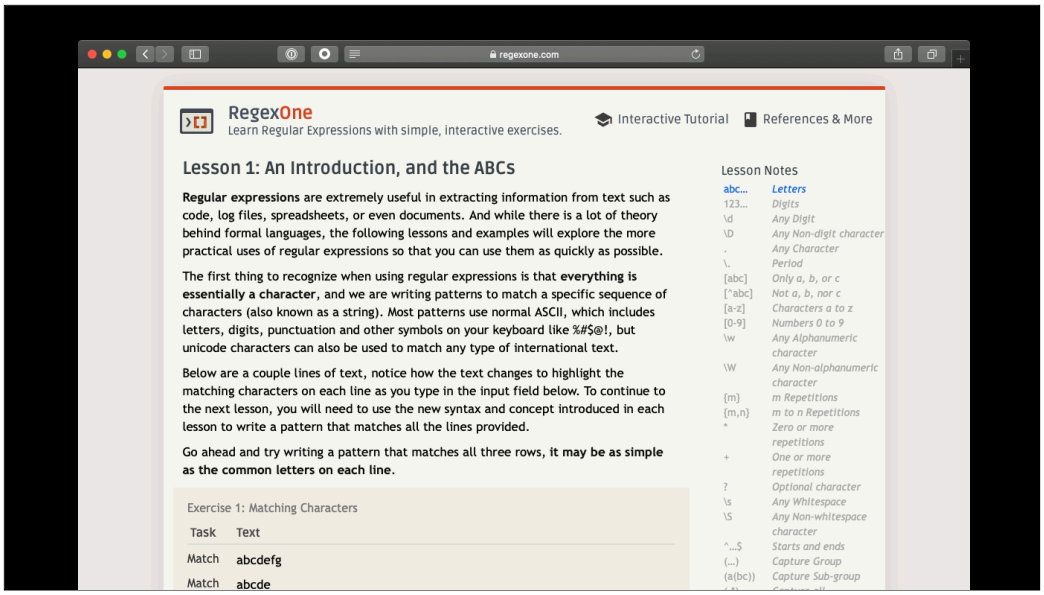


Those were a couple of examples for using regex in the real world and the regex version script is **extreme**! It's a little bit of a warning but also a little bit of a consolation that not all regex patterns will be short and sweet.

The more complex the string we need to match, the longer the pattern will likely get. So, I'll reiterate what I said earlier a couple of times. It's far easier to write a regex than to read someone else's regex. But look for characters and shortcuts that you recognize and **look up** those that you don't. It's like looking up a word in the dictionary. It makes sense once you know the definition.



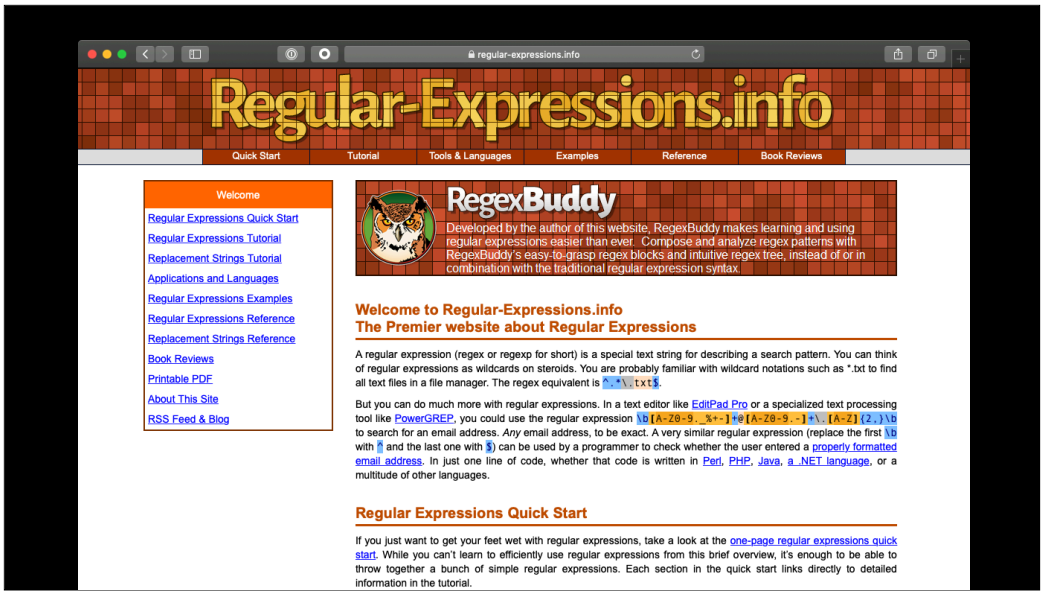
Now, here's a quick list of resources to help you with building regexes.



If this is your first journey into regex or if you want to solidify and expand what you know, go to RegexOne.com.

Each lesson takes about five minutes and concentrates on a specific concept like the "\d" and "\w" shortcuts or ranges or capture groups. If I've gotten you interested at all in regex, this is the next place for you to go. It makes learning regex fun.

RegexOne.com.



Once you've learned the basics from RegexOne, Regular-Expressions.info is a great resource for detailed and more extensive uses for regex.

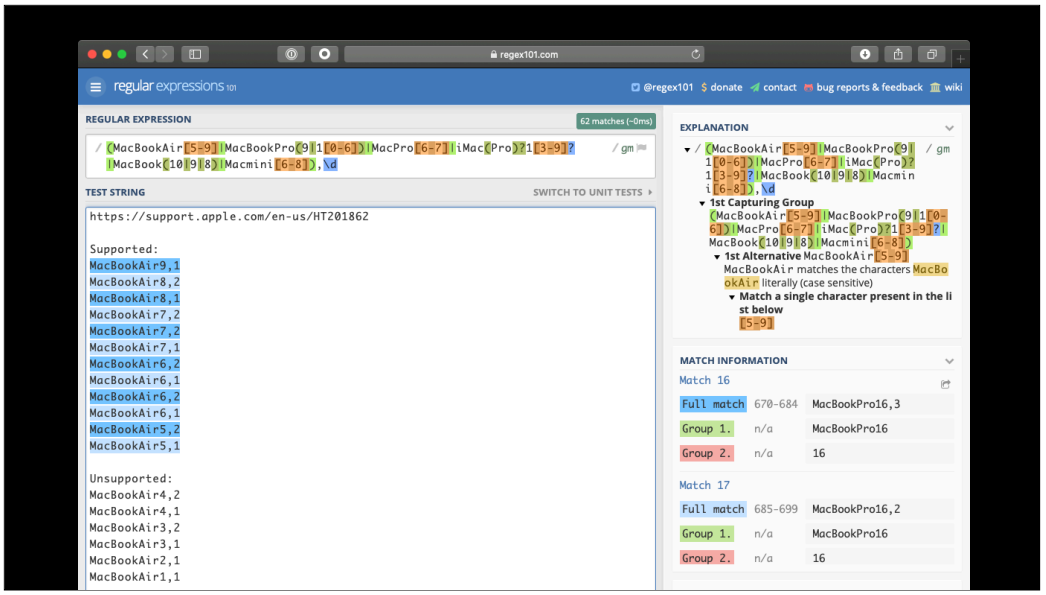
While I don't recommend it because the site is really optimized for online use, you can download a 400-page PDF with all the content.

There's even a list of book reviews for regex books **like O'Reilly's Mastering Regular Expressions**. But don't let the publication date turn you off. Even though it was first published in 2002 and later updated in 2006, regex really hasn't changed that much in the meantime. The content is still as relevant today as it was 15 years ago. In fact, you'll probably have a hard time finding something that's newer that's not already covered in some of the older books.

Once you've learned the basics from RegexOne, Regular-Expressions.info is a great resource for detailed and more extensive uses for regex.

While I don't recommend it because the site is really optimized for online use, you can download a 400-page PDF with all the content.

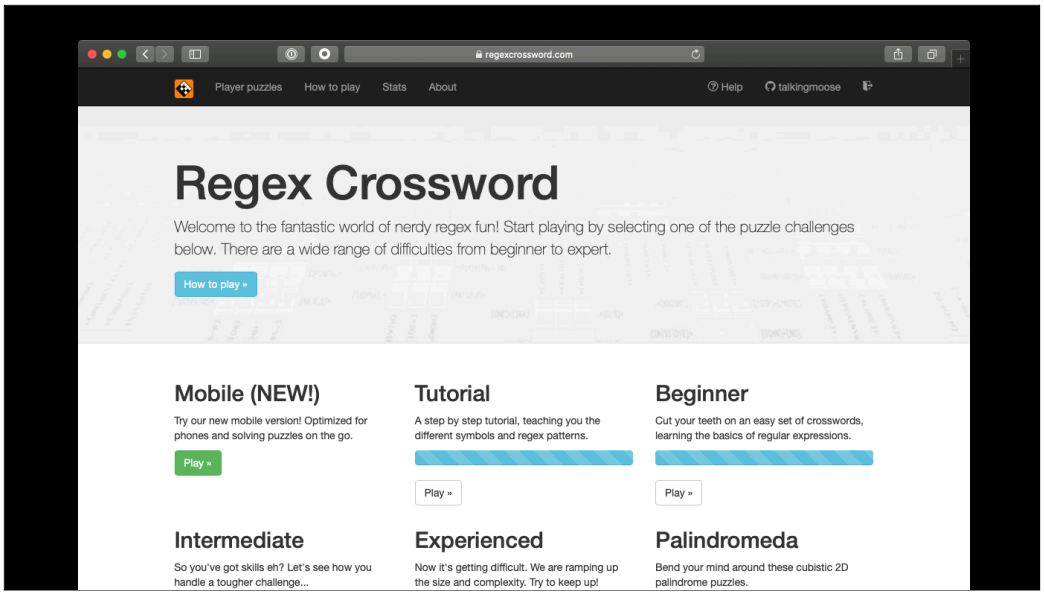
There's even a list of book reviews for regex books **like O'Reilly's Mastering Regular Expressions**. But don't let the publication date turn you off. Even though it was first published in 2002 and later updated in 2006, regex really hasn't changed that much in the meantime. The content is still as relevant today as it was 15 years ago. In fact, you'll probably have a hard time finding something that's newer that's not already covered in some of the older books.



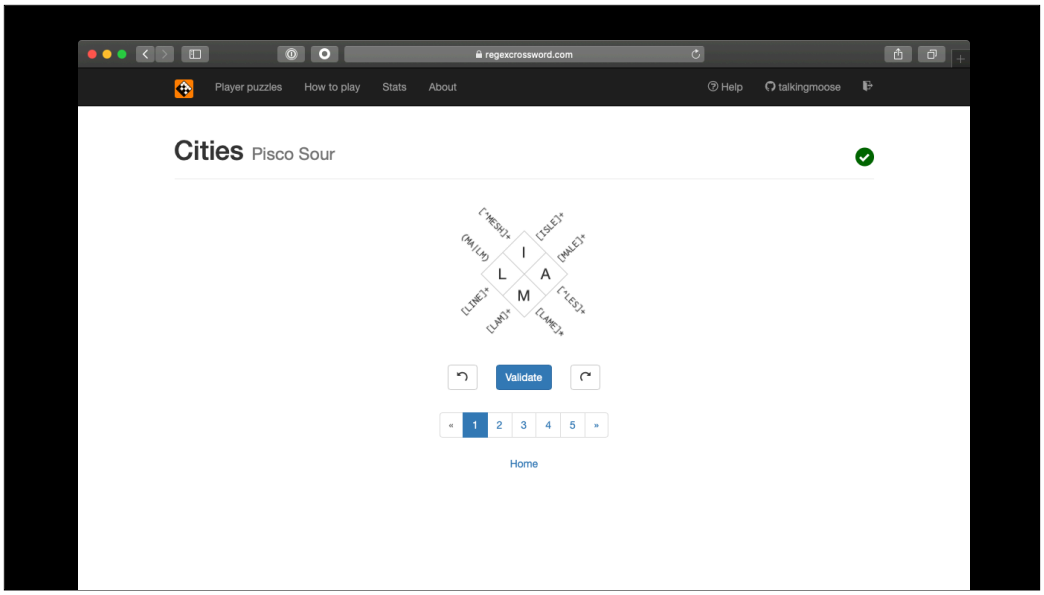
I've mentioned a couple of times Regex101.com, which lets you test your regex patterns and experiment. The explanations to the right are useful when you're trying to understand why something does or doesn't work.

You'll find dozen's of sites like this for regex validation. This one just happens to have an interface that I find easy to use.

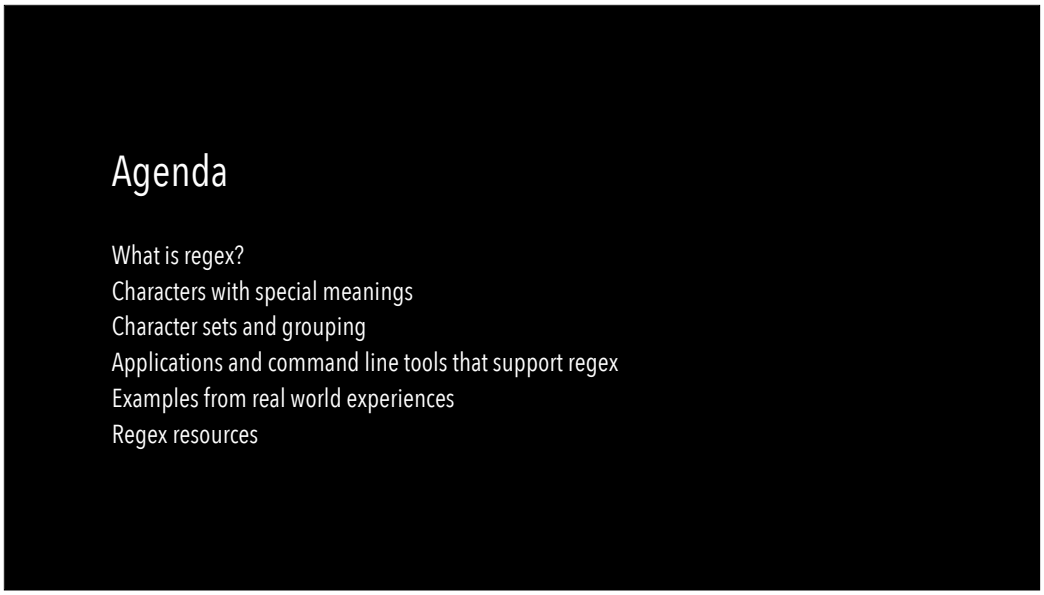
Regex101.com.



And finally, if you want to practice your regex and have fun at the same time, head over to RegexCrossword.com.



They have dozens of crosswords across several themes like Cities. So, if you understand the clue, then you're on your way toward solving the puzzle. If you don't understand the clue, you've got regex to help you along.



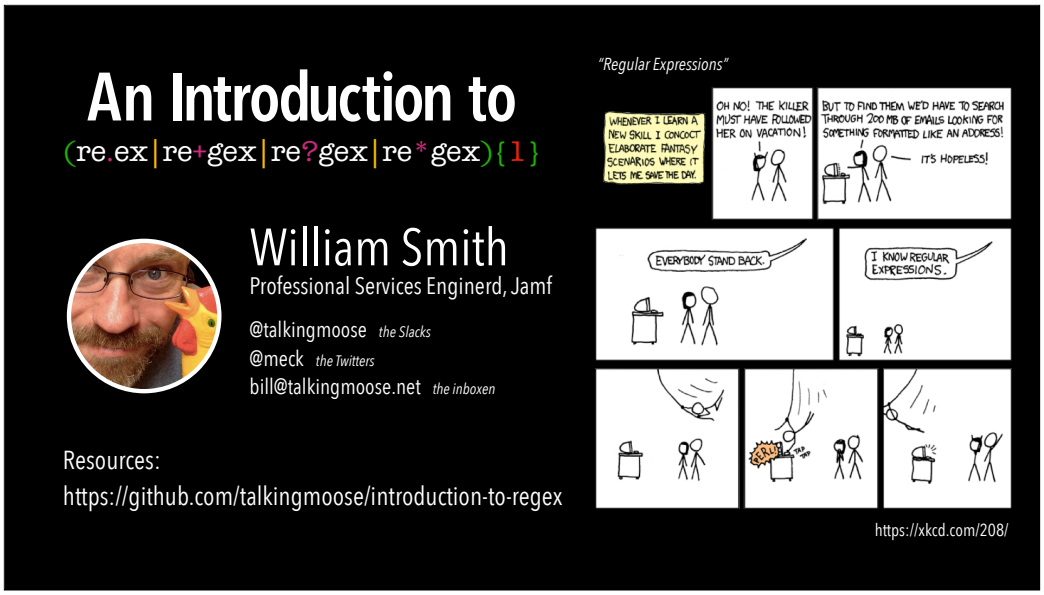
Remember, regex is about understanding the language of something like an email address, a phone number or a zip code. The words that make up these languages are structured, which makes them **"regular"**. And if they follow a convention for putting characters together to make words, then we can identify patterns. It's the ability to identify patterns that lets us validate whether words are part of a language.

Patterns in regex are made of literal characters, as well as special characters. Sometimes a character in a pattern matches a **single character** in the string we're validating. That's a basic regex. And sometimes special characters and sets and groups in a pattern can match **multiple characters** in the string. That's an extended regex.

You may not have realized it, but you've probably been using some tools for a long time that include regex support, especially the grep command, which in itself is about regular expressions.

I've shown you many examples from my real world experiences using regex and hope they give you an idea for how you can use it in your world. The web is full of resources for you to learn and understand it.

Regex isn't the new kid on the block, but it could become your new best friend when having to solve problems with data.



Here's a link to the resources I mentioned today. I'll have the slides and files posted there shortly.

Let's see if you have any questions.