Course Overview

Hello everyone, my name is Josh Duffney and welcome to my course, Introduction to Regular Expression. I am a DevOps engineer at a software company called Paylocity. Most people consider regular expression to be some form of magic that they can use, but don't understand. This course demystifies the notion of magic around regular expression by teaching you how to use metacharacters and meta-sequences in regular expression to craft meaningful expressions. In this course, you will learn how to use shorthand metacharacters, anchors, word boundaries, lookarounds, and much, much more. By the end of the course, you will have the knowledge required to both read regular expressions you find online, as well as how to craft elegant expressions of your own. I hope you take this opportunity to stretch yourself a little bit and join me on this journey to learn regular expression. Who knows? Maybe you'll even become a wizard of regex.

Using Wildcards and Literal Matches

Introduction

Hello and welcome. My name is Josh Duffney, and this is my Pluralsight course titled Introduction to Regular Expression. Regular expression is often referred to as wizardry or magic. However, magic is a mystery to everyone but the magician. To the magician, it's a simple routine practiced over and over until it has become so effortless it appears as magic. The same is true for regular expression. To the untrained eye, regular expression seems, well, a bit magical. But with just a little bit of practice, you will start to see through the smoke and mirrors of regular expression. Since this course is an introduction to regular expression, it is applicable to a wide audience. If you're an IT professional, you're probably familiar with this scenario. Your management provides you a list of email addresses that were copied straight out of a distribution group. Sadly, this list also includes junk data, and the email addresses you need are stuck somewhere inside. If this sounds familiar, this course can help you learn to use regular expression to extract out those email addresses and discard that junk data. Or perhaps you've become quite good at scripting, and you're a scripter or an automater. Either way, this course can help you solve more complex problems, such as determining the domain name of a user from their distinguished name in Active Directory. Or you could use it to validate parameter inputs of your advanced functions. You could even use it to validate the status of your infrastructure with something like Pester. Maybe you're not an operations person at all and you're more of a developer. This course can also help you. The point I'm trying to make is that if you deal with data of any kind, knowing regular expression will help you understand that data and it will make you much more productive when working with that data. Before we begin, I

want to give you a feel for the course structure. I'll always start with the syntax of whatever topic we're covering. This allows you to get an idea of what the syntax looks like and how it might be applied. This is also where I'll take a little bit more time and I'll walk through what the syntax is actually doing and what makes up those regular expressions. I will then take those syntax examples and use them inside of a live demo. I'll refer to this as the academic demonstration of the syntax. Lastly, I'll show you how to apply the knowledge you gain from each module to real world, practical examples of using regular expression. The very last thing that I want to cover before we get started with this course is the lab setup. The lab setup for this course is very simple. It consists of a single virtual machine on a Hyper-V host. I've chosen Hyper-V to be my hypervisor, but you can use whichever one you like. The virtual machine that I will be using is running Windows Server 2016. However, Windows Server 2012 R2 will work just fine. The server I am using has Active Directory domain services installed and is a domain controller for the domain Globomantics. Internet access is optional. If you don't have those Windows features or modules to use DSC to promote your domain controller automatically for you, you might want to have some Internet access to the VM, but it is not necessary. For assistance in setting up the lab environment, please reference the course's exercise files. Now that you have an understanding of the course structure and how to set up the lab environment, it's time to talk about what regular expression actually is. Regular expression is a special sequence of characters that define a search pattern. Those search patterns are then used for pattern matching within a string. This might be a bit confusing, but it's actually a really good definition. And if you think about, us humans do something very similar when we search for a pattern in a blob of text. For example, say you were looking for your name on a roster. To find your name, you would scan the document for your first name followed by a space and then your last name, and to be more specific, you would start by looking for the first letter of your first name followed by the second letter and so on. Then there would be a space separating your first and last name and then hopefully there would be the characters of your last name. Regular expression is a mechanism or a language you would use to make the computer do this for you. If this still isn't making complete sense, that's fine. That's what this course is for. All you need is a little bit more context. So let's move on to the first topic of this course, wildcards.

Wildcards

When I first started diving into regular expression, I was shocked to learn that I'd been using it for years. Since the beginning of my career, I've been using wildcards. Wildcards are a form of regular expression. I just never considered them to be regular expressions. But what are wildcards and what do they do? Well, a wildcard is usually represented by an asterisk, and it is used to represent any character any number of times. That is why it is called a wildcard. Taking

a look at the first example, I'm using Get-ChildItem to retrieve a list of all the text files found on the system. To do that, I'm using the asterisk or wildcard followed by four literal characters, .txt .txt is the file extension for a text file, so when this runs, it's going to retrieve a list of all text files. In the next example, I'm using an alias for Get-ChildItem called gci. I'm then using two wildcards on each side of the word demos. What this will do is it will retrieve a list of files and folders with the word demos inside of it, regardless of file extension or even file type. In the last example, I'm using Get-Command with a series of wildcards in between a commandlet name. We'll take a look at what this does in the upcoming demo.

Using Wildcards

Before I jump into the demo, let's take a quick look at the topics that I'll be covering. We'll start by searching for files. We'll search for file extensions and also for words inside a file name. After that, we'll look at finding services with wildcards, using wildcards with parameters as well as with the where-object commandlet. I'll wrap up the demo by demonstrating how to use wildcards to search in Active Directory. We'll use wildcards to look for users withinside Active Directory, as well as how to find a user's manager. The first thing that I'll cover in this demo is how to use the wildcard to search for text files on the system. Before we do that, on line four I will change into a directory where I know that I have some text files to take a look at. Now that I'm inside the directory where the text files are, I'll go ahead and run line six, which runs Get-ChildItem in conjunction with a wildcard followed by .txt As you can see, it only returned text files, as I expected. Just to ensure you I don't have any tricks up my sleeve, I'll go ahead and run Get-ChildItem without the wildcard just to show you that there are a few other file extension types in there other than .txt This just demonstrates that the wildcard was working properly to filter out only the text files. I know this is really simple and it's probably even something that you've done before, but I assure you it gets better. So let's move on to the next example where I'm going to use Get-ChildItem again but to look for any file or folder that contains the word demos. To do that, I will change into a different directory where I know that I have some demo scripts. Then I'll go ahead and run line 11. As you can see, we had two results return to us. Both of them are PowerShell files, and they contain the word demos inside the file name. It's worth mentioning with the way that I use these wildcards for this example that demos could've been located anywhere. It could've been at the front. It could've been at the end. It could've even been the file extension. It would've found it, because what I'm saying when I use asterisk or wildcard, demos, and then another asterisk, the wildcard, I'm saying, "Find demos anywhere withinside the file name." The next example that we'll take a look at is how to use wildcards with the commandlet Get-Command. Before I run this, take a moment and try to think about what this command is going to return. So I'm using Get-Command. That obviously returns back

PowerShell commandlets that are available to you, but I'm using a series of wildcards. So the first one is before et, so if you're familiar with PowerShell, it's a verb dash noun syntax. So there are several verbs that end in et. So think about what those might be. Then, I'm looking for any commandlet that has the word net inside of it. So there are probably going to be quite a few when I run this, so let's go ahead and take a look. Now that the commandlet has finished running, let's take a quick look at the results. There are two obvious verbs that you'll see here, get and set. Those both end in et, and they're pretty common in the PowerShell language, so we're going to see a lot of them. You'll also notice that net is somewhere in the commandlet name. It's after hyphen, but then it seems kind of random throughout the rest because I used a wildcard before and after the word net, so it can appear anywhere. It could be at the beginning, it could be at the end, or the middle. One commandlet that showed up that kind of surprised me was reset. Reset is not very common, I would say. At least, I haven't run across it too many times in my experience with PowerShell. But it does match, right? It ends in et, so it matches the wildcard because the wildcard does represent any character any number of times, so the wildcard matched res and then the literal characters, which we'll talk more about later, matched et- So there you have it. That's why it did match the reset commandlets. Let's move on now and take a look at using Get-Service with some wildcards. A little bit more applicable and practical examples. So, we'll go down here, scroll down to Get-Service. We'll go ahead and run this, and it's going to retrieve a list of all of the services that the name contains net somewhere in it. As you can see, a list of several services did come out. Let me introduce you, though, to another way of doing this. So it's up to the parameters if they supports wildcards or not, but one thing that you can always use wildcards with is where-object. So we're going to do this same exact thing, except we're going to take the result of Get-Service and pipe it to where-object. So where-object, then, is going to filter the name where it is like net anywhere in the name. Produces the same exact results, but just another way to do it. Now if we want to get really tricky, well, it's not too tricky, but we can use Get-Service with wildcards and then pipe it to where-object and then filter the status where it's like s. So it's going to filter out all the ones that are stopped. So when we run this, we get exactly what I said we were going to get. We were going to get all the stopped services, because the only two statuses in there were running and stopped, and running doesn't contain an s anywhere, so that's why it returned stopped. Now, let's move on to some more fun examples, working with Active Directory and wildcards. So, in a previous life, I managed Active Directory for a large organization, and I used the Active Directory module every single day. I can also tell you firsthand, without PowerShell, the Active Directory module, and knowing regular expression, I wouldn't have been able to do my job. With that said, let's go ahead and take a look at line 24. What I'm doing here is I'm using the Get-ADUser commandlet and then I'm going to filter out all the users where their name is like green or contains green. So let's go ahead and run that. And we should get just one user returned to us. So Jason Green was the lucky user that got returned. He's the only one in the

company with the last name Green. So I'm going to let you in on a little secret. I am actually a terrible speller, and I use this method all the time to find Active Directory users in the environment, whether I needed their email or I needed to find their manager, which actually just so happens to be the next thing that we're going to do. So on line 26, I'm using Get-ADUser again, but I'm filtering on the Manager property. So I'm looking for any user who has the manager Smith, or a manager with the name Smith, rather. So as it turns out, when you're using the Manager property inside the filter parameter, it does not allow you to use anything other than equal or not equal. So, just like I said before, there's always more than one way to do something. So if we scroll down just a little bit, you can see that I presented you with two options. Option one on line 28 is to use the equal operator and the full distinguished name, which is kind of annoying, but if we run it, it does do what we want it to do. So when we run that, it does work. It returns all the people whose manager is Smith, but we had to use the exact distinguished name, so that's not very dynamic. It could be S. Smith or J. Smith. We want anybody whose managed with Smith. So let's go on to line 30, and we'll use GetADUser again, and then we're going to filter star, which is not the most efficient way to do it, but it gets the job done. So we're taking all of that, and then we're selecting out the manager's property, so we're adding that to the collection, and then we're using where-object on that Manager property and then filtering it with Smith. So when we do this, we get the same result, but it's a little bit more dynamic. If I had more people in this Active Directory environment or in the company, it would've returned back more people with the last name Smith or the first name Smith. So there you have it. That wraps up this module on wildcards. Next, we'll take a look at what literal characters mean in regular expression, as well as the match operator.

Literal Character

In regular expression, literal characters are used to match a single character, or in the case of a case insensitive engine, such as PowerShell, it can represent two characters, the upper and lower case of that character. Case sensitivity is something that can be turned off and on, but we'll talk more about that later. In the first example I have listed here, it demonstrates how to match the server name from a UNC path. I'm telling the regular expression engine to find a sequence of characters, GFS-01 This example also introduces you to one of the several ways to use regular expression in PowerShell, the match operator. In the second example, I'm using the match operator again. Instead, this time, I'm looking for a username, jgreen, inside a user's distinguished name. In this last example, I'm demonstrating how to convert the end of a distinguished name to a more readable domain name by using the replace operator. This is yet just another operator that you can use or a way that you can use regular expression withinside PowerShell. Just like the match operator, I start by telling it what I'm looking for, which is, dc

and then an equal sign. However, after that, I'm telling it what to replace it with. In this case, it's a period. You'll get to see all of these in action in the upcoming demo. Here's what's covered in the next demo. I will start by talking about the basics of character matching with the literal characters. To do that, I will match a server name from withinside a UNC path. After that, I'll talk a little bit more about the match operator and how it behaves. I don't think it'll behave the way that you would expect it to right out of the gate. We'll also talk about something called $Matches This is a special variable that holds all of the match results from the match operator. I will then talk a little bit more about filtering Active Directory data, things like extracting a user's name from the distinguished name, and converting the end of a distinguished name to a more readable domain name with regular expression.

Using Literal Character with the Match and Replace Operator

Line five should look very familiar to you. It is the exact example that I used in the previous slide. What I'm doing here is I'm using the match operator to match the letters GFS-01 What this will do should be no surprise. It will match the server name from the UNC path in this text string. What might be surprising, though, is the results that are returned when I run line five. So let's go ahead and do that now. I don't know about you, but this wasn't what I expected. I expected the results of the match, not a Boolean value telling me whether or not it matched. But as it turns out, all of the matches are stored in a special variable called $Matches So if you want to find all of the results of your matches, you have to call out the matches variable, which is what I am doing on line seven. Now this is something more along the lines of what I was expecting when I ran line five for the first time. So what you see here withinside the matches variable is the match of the match operator itself. So we can see that GFS-01 did match withinside that string in line five. So there's our value, and we can extract that out from this variable when we need to. Moving on to the next example where we will take a look at extracting a username from a distinguished name. It's a very similar setup. What I have here is a user's distinguished name in the form of a string, starting on line 11, and then we're using the match operator again to just extract out the first section after CN= So we're matching J-G-R-E-E-N So we'll go ahead and run that now. And the same thing is true. It returns back a true value. If it didn't match at all, it would return to false. And if we take a look inside the matches variable, we will see that jgreen now is there. All right, jgreen is there, but where did the server name go? Well, whenever you run the match operator, it's going to overwrite that matches variable. Then you might ask the next question, well, what's the point if there's only going to be one match in there? Why is there an array of these things? Well, that's a good question, and we'll cover that later. The short answer to that question is you can actually have more than one match withinside a regular expression. But let's not get too far ahead of ourselves and let's just

move on to the next example where we take a look at extracting a domain name from a user's distinguished name with a replace operator. So before I run line 18, let's take a quick moment to walk through what's going on here. I have a user's distinguished name. It's actually the same distinguished name from above. And what I'm doing though is I'm matching the end of it, which is globomantics,dc=com So this is the end of the distinguished name that represents the domain name. So when I go ahead and run this, I will get a true value, because it'll go ahead and match that section of the string. But before I run line 20, let's take a look inside the matches variable just to see what the value is, just to see what it matched. So I'll take a look inside there. So we have globomantics,dc=com the same thing that I told the match operator to look for, which is great, right? So on line 20, I'm using the matches operator. I'm pulling up the first item in that array, which is the one that you see here, the Globomantics one. Then I'm using the replace operator to replace the comma, the D, the C, and the equal sign all with a period. So let's go ahead and do that now and see what it looks like. The result that I get is a much more legible and readable domain name. That wraps up the rest of this demo and the module, for that matter. So let's head over to the summary slide and review all the topics that we learned throughout this module. I started off the module by defining what regular expression is. I then introduced you to what wildcards are and how to use them withinside PowerShell. After that, I talked about how to use literal characters in regular expression to find matches. To find those matches, I demonstrated how to use the match operator, as well as the replace operator, withinside PowerShell. I know that I used a lot of simple examples throughout this first module, but I promise, things will get more interesting in the next module where you'll learn about character classes in regular expression.

Using Character Classes with Regular Expression

Introduction to Character Classes

Hello, and welcome to this module titled Using Character Classes within Regular Expressions. Character classes are used in Regular Expression to specify multiple values at a given point in a match. Using character classes will make your expressions more dynamic and easier to read. So what is a character class, and how do I use it? Well, the best way to understand it is to see an example of it. Let's say, for instance, you wanted to match the word Grey, spelled G-R-E-Y. That would be pretty easy to match with some literal characters. You would just spell out the word gray. However, there's a different way to spell gray, which is G-R-A-Y, so how can you craft a regular expression that deals with both variants without having to use two separate expressions? The answer to that question is what's called a character class. A simple way to think of a character class is a basic or statement. So, in this expression, where it says Gr it's actually saying, match G, followed by an R, and then an A or an E, but not both. One or the

other. And then ending with a Y. Now that you have a better idea of what a character class is and how you might use it, let's move onto the next slide and define a character class a little bit better. A character class is a set of characters used to express all the possibilities at a specific point in a match. They are represented by square brackets, and inside those square brackets are the characters that will be used to attempt a match. In the example, I'm using a singular regular expression, Gr[ae]y to match two possible spellings of the word gray. Keep in mind that this character class, ae, does not mean look for an A followed by an E, but rather, look for an A or an E. If it helps, place a mental or statement between each character in a character class. Using character classes will give your regular expressions more flexibility, and in most cases, it will make it easier to read. On the next slide, I'll talk about the different types of character classes. In the first example I have listed here, it re-demonstrates the gray example I've used in the previous slides. It is what I'll refer to as a basic character class. It's a basic character class because it doesn't include any additional metacharacters or metasequences, unlike the next example. Character classes allow you to use a special metasequence called a range. In the second example, it demonstrates how to use an alphabetic range to match any letter of the alphabet, A through Z. The use of ranges is extremely useful when trying to match a wide range of possibilities. You can also use ranges for digits, as shown in the next example, Ranges are a metasequence you can use withinside a character class, but it is not a type of character class. There are only two types of character classes, non-negated character classes, referred to simply as character classes, and negated character classes. Every example you've seen so far has been a non-negated character class, or just a normal character class, for that matter, so let's move onto the next two examples and discuss negated character classes. Negated character classes are used to match characters that are not listed withinside the character class. Negated character classes are declared by using the caret symbol immediately after the opening bracket of the character class. The first negated example listed here is And what this expression means is match every character that is not A, B, C, or D. The last negated example is the same thing, but using a character class range, a-d, to accomplish the same thing. Now that you're armed with a better understanding of character classes, let's dive into some demos. In the upcoming demo, you will use character classes to discover specific Active Directory cmdlets, to validate IP addresses, to replace invalid characters within character names, and then you will use character class ranges to validate an IP address range and to discover Active Directory users. The demo finishes up by demonstrating negated character classes. You will use negated character classes to match different date time formats, as well as to split email addresses.

Using Character Classes

I want to start this demo off by walking through the gray example I've previously used throughout the slides. Just to reset the scene, the problem I'm trying to solve is, how do I craft a singular regular expression to match the two possible spellings of the word gray, one with the A, and one with the E? If you look further into the problem, what I'm really wanting Regular Expression to do is allow an A or an E at the third character position. When you put it in those terms, it makes it an easy problem to solve within Regular Expression. The answer to this problem is of course a character class, but pay close attention to the output as I run line five. Did you notice that it didn't return a true or false this time? It simply just returned the results of the match to the console? And if we take a look inside the matches variable... You'll notice that it isn't populated either. The reason for all of this is that I passed in two different strings to the match operator at once. When you do this, it simply runs the expression against the string and outputs only if it's a match. If it isn't a match, it simply just discards it. The next question that you might have is, well, what if you put both the words inside a single string and try to match it? Well, that's a great question, so let's go ahead and run line seven and see what happens when you do that. This time, we get a result that we expect. It returned true, and if we take a look inside the matches variable again, we can see that it did match, well, it only matched to one gray, and to the defense of the match operator, it is match, not matches, so what's going to happen when you use this operator is it's only going to match the first instance that it finds, so if it has several matches in there, you'll have to use a cmdlet called Select-String, so if we scroll down just a little bit to get a closer look at line 11, what I have here is a string that contains both of the spellings of the word gray, separated by a space and the and symbol, and then piping that string to Select-String, and then specifying the regular expression pattern that we've been using previously to match the word gray. Then I'm using the -all parameter to match all matches withinside the string. Then after that, I am piping it again to an alias for ForEach-Object, which is the percent sign, and then using the Matches keyword. This is what's going to return back all the matches that are found, and when I run this, you'll see that it returns back a list of the matches but in the form of an object. So we did get two results back, the gray spelled with an A and the gray spelled with an E. And that concludes the mini-lesson on the match operator and the Select-String cmdlet. We're next going to move onto using character classes to search for Active Directory cmdlets. On lines 15 and 17, I'm using the getCommand cmdlet to search for Active Directory cmdlets that begin with get and set. So, if we run both of these lines, we'll just get a list of cmdlets returned to us. The first one retrieves all of the get commands withinside the module, and the next one retrieves all of the set commands within the module. Now, what line 19 does is it combines the search by using a character class, since get and set both end with et, the only characters that you need inside the character class are an E and an S, as you can see at the end of line 19. And when I run line 19, we'll see that the results now include both the set and get commands from the Active Directory module. But it appears that we didn't just find get and set cmdlets. We also found a reset cmdlet. There's nothing wrong

with the expression, set is inside of reset, so the match is still valid. To fix this, I could use something called an anchor, but I'll save that for a later module in the course, so just to confirm that line 19 gets all the same results that line 17 and 15 do, I'm going to run lines 21 through 23, and what the three lines of code is doing is it's going to get all the set commands, all the get commands, it's going to add them together, and then it will compare it to the results of the character class, so when I run this, it should return a true value, meaning that both of them are equal. The sum of the get and set commands is equal to the character class on line 19. And it does indeed return true, so both of them are equal, and the character class is working as expected. So let's move onto the next example. In the next example in line 26, what we have here is a potential username for an Active Directory user. I am then using the replace operator to replace a number of special characters with nothing, so that way we have a valid username. So this could be a very common scenario, where you're given a listen of usernames from maybe an HR system, and it has some kind of special characters or white spaces in the middle that it interjects. So you could use the replace operator and a character class to replace a range of characters, and that's exactly what I'm doing here. I'm using a character class to look for square brackets. You have to be very careful when you use square brackets inside a character class. You should always put them at the beginning of the character class and then opposing, so it is not confused with a nested character class. The next character that I'm looking for in this character class is an exclamation mark, followed by a forward slash and a backslash. Notice how there are two backslashes here. The backslash is actually a metacharacter withinside Regular Expression, so you have to escape it. But we'll discuss escape characters more in a later module, but just keep that in mind. I could list out the rest of the characters, but I think that you get the point. What this character class is saying, look for any of these characters, and if you find them, replace them with nothing. So if you look at the string on line 26, there is an opening square bracket somewhere in the middle of the name Elizabeth. And then there is an exclamation mark at the end of Walker. So when I run this it will replace both of those characters, leaving us with a usable Active Directory name. So let's go ahead and do that now. And, as you can see from the output, it did replace both of those characters, and now we have a cleaned-up username we could use to potentially create an Active Directory user with. The last example that I have here for you, before we move onto character classes and ranges, is validating an IP address. So starting on line 30, I'm getting the IP address of the server that I'm on, and storing it in a variable called $IPAddress, so go ahead and run that line to populate that variable. Then I'll run line 32 to take a look inside that variable to see what the IP address is. We get an IP address of 192.168.176, so to match that, what we are doing on line 34 is we're saying, match the first three octets, which are 192.168.1, but in the last octet, it's allowing several different variations of characters, and to do that, I am using two character classes. The first character class is going to match the very first digit, and it is allowing a value or a digit from zero through nine, and I have them all listed out, one, two, three, four, five, six, seven, eight, nine, and the same is true

for the second digit in this. So it is only going to match the first two digits in the octet. If you're in any way familiar with networking, you know it can go up to 255, 256 would be the broadcast, but we're only going to match IP addresses that have two digits at the end. So I'll go ahead and run this to see if we get a match, and we do get a match. If we took a look inside the match's variable... We can see that it matched the entire address that we had, 192.168.1.76. That wraps up this first demo on character classes. We'll now move into the second demo in this module where we talk about character classes and ranges withinside those classes.

Simplifying Expressions with Ranges

In this first example, I'm going to use a range within a character class to validate host IP addresses for the network 192.168.20.0/29. If you are not familiar with subnetting, just know that the IP address has to be between 192.168.20.1 through 192.168.20.6. On line six of this demo script, I am populating a variable $IPAddresses with several possible IP addresses. On line eight, I am taking all of those IP addresses that are in that variable, and then I'm running them against the match operator in the expression 192.168.20., and then a character class range of one through six. So this is going to validate that the IP address is within that range. Now, if I didn't use a range for this expression, I would have had to list out all of the digits between one and six to make sure that this character class was accurate. But, that's kind of the benefit of a range, is you can save some space and make it easier to read. So let's go ahead and run those two lines of code to see which IP addresses are valid. And as it turns out, the only two valid IP addresses that were returned were 192.168.20.1, and 192.168.20.3. And both of those are withinside the valid host range, so we're good to go. We had an accurate expression there that kind of filtered through those IP addresses, and told us which ones would work and which ones wouldn't work. I will now move onto the next example where I use a character class to search for Active Directory users. A fairly common username convention is the first character of the first name and then the full last name. What line 11 does is it grabs all the Active Directory users with Get-ADUser -Filter *, and then it filters it with where-object on the user's name with the expression This expression of course contains a character class, but inside of that character class is an alphabetic range of the characters A-Z. Now, because PowerShell is case insensitive, this A-Z really means all upper and lowercase characters, which we'll talk about in just a second. It then takes whatever objects match that expression and then pipes that to Select-Object to filter out only the surname and the given name, so I will go ahead and run line 11, and then we will discuss the results. As you can see from the results, there were only two users that were returned: Brandon Bailey and Grace Bailey. So this expression was successful. It did find all of the users with a last name Bailey and where the first name was any character in the alphabet. Now, I wouldn't imagine any kind of service accounts or anything like that that had any kind of

digits or underscores in it. This expression was designed to only match the users with the last name Bailey and following the normal naming scheme. I'll now move onto the last example I have for this particular demo and talk about case-sensitive ranges, so starting in line 15, I am using a different operator called -cmatch. Now, this is the way that you force PowerShell to become case sensitive, and the expression I'm using on line 15 is a lowercase alphabetic range A-Z that is going to evaluate the first character of the word REGEX. Since REGEX is all capital characters, this expression should fail, but let's go ahead and run line 15 and find out. As you can see, the expression did fail, not a surprise there. We were trying to match an uppercase character with a lowercase expression, while on line 16, we're flipping it around. We are now changing the character class to look for all uppercase characters, so let's go ahead and run that and see what happens there. Changing the range inside the character class to all uppercase characters fixed the issue. Case sensitivity really isn't a big deal in PowerShell, because the engine itself is case insensitive. However, it's a huge deal in other languages like Pearl and Python and even when you're using Grip on a Linux system. And that's why the next expression is so common. The next expression on line 17, I am using two alphabetic ranges, one for the lowercase, and one for the uppercase. So if I add another string in here to have a lowercase r, this should now match both variants of REGEX, and as you can see from the results in the console screen, both REGEX strings got returned, meaning they both matched the expression. That wraps up this demo. Next, you will learn about negated character classes.

Crafting Expressions with Negated Character Classes

As I mentioned in the slides, negated character classes allow you to specify what you don't want to match. And the problem that I'm hoping to solve here in this first example with a negated character class is, how do I match a date when the delimiting character is different? I'm attempting to solve this problem by using both non-negated character classes as well as a few negated character classes. Breaking down the expression used on lines five through nine, I start off with two non-negated character classes. Both of these classes are looking for a digit. The first one will match the zero, and the second one will match the one. Next, I have the first negated character class. You can tell that it's negated by the caret symbol right after the opening square bracket. The other character inside this negated class is a space. Regular Expression will interpret this as match any character that is not a space. The pattern of two non-negated character classes looking for digits and a negated character class looking for anything that isn't a space is repeated to match the section of the date that represents the day. The expression then ends with four non-negated character classes looking for digits. These four classes will match the year portion of the date. On line five, the month, the day, and the year are all separated by a hyphen, and because the hyphen is not a space, the regular expression

that we're using on line five should match. So let's go ahead and run that to find out. So the expression matched, but to find out what exactly was matched, we'll take a look inside the matches variable. And as you can see, it matched the entire date. It matched the year, then a hyphen, followed by the day, followed by a hyphen, and then the year. On line seven, I'm doing the same exact thing. I'm running the same expression, but against a different date format. So in this example, on line seven, I am using a forward slash instead of a hyphen to separate the month, day, and year, so let's go ahead and run and see if the expression validates against that set. And it does. We get the results of true, and if we look inside the matches variable, we should see the entire date. And, not too surprisingly, it did match the entire date. Now, just to validate that this expression is working, the only character that it shouldn't allow is a space, so on line nine, I've replaced the forward slash with spaces, and when we run this, we get the result of false, so it didn't match because of that space in between those dates. Now, I must confess, this isn't the best expression for this type of work, because it can allow any type of character to exist in between these different dates. It can be an at symbol or a plus sign, really, it could be anything, just, it can't be a space. One of the things you will learn while working with Regular Expression is the uglier the data, the more thought you have to put into crafting the expression used to match that data. So below, on line 13, I have a different expression that doesn't use a negated character class, but it's a little bit more accurate when working with dates, because dates are usually separated by a hyphen, a period, or a forward slash. One thing to notice with this particular expression is the hyphen inside the character class that matches the delimiting character is first. If the hyphen was placed in between the period and the forward slash, it would have been interpreted as a range, and that wouldn't have been valid. So, enough about dates. I will now move onto the next example, where I demonstrate splitting up an email address with Regular Expression. On line 17, I am taking the email address ewalker@globalmantics.com, and then I'm splitting it at the at symbol, and then I'm taking that result, and I'm splitting it again at the period, because I want to split it at each of those sections so I can grab out the domain name and the username. However, there's a better way to do that, which I've demonstrated on line 19, where we could use some more advanced Regular Expression and take advantage of using a negated character class to handle all of those non-alphabetic characters. Taking a closer look at the expression used on line 19, I'm using a negated character class to split at every character that is not A-Z. So in this example, it will split at the at symbol, and at the period in between globalmantics and .com. And to prove that both line 17 and 19 do the exact same thing, I will go ahead run each of those individually. So here are the results from line 17, where it split at the at symbol and at the period between .com or globalmantics and .com. And then we should get the same thing when we run line 19. Again, it splits at the at symbol, and then at the period between globalmantics and .com. That wraps up the rest of this demo. Let's now head over to the summary slide and review everything that was covered throughout all of these demos and this module. Throughout this module, you learned

about character classes and how they can be used to create more dynamic expressions. You also learned about character class ranges and how you could specify a range of characters to be used to match within an expression. You learned about negated character classes and negated character class ranges. In addition to that, you were introduced to two new ways of using Regular Expression and PowerShell, the Select-String cmdlet, and the -split operator. Character classes are an important part of your Regular Expression knowledge, and you'll use them again throughout this course. In the next module, you'll expand your knowledge of Regular Expression even further, and learn about alternations.

Crafting Regular Expressions with Alternations

Introduction to Alternations

Welcome to Crafting Regular Expressions with Alternations. Previously, you learned how to use character classes to match one character out of several possible characters. Alternations are very similar, except for they are used to match a single regular expression out of several possible regular expressions. So what do alternations do? I just mentioned that they are used to match a single regular expression out of several possible expressions, but does that really mean? Well, let's say that you wanted to match a domain name that could end in either .com, .net, or .gov. You wouldn't be able to use a single character class to match all these different domain suffixes, because they are just too different, but you could do it with an alternation, and here's what that would look like. Each of them begins with a period, so you would type that first, and then, you would list out every domain suffix that you wanted to match. Com, net, and gov. And then separate them with the pipe symbol. This expression is very similar to what you've seen before with a character class, except it allows you to define several possible expressions, not just characters. Now that you have an idea of what they are used for, let's move into the next slide and define alternations a bit better. Alternations are represented by a vertical bar, also known as the pipe symbol. They are used to match a single expression out of several possible expressions. In the top example, I have two totally different words. Right away, you can tell that you cannot use a character class to match both of them within a single expression. However, I can use an alternation, which allows me to match both of these expressions within a single regular expression. I simply type both of them out, learning and regex, then separate them with the pipe symbol, and now I can match either of these words within a single expression. It's easy to forget that learning and regex aren't words, they are actually expressions. Alternations give you the ability to match a single expression out of several possible explanations. So on the bottom line, I'm looking for all the get set and add commands, but on the left side of the alternation, I'm using a character class to combine

searching for get and set. Let's move on to the demo slide, and I'll discuss what we will be covering in the upcoming demo.

Using Alternations Within Regular Expression

In the upcoming demo, I will be using alternations to match several words within a string. I will then use alternations to discover specific PowerShell commandlets using the verb get, set, and add. After that, I will use alternations to match PowerShell file extensions in conjunction with character classes. Lastly, I will use alternations to match Windows event logs. I will use regular expression to filter through created user events and deleted user events. To start off this demo, I will be using an alternation to match two words within the string. Learning regex is awesome. Some might disagree with that statement, but hopefully you don't. I remember when I first started learning regular expression. Every time I would figure out something, I kind of felt like a wizard. Anyway, back to the example on line five. To match the two words within the string, I'm using the select-string commandlet with the expression Learning|Regex. The alternation is indicated by the pipe symbol, and this allows me to match both of the words within a single expression. Now, this would also match either of the words, but since I have both words listed in the string, it will match both of them. I am also using the -all parameter to find all the matches, and then, I am piping it to the for each object and returning all of the matches to the console with the match's keyword. So, let's run line five and take a look at the results. I will scroll up just a little bit so we can see both of the results. The first match was learning, so that captured, and the second capture is regex. So, that means this expression was 100% successful in capturing and extracting out both the word learning and regex. Moving on to line nine, we will discuss using a character class inside an alternation to find get, set, and add commandlets. Staring on line nine, I am using Get-Command to retrieve a list of all of the available commandlets on the system. I am then using the match operator with the expression opening square bracket, G-S, closing square bracket, which is a character class. Followed by E-T, hyphen, then an alternation, which is the pipe symbol A-D-D hyphen. The left side of this alternation expression is using a character class to look for both the get and set commandlets. You've seen this expression before, so I won't go too much in detail, but we know that it's going to be looking for those two different verbs inside of the commandlet name. Then, on the right side of the alternation, is just looking for a literal character match of A-D-D hypen, which will, of course, match all of the commandlets with the verb add, and when I run this, I will get a very long list of commandlets. Scrolling through the output, we see several set commandlets, those are to be expected. Just above that, though, we see some resets. These snuck in there again. That's because the word set in inside of reset. I will show you how to fix this with some anchors in an upcoming module. That way, they don't get returned. Only the set commandlets get

returned. Just above that, though, are some get commandlets. Those are good, we expected to see those, and way at the top, we'll see all of the add verb commandlets. Before we move on to the next example, let's go ahead and clear the screen, and then, we'll scroll down just a little bit, and we'll find the next example, where I will demonstrate using regular expression to find all files with the PowerShell extension. So, before I run line 13, let me change into a directory that I know has some PowerShell scripts, and then we'll talk about what's going on on line 13. So, I'm using Get-ChildItem to retrieve all of the items within the current directory, and then, I'm using the match operator with the expression ps1 pipe, which is the alternation, ps1, opening square bracket, M-D, closing square bracket for a character class, and then the number one. If you're not familiar with PowerShell extensions, they can end in either a .ps1, .psm1, or .psd1, and that's what this expression was designed to match. Before I run line 13, let's take look inside the directory to make sure I don't just have PowerShell files. So, I have a .bat file, a .vbs file, a few PowerShell files, and even a Python and a Sequel file. So, now I can run line 13, and you can see that I filtered out all the PowerShell file extensions. We have a .psd1, a .psm1, and a .ps1 file. In the last example, I will be searching the Windows Event Logs for created user events and deleted user events. I will be using an alternation to look for two specific usernames within the message of the event. But before I can do that, I have to first generate these events. Starting on line 17, I'm clearing the security event logs. The only reason I'm doing this is to make my queries run faster. Cleaning the logs prevents me from having to search a lot of logs, which would take a good chunk of time. So, I'll go ahead and do that now. On line 19, I'm populating a variable called $Users with several new usernames. Then, on lines 21 through 24, I'm going through a foreach loop, which will go through each of these users and $Users. And then, it will create the new user account and then delete it. Doing this will create two events per user. One for creating the user, and then, one for deleting the user. I'll just highlight these few lines of code, and then, hit Run Section, and you can see that the verbose which was turned on so we can see these user accounts being created and removed. Now, if we scroll down just a little bit further, we'll see two examples of doing the same exact thing. So, they are both going to retrieve deleted user events and created user events, where the message contains mhall and alewis. One of them is a little bit shorter because it is using an alternation instead of wildcards. The wildcard version is on lines 28 and 29. It starts by using the Get-EventLog commandlet, and then is using the InstanceId parameter to search for 4-7-2-0 and 4-7-2-6. These are the two event IDs that are used for created user events and deleted user events. I am then using the LogName of security, because they're inside the security logs, and then I'm piping that to where object. If you haven't seen this where object syntax before, this is the syntax that allows you to find multiple statements. Within the two curly braces, I have two statements. One is looking for a message that is like mhall, and the second is looking for a message that is like alewis. I've placed a logical or operator in-between these two statements. So, it will find either of the two users within the message of the event. I will now go ahead and run lines 28 and 29, so we can

take a look at the results. As you can see from the output, we have four results returned to us, and this is accurate. We should have one event for the user being created and one event for the user being deleted, and since we're only searching for two of the three users that we created and deleted, only four records should be returned. Now, you can't see inside the message, and we'll take a look at that later, but let's go on to line 33, and take a look at using an alternation to do the same thing. Line 33 starts the same way with Get-EventLog, it's using the same two InstaceIds for created user events and deleted user events, and it is also using the security log. It is then piping the results to where object. This time, it is doing a match on the message, where it matches mhall or alewis using an alternation instead of the like operator, and when I run this, we should get the same results as lines 28 and 29. So, let's find out if we do. And just from a quick glance, it looks like we do get the same results. If we take a look at the indexes, those should line up. So, index 43, 42, 37, 36. Yep, those are all still the same. Now, if we wanted to look inside the message, we can go down a little bit further, and I'm just using the same thing, but I am selecting out then expanding the message of this event. So, we'll go ahead and run that and take a look at the message. Now, this is the entire message for both of those events. Well, actually, all four of those events for mhall and alewis. If we scroll up through this, we should see their usernames in here. Mhall, as we see right there. It gives us some additional information. Everything that's included in the message will be here, so their security ID and so forth. Scroll up just a little bit further until we find an alewis example. So, both of these, all four of these records did get returned, and we can take a look inside the messages now with ExpandProperty. It's worth mentioning at this point that most of the things that you work with on Windows are objects, but in certain cases, such as this message, is just a string, and when you are dealing with just a string, it becomes harder to parse without knowing a little bit of regular expression. We'll take a look at parsing this message a little bit more later in the course. But for now, let's head over to the summary slide, and review what you've learned about alternations. At the beginning of this module, you learned what alternations were. You learned that they are used to match one of several possible expressions, and that it also gave you the ability to use several expressions within a single regular expression. You then learned how to use alternations by matching PowerShell file extensions and for searching Active Directory user events within the event log. In the next module, you will learn how to use shorthand characters to shorten your regular expressions.

Using Shorthand Metacharacters to Shorten Regular Expressions

Introduction to Shorthand Metacharacters

Welcome to "Using Shorthand Metacharacters "to Shorten Regular Expressions." Shorthand metacharacter shorten regular expressions by using a single metacharacter to represent several

characters. They can be used to represent a wide variety of things, such as whitespaces, which includes tabs, spaces, newlines, and so forth. Other shorthand metacharacters are used to represent digits. There is even a shorthand metacharacter to represent every possible character. In my opinion it's a bit overused, but it does come in handy. On the next slide you'll see a table that contains several of these shorthand metacharacters. This table consists of several shorthand metacharacters and their purpose, it isn't a conclusive list, but the ones listed here are very common, and are used across many different flavors of regular expression. The first row in this table starts off with the metacharacter \w, which is used to represent a word character. You might at first think that this only means alphabetical characters, and that's a good guess, but it's not quite right. In regular expression a word character is anything A through Z, capital or lowercase, a digit zero through nine, and/or an underscore. At first this struck me as a little bit odd, but the reason is, because all of these characters are valid in a variable name, it was decided, way back when, that a word character ought to include all of these characters. Another shorthand metacharacter, \d, is a little bit more straightforward. This metacharacter is used to only represent digits zero through nine. Notice that both of these metacharacters that I just talked about, \w and \d, have an opposite with a capital letter. So you have \W, which means anything that is not a word character, which is A through Z, zero through nine, and an underscore, and then you also have the same thing for the digit character, with the \D, which means anything that isn't a digit. The other two shorthand metacharacters worth mentioning before we move on are the \S and the period. \S is used to represent any whitespace, this includes any spaces, tabs, new lines, et cetera. \S is an extremely useful shorthand metacharacter, and you'll learn more about it in the upcoming demo. The period is a very special metacharacter that acts almost like a wild-card, because it can match any character. Now that you have an idea of what shorthand metacharacters do, let's move on to the next slide and see a few examples. Shorthand metacharacters are used to represent more than one character, or a range of characters, as you saw on the previous slide. The use of shorthand metacharacters will greatly reduce the complexity and the length of your regular expressions. In the first example I am using the expression \w, repeated four times, followed by the letters "a-c-c-o-u-n-t". \w is looking for word characters A through Z, zero through nine, and an underscore, which allows me to match the service account name. If I had used a character class here instead, I would have had to type out 28 additional characters. The next example, I am using the \d shorthand metacharacter to match all the digits in a phone number. Using shorthand metacharacters here instead of a character class saved me 30 additional characters. I'll be able to shorten this even more with a quantifier later on in the course, but for now, I think you get the point. Using shorthand metacharacters saves you a lot of typing. I will now move on to the demo slide so you can learn to use these shorthand metacharacters in your expressions.

Crafting Expressions with \w \d and . Shorthand Metacharacters

In the upcoming demo I will be demonstrating how to use shorthand metacharacters to match any character with the period, and how to match any word character with \w. After that we'll move on to looking for active directory users. We'll talk about the differences between a character class, and then using the shorthand metacharacters, followed by validating an IP address. We will use the \d character to match the digits of an IP address. After that we'll walk through several examples of using the \s metacharacter to replace whitespaces. Lastly, you will learn how to convert an array to a string by using the \n metacharacter to replace new lines. Starting on line five I am populating a variable called "$matchAnything". It will contain an array that has the letter A, the number one, an exclamation mark, and a space in it. On line seven I'm running every item in the array against the shorthand metacharacter period. The purpose of this is to show you that the period will match every single one of these items in the array. I will go ahead and run both of those lines now, and as you can see from the output, it did match every single one of those items in the array, the A, the number one, the exclamation mark, and the space. You can't quite see the space here, but it does exist. It did match the space with the period metacharacter. Scrolling down just a little bit, I will now demonstrate the same thing, but by using the \w shorthand metacharacter. Now if you recall from the slides, the word character should match letters A through Z, capital and lowercase, as well as numbers and the underscore. So when I run this, it shouldn't match against the space that I have in the array, and it shouldn't match the dollar sign. I'll go ahead and run these two lines, and we'll see if I'm right. So as you can see from the output, it matched the A, the B, and the number one, and as I predicted, it did not match the space or the dollar sign. The next example that I have listed on line 16 is a little bit more practical. Line 16 starts off by using "Get-ADUser -Filter *", and then pipes it to the where-Object commandlet. It is then matching the name against the expression \w, repeated four times, and then the word "account". A fairly common naming convention with an active directory is to prefix some of the account names with "svc" or "adm", and then an underscore, so using the \w will allow me to match all of those different types of accounts. So I'll go ahead and run this now. As I scroll through the output, you'll notice that I found two different accounts. I found this defaults user's account, and I also found this svc_account, which is a service account for the Globomantics domain. The intent behind this expression was to find accounts that were prefixed with three characters, followed by an underscore, or any other word character. By this definition, "DefaultAccount" shouldn't be in the output, the reason it is however, is that it does contain four word characters, and the word "account". To avoid this, I should use something called an "inker", which will be demonstrated in an upcoming module. I'll now move on to the next example where I'm using the \d shorthand metacharacter to match an IP address. On line 20 I am using the "Get-NetIPAddress" commandlet to get the IP version four address from the ethernet adapter of the server, and then I am storing it in a variable called "$IPAddress", and then on line 22 I'm taking that IP address, and then I'm running it

against an expression that uses the \d shorthand metacharacter to validate all of the digits inside the IP address. I've separated each of the octets in the IP address with a period, as you would see in a typical IP address, and when I run line 22, we do get a result of True, and if we take a look inside the matches variable, I should see the entire IP address, 192.168.1.76. Now there is one small problem with this expression. Remember that I said that the period is actually a shorthand metacharacter that will match anything, so on line 24 what I'm doing is I'm replacing the period with a dollar sign, what's inside the IP address, and I'm running it against the same expression. And when I do this, we'll see that it returns True as well, and if we look inside the matches variable again, we'll see that it did match the IP address, but now it's separated with a dollar sign. Now this is because the period is a metacharacter to match anything, and when we're using it as a literal character, we need to escape it, and if you take a look at line 27, that's exactly what I do. To escape metacharacters in regular expression, any metacharacter, not just a period, you just put a backslash in front of it, and if we run line 27 with the escaped period, it should not match the dollar sign example with the IP address. So we'll run line 27 now, and we'll notice that the result is false, because we should not be allowing IP addresses to match if their delimiting character between octets is a dollar sign. That wraps up this demo, in the next one we'll take a look at how to use the \s metacharacter to replace new lines and spaces with inside a string, and then we'll take a look at converting an array to a string.

Splitting on Whitespaces and Replacing Newlines with \s and \n

The next metacharacter that I will show you is the \s metacharacter. If you recall from the slides, this is the metacharacter used to represent any whitespace character. Lines three and four populate a variable called "$text" with the sentence, "This text contains whitespaces and a newline." I'll run these two lines to populate that variable. Now on line eight I'm attempting to use the Split method to split on a space character. What I expect from this is one word per line, however, as you can see, that isn't the case. The reason for this is not all whitespaces are equal. For example, a newline is different from a space, and a return is different from a newline, and a newline is different from a tab. The other reason is, the split method can't handle more than one space. I would have to specify, either split on a single space, or a double space, it doesn't have the flexibility to say, "do both, "split on a single, or a double space," and that is where regular expression comes in. If I scroll down just a little bit, we'll see another example where I'm using the Split operator and then some regular expressions to split the text variable. On line 12 I am using the expression \s, and then the pipe symbol for an alternation, and then two \ses, so this is going to look for a single space or a double space. Now when I run this we should get, hopefully, a single word on each line. Sadly however, we still have some blank lines between

the words, but why? The expression that I ran on line 12 should have solved the double space problem, because I'm using an alternation to say, "Split on either a single space, or a double space." Now this is an important lesson to learn when using alternations, and that is, the order of which you put them matters a great deal. The reason this didn't work was because the expression was looking for a single whitespace character first. This is a problem because every time it ran into a place where there were two spaces, two whitespaces, one was left over, which is why you see a few blank lines in the output. I can fix this by simply just switching the positions inside the alternation, as you can see on line 16. The expression on line 16 will now favor two whitespaces over a single whitespace, so it will split on two whitespaces first, and if it isn't double whitespaces, it'll split on a single whitespace. And now when I run line 16 we can see that there is a single word per line, so we've effectively fixed that issue, and now we have each word of the sentence on its own line. The very last example that I have for you in this module is how to convert an array to a string. Before I show you how to do that, let me set up the problem that I'm trying to solve here. So what I have is a list of roles from Wmi, that I want to get and put into a custom object. So these are all the roles on this server, it is an LM_Workstation, an LM_Server, a Primary_Domain_Controller, and it's a Timesource, and it looks like there's something else, but you can't quite see that with the dots. So again, my overall goal here is to take all of these roles, and then put them into an object that has the server name, and then the roles listed out, seperated by a comma. So to do that, I'm going to grab all of those roles, and I'm going to put them in a variable called "rolesArray". Now if I wanted to, I can look inside that array to find out what all of those different roles are. This is a little bit better than before, at least we can see all the roles, there's not that dot, those three dots there, hiding some of them. But now I want to convert this to a string, and then put it into an array so I could possibly maybe export it as a report, or something like that. So on line 24 I am creating a variable called "$roles", this will be a string, and then I am taking the rolesArray, piping that to Out-string, which just converts it to a text string, and then I'm using the replace operator and the shorthand metacharacter \n to match all the newlines, and then replace those with a comma. Now if I take a look inside this roles variable, we should see all of them separated by a comma. Well there's one extra comma at the end, and we can quickly take care of that. As you can see on line 26, we're just using the Trim method to remove that last comma. Go ahead and take a look inside again, we shouldn't see that. So now we have what we wanted, we have all of the roles separated by a comma, and now to complete our project, we will put them inside a PS custom object where we have a computer name and a roles property, and then we'll populate that with the environment variable for the computer name, and our new string "roles", and when I take a look inside this new object, $obj, we now see the computer name property and the roles property. Let's now head over to the summary slide and review what you've learned. In this module you learned about shorthand metacharacters. You learned that they were used to represent many different characters, with just a few characters. You

then walked through several examples of using shorthand metacharacters. You use them to discover active directory users, to validate an IP address, to match and split on spaces, and to convert an array to a string. You also learned the escaping metacharacter. You learned that the \ can escape a special meaning of metacharacters, to convert them to their literal form.

Matching Start of Line and End of Line Positions with Anchors

Introduction to Anchors

Throughout this course you've learned several ways to match characters in regular expression. However, it is possible to also match positions. In this module, you will learn how to use a special meta-character called an anchor to match the start of a line and the end of a line. Before I get into the long winded, technical description of what start of line and end of line anchors are, I will first talk about when to use an anchor. In a previous module, I use an expression with a character class to get all the get and set cmdlets from the active directory module. There was, however, a slight flaw with this expression. As you can see from this snippet, the expression also returned a cmdlet with the verb reset. The reason this cmdlet was returned in the results was because set is found with inside of reset. To avoid this, I would need to get more specific with my expression and say, only match things that start with set. To accomplish that in regular expression, I would use something called an anchor. Which we'll see in the next slide. Anchors are used to match positions, instead of characters, in regular expression. By doing this, it effectively anchors the regular expression to match a specific position, hence the word anchor. In this module, I will be covering two different types of anchors, a start of line anchor and an end of line anchor. The first example I have shown here demonstrates how I can use a start of line anchor to solve the problem I had when trying to match only cmdlets with the verb set. By using a start of line anchor, which is represented by the caret symbol, I can prevent reset from matching. The reason it no longer matches is because the expression is now looking for a start of line position, followed immediately by an S, then an E, and then finally a T, and because of the first letter in the word reset being an R, it will no longer match due to the start of line anchor being used. In the next example, I have two words, script and scripts. I am then using an expression that is looking for the word script, followed by an end of line anchor which is represented by the dollar sign. If I remove the end of line anchor from this expression, script would match both the word script and scripts. The word scripts would match all the way up until the end where it found the s. The end of line anchor is required when I only want to match the word script. When thinking about anchors, it helps to translate them literally. Think of them like this, first match the end of the line position, which is the dollar sign, followed immediately by t-p-i-r-c-s. It works just the same way as the start of line anchor except for, you're of course anchoring the end of the line, not the start of the line.

Now that you have some idea how to use start of line and end of line anchors, let's move onto the demo, where we can see these being used in action.

Using Start of Line Anchors

In the upcoming two demos, we will start by using start of line anchors. We will use these anchors to match the start of the line positions. I'll then use them to discover active directory cmdlets with the character class being anchored, as well as how to find active directory users with shorthand meta-characters in conjunction with anchors. After that we'll move on to another demo where I demonstrate using end of line anchors. We'll use them to anchor the expressions to the end of the line, as well as how to replace trailing whitespaces. We'll then walk through finding specific windows features with end of line anchors, and then finally wrap up the demo by matching specific events and take a look at how you can use a switch statement with regular expression. I'll start this demo off by first walking through the set/reset example I talked about in the slides. To simplify it, I have just the two words set and reset specified as a string and then I will run them against the expression set and when I run line five, you'll notice the problem immediately. This expression matched both the set and reset strings. There isn't anything wrong with this expression, it's doing exactly what I told it. I told it to look for an s, followed by an e, and then a t, and it did exactly that. The real problem is I wasn't specific enough. What I really want is the expression to match s-e-t only if it's at the beginning of the line. To do that I can use a start of line anchor, which you see on line nine. The anchor is represented by the caret symbol. It is important to know that regular expression has a few different meanings for characters. In a previous module, I used the caret symbol to declare a negated character class, but because this caret is outside of square brackets, the meaning is different. It no longer means a negated character class, it means an anchor and specifically a start of line anchor. Now when I run line nine, we will see that reset is no longer returned from the results because we are using that anchor to match the very first position of the new line, immediately following an s, an e, and then a t. Because of that, reset no longer matches because it starts with an r, not an s. With that somewhat academic example out of the way, let's scroll down just a little bit and take a look at some practical examples of using anchors. On line 13, you see a somewhat familiar example of using get command to retrieve all the active directory cmdlets and that I'm piping that to where object and then filtering the name on the expression caret, opening square bracket, GS, closing square bracket, et. This expression should look familiar to you as well, this is the one that we used to get all the get and set commands. It did have that problem in the expression, where it did return some of the reset cmdlets, but now when we run this you can see that there are no more reset commands returned in the result set. Scroll all the way through this. Only get in set commands are returned. Now that

fixed the issue, but if you look closer at the expression, I'm anchoring the entire character class which means g or s. If that isn't what you want, you can use an alternation and then say, look for either get, g-e-t, or a new line character followed immediately by s-e-t, as you can see on line 17. When I run line 17, you'll notice that I get the same results and that's because there aren't any cmdlets that have the word get inside of it. In the next example I am using a start of line anchor in conjunction with the \w shorthand meta-character. \w is used to represent word characters. When I ran this in a previous module, I didn't get exactly what I was looking for and to show you what I mean I'll go ahead and copy this line and I'll clear the screen, paste it into the command prompt, and then remove the anchor so that we can see the results. As I scroll through these results, you'll see that two user accounts were returned. We have this default account which matched and we have this service account. So svc_ is what I was trying to match against. The problem is there are four word characters with inside of default account, so again, the fix to this is to use an anchor and say there should be four word characters between the start of line and the word account, but no more than that. That's how I am fixing it on line 21. So now, when I run line 21, you'll notice that only one account is returned. That wraps up the first demo on start of line anchors. In the next demo we will walk through using end of line anchors.

Using End of Line Anchors

To start this demo off, I have the script, scripts example that you saw in the slides. As you can see on line five, I have two words, script and scripts, and then I'm running it against the expression script and when I do that, I get both returned to me. The reason is fairly obvious, script exists with inside of scripts. But we can't use a new line anchor here because that would be at the very beginning of the expression. We now need to use an end of line anchor, which you see on line nine. The end of line anchor is represented by the dollar sign and when I run line nine, you'll notice that only script gets returned to me. The reason scripts no longer matches is because we're saying match the end of the line position and then right before that needs to be a t. If it is not a t, it does not pass this expression. Well, technically, it says end of line position tepi, and so forth, but you get the point. Scrolling down just a little bit, in the next example we'll take a look at replacing whitespaces. On line 13, I have a sentence inside of a string that says has a whitespace at the end that I'm attempting to use the replace operator in the expression \s, to replace all the whitespaces. Really what I want to do here is replace the whitespaces at the end, but as you see when I run this, it takes them all out and it kind of just kluges them together. So we need to use an anchor to match only the whitespaces at the end of this line. That's what I'm doing on line 15. On line 15, I am using the same expression, except for I'm putting the dollar sign, which is the end of line anchor, at the end. Now this is only going

to replace a single whitespace and later on we'll talk about quantifiers, which allow you to repeat the same meta-character a specific number of times or any number of times. When I run line 15, you'll notice that there are spaces in between these characters now and we only removed the space at the end of the sentence. Moving onto the next example, on line 19, I'm using the get windows feature and then I'm piping that to where object and filtering to name where it matches two different expressions. I'm using an alternation here to match tools and console, however it has to match at the end of the name. This will look for all the features that end with tools or console. Might come in handy when you're looking for tools to install with windows feature. You can see here, a few have already been installed, such as the remote administrator tool kit, which is rsat forshort. If we scroll up there's several other ones for IIS and for shielded Vms, which sounds kind of cool. Alright, time for the final example in this demo, where we're going to use regular expression to match specific event IDs. We're going to use a switch statement to do that. So you can see the code better, I'm going to clear the screen and maximize the script panel. On line 23, I'm populating a variable called deller Ids with several random event IDs. Then on line 25, I enter a for each loop and within this for each loop, each ID will be processed by a switch statement. This switch statement is using regular expression. The ID passed to the switch statement is tried against three different expressions. The first one uses an end of line anchor to match numbers that end with 65. If that is a match, the variable $messagedata, equals end of line match found and then the variable ID. In the next expression, I'm trying to match all the numbers that begin with 65. To do that I am using a start of line anchor, which is of course represented by the caret symbol. When that match is found, the $messagedata equals start of line match found and then the ID variable. The very last expression used in the switch statement is using both a start of line anchor and and an end of line anchor, which is effectively looking for only the number 65 and when that match is found, the $messagedata is equal to start and end of line anchor match found and then it has the $ID in there as well. Right after the switch statement, but still inside the for each loop, I am creating a hash table called $splat. With inside that hash table I'm defining several key property values. The source identifier, which is set to equal intro to regex. The message data is going to equal the variable defined within the switch statement. Then the sender is going to be windows.timer. After that hash table has been created, I will then use it to pass all of the parameters and the values of those parameters, to the new event cmdlet. After that, I am piping it to out nulls, we won't see any of the results from it. Then the for each loop closes. I will go ahead and run lines 23 through 40 to generate those events. There we go. Because I had the script panel maximized when I ran that, it automatically just maximized the command prompt. You can hit control r to minimize that again. Now we have the events created and we want to retrieve them and make sure that our regular expressions matched. To retrieve or to get those events, I will run line 42. The very last event returned was the one that matched 65, so if you look at the message data, it does say start and end of line match found and then it has the ID

that it was matched against, which was 65. If you move up to the next example, with the event identifier of two, you can see that the message data contains start of line match found, which matched the event of 6587. As you can see there, 65 was at the beginning of that event ID and when I scroll up just a little bit, the very first event that was created was the end of line match. If you looked at the message data, it says end of line match found 5565. At the very end of that ID it did find 65. The events I created for this demo are kind of random and they don't serve a very good purpose, but it gives you the tools necessary to build some really powerful switch statements using regular expression to do any number of things. It could be to validate an IP within certain ranges and then do certain actions on that or it could be to define different active directory roles or even server roles. Then you could do different actions based on that. With that said, that is the end of this demo and the end of this module. In the next module, we'll talk about word boundaries which basically combine the previous two modules, where we talked about character classes and we talked about anchors and the one thing that you can use to match words.

Creating Word Boundaries Within Regular Expression

Introduction to Word Boundaries

Throughout this course, I've run into the same problem a few times, where the expression I used matched more than I intended it to. I was able to solve a few of those problems with using start-of-line and end-of-line anchors, but if what I was matching wasn't at the start of the line or at the end of the line, these anchors wouldn't have helped at all. This is where word boundaries come in. Word boundaries are another metacharacter in regular expression that matches by position rather than by character. Before I define and talk about what word boundaries are, let's first discuss what problems we can use word boundaries to solve. Let's say that you had a web.config file with two different connection strings. These connection strings contain the name of two different databases. You need to update the connection stream that is currently using the database name of Customers to Customer. By only using the techniques that you've learned so far in this course, you wouldn't be able to match the connection string with just the database name of Customers. The reason is because Customers is found in both connection strings. One contains the database name of Customers, while the other contains the database name of CustomersOrders. This is a perfect example of when to use a word boundary when your expression is matching too much. On the next few slides, I will show you a few examples of using word boundaries as well as walk you through how it matches by position. Word boundaries are commonly used to avoid matching a word within another word. It does this by matching the position before a word character, and then matching the position right before a non-word character on the other side. These two position matches, combined

together, form the word boundary. In the first example that I have shown here, I have several words, scatty, scatter, and cat. Each of these words contain the word cat. If I were to simply use the expression C-A-T, it would match all three. I can't use a start-of-line or end-of-line anchor because cat is in the middle of these two words, and it is not at the beginning or at the end. I can, however, insure only cat is matched by using a word boundary. In .NET and in PowerShell, the metacharacter for a word boundary is the backslash b. You have to place the backslash b at the start and the end of the expression you are looking to create a word boundary around. In the next example, I am looking for the word list. However, list is found with inside listen. Using the word boundary, I can prevent listen from matching. You could, however, in this case, use a start-of-line anchor here, as well. Much like everything, there is always more than one way to do something. On the next slide, I will discuss the positions a word boundary is allowed to match. There are only a few positions that qualify as a word boundary position, the first of which is the position before a character in a string, only if the first character is a word character. If you need a review of what word characters are, please reference the slides from the shorthand metacharacter module. The second position that word boundary positions can match is right after the last character in a string. These two positions are the same as the start-of-line and end-of-line anchors. That's why the word boundaries and the start-of-line and end-of-line anchors have some overlapping functionality. The last position is different. Word boundaries can also match between two characters in a string, where one is a word character and the other is not. The position matched in this example is actually right between the d and the question mark. The question mark is a non-word character, and the d is obviously a word character, so that is where it matched. It matched the position, remember, not an actual character. Now that you have an understanding of what word boundaries are and how they can be used to match positions, let's move on to some demos.

Crafting Expressions That Use Word Boundaries

In the upcoming demo, I will be discussing word boundaries. I will start off by discussing matching a specific word within a string. This will demonstrate the similarities and differences between word boundaries and line anchors. I will then use word boundaries to audit folder positions for a file share. After that, I will verify a database name with inside a web.config file by using regular expression and word boundaries to validate the database name. Starting on line five, I have a very similar example to what was shown in the slides. I have a few words here. I have cat, scatty, cat, and scatter, and this is to prove out that you can use a starting line position to find at least one of the instances of the word cat. So, when I run this, it's only going to return one of the two instances it should return for cat, and the reason for that is, the start-of-the-line anchor can only match the first cat, and in the next example on line nine, I move the

word cat from in between scatty and scatter to the end of the string. So now, when I run this, I will get cat returned just one time. That's because it matched the end-of-the-line cat and not the cat at the beginning of the line, because we are using an end-of-line anchor. Now what I'm really trying to accomplish with this expression is to match all of the words cat, without matching words that have the word cat inside of them, so this is the perfect example of when you should use a word boundary, and if I scroll down just a little bit, you'll see the example where I use a word boundary to find all of the instances of the word cat. So this string is a little bit different. I have cat at the very beginning, I have it somewhere in the middle, and then I have it at the end. Now, keep in mind that word boundaries can be used very similarly to start-of-the-line anchors and end-of-the-line anchors. The difference is, it's also able to find matches in the middle of the string. So, when I run this, I should get cat returned three times, and, as you can see from the output, I did get cat returned three different times, and that's because it matched the cat at the beginning, the middle, and the end. Now, if you take a look at the expression, I updated the start-of-the-line and end-of-line characters with a backslash b. Notice, though, that this has to be at the beginning and the end of whatever you're trying to put a word boundary around. So, in this instance, it's around the word cat, or the expression C-A-T. I will now move on to the next example, where I will use regular expression to search the file permissions on a share. Before I do that, I have a helper function here, called New-SalesShare, and this is just a helper function that I wrote that's going to set up a few folders for us and then set the permissions on them, based on some active directory groups. So, that is included in the course demo file, so if you're following along, please take a look at that, and then make sure that you .source that in to be able to run this function. I will go ahead and run that function now to set up all those folders for us, and then I will change into that location at the root of the share, and then I'll just do a Get-ChildItem so you can see the folder structure. So I have several folders underneath this Share folder. One is Sales, one's Sales_Engineers, another is Sales_Managers, and then, underneath Sales, I have EU_ and US_Sales. These are the directories we will be using regular expression to take a look at the ACLs and to audit these share permissions. So, if I scroll down just a little bit, we'll see the first example that I have that's going to be using a wildcard. I'm using this to demonstrate that, currently, with the knowledge that we have throughout this course, we can't get exactly what we're looking for. So, taking a closer look at this block of code, I'm starting by using Get-ChildItem from the current directory, and then I'm using the recursive parameter to get all of the child items, then I'm piping that to Get-Acl. Get-Acl is the commandlet that you can use to get the access control lists from these objects. I then pipe the output from the Get-Acl to select object, and I'm selecting the PSPath and then I'm expanding the access property to look inside. From there, I'm piping it to where object, and then finding all the instances that have the identify reference like sales. So, in that instance, I'm using some wildcards. I then pipe that to select object again, and then I am selecting the IdentityReference and the PSPath, but I'm using this neat little trick

where I'm renaming the PSPath property to just Path, and then I'm changing the expression or what it's going to represent by using this sub-expression to take the PSItem.PSPath and then use a regular expression to replace all of the unwanted text that I don't want to see in the path. Within that expression on line 30, I'm using two metacharacters you haven't yet learned in this course: one is a quantifier, and the other is a sub-expression or a capture group. We will talk about those in the upcoming modules. I will go ahead and run lines 27 through 30, so we can take a look at the output, and what you see here are all the IdentityReferences that have the word sales somewhere in them. The very first row in this output is an active directory group, GLOBALMANTICS Sales. It has permissions to the path C share sales. The next is SalesEngineers, and it has access to a folder called Sales_Engineers. So, the same is true for SalesManagers, but the active directory group Sales has three folders it has permissions to. It has Sales, and then it has two subfolders underneath that, EU_Sales and US_Sales. Now, let's say you take this report here to your audit team and you say, "Hey, here are all the directories "that Sales has access to." Now, typically, what will happen is, they will return back to you and say, "Hey, we really only care about Sales. "Can you remove SalesEngineers and SalesManagers "from this output?" Now, you could manually edit this, but say you had thousands of different folders, that might be difficult and very time-consuming. Instead, we can take a look at, in the next example, how we can use a word boundary to do just that. So, allow me to scroll down just a little bit and we'll take a look at using a word boundary to do the same thing, but remove the SalesEngineers and SalesManagers from this output. This block of code is exactly the same, except for line 36. So on line 36, I changed the -like operator to -match, and then I'm using the word boundary characters, backslash b, around the word sales, so that way, it will create a word boundary around that word, and hopefully, with any luck, we will use it to successfully match only the Sales groups. So, let's go ahead and run that now, and take a look at the output, and, as you can see, that expression was successful. We only have IdentityReferences of GLOBALMANTICS Sales, and then we have the three separate paths for that. We have the Sales, EU_Sales, and US_Sales, and with that example out of the way, let's collapse this region, we'll clear the screen and then expand out the last region of finding_server_names or database names in a connection string. On line 42, I am populating a variable called $xml by using Get-Content and then pulling in the content from an XML file. I'm not importing it as XML; instead, it's just going to be an array of strings, which you'll see in just a moment. So, if I import that, and then I run this, we'll see that it is a System.Array object. Now, before I run line 46, what I'm trying to do with the expression here is find all the connection strings that have the word customers in it. I want to find the exact database name of Customers so I can replace it possibly with Customer, or maybe just validate that it's there, or that it has the correct password in the connection string. However, when I run line 46, you'll notice that I get two connection strings returned back to me. The reason both of these got returned back to me was because one of them has a database name of Customers, and the other one has CustomersAddresses. So, we can use a

word boundary, like we have throughout this demo, to minimize our results and get only what we need. So, on line 48, that's exactly what I'm doing. I am changing the expression and putting a word boundary around the word customers, and when I run line 48, you notice that I only get the connection string that has the database name with Customers. That wraps up the rest of this module. Let's head over to the summary slide and review what we've learned. I started out this course talking about what word boundaries are, what problems they solve, and how they can be used. We then got into a demo where we discussed word boundaries for start-of-the-line and end-of-the-line anchors, and we learned when to use word boundaries, when does it make sense to use a word boundary versus a start-of-the-line or end-of-the-line anchor. After that, we went into examples where I used regular expression and word boundaries to find specific folders that a group had permissions to. I used it to audit the folder permissions on some shares. After that, we took a look at verifying database names with inside a web.config file. In the next module, you will learn about quantifiers, the little plus sign that I used in this module, to repeat expression patterns.

Using Quantifiers Within Regular Expression

Introduction to Quantifiers

Quantifiers are a metacharacter in regular expression that allow you to repeat characters, metacharacters, and entire subexpressions. They can even be used to make a match optional. Quantifiers are essential for both understanding and using regular expression effectively. On the next slide, you'll be introduced to several quantifier metacharacters. Quantifiers allow you to do a number of things. The first quantifier in this table is the asterisk. When used in regular expression it is often referred to as a star. And what it allows you to do is to match the preceding character zero or more times. There is no limit on how many times it can match. You do have to be careful when using this metacharacter, because it does allow zero instances of match to be found. Which basically makes it optional, and it will match anything. We'll take a closer look at that in the demo. The next metacharacter in this table is the plus sign, which is used to match one or more instances of the preceding character. The only difference between the star and the plus is the plus requires at least one match to be found, otherwise it will fail the match. This can be used when you want to find at least one instance, you want to make sure that it's there, but you don't care how many times it repeats. After the plus sign is the question mark metacharacter, which allows you to make a match optional. Unlike the star metacharacter, the question mark only allows one instance of the match to be found. For example, if there were three words, admin, admins, and adminss with two S's, the question mark expression would only match admin and admins. However it would not match the adminss with two S's, because that would exceed the number of matched instances the

question mark allowed. The next three metacharacter sequences allow you to specify the number of matches by providing a minimum and/or maximum value. To specify the exact number of matches, you would place them inside two curly braces, as seen in the example, backslash d, followed immediately by an opening curly brace, the number four, and an ending curly brace. Which means, match any digit character repeated exactly four times. Now if you didn't want to match something exactly n number of times, you can also say at least n number of times, which is the next example that I have here in this table. And if you take a look at the example provided here, I have the backslash d, metacharacter for digits, opening curly brace, the number three, followed by a comma and then the closing curly brace. The very last quantifier that I have listed here in this table allows you to specify a minimum and a maximum number of matches to be found. In the example, I have the backslash w, metacharacter for word characters, followed by an opening curly brace, the number one, comma, three, and then a closing curly brace. This expression would find a match where there was at least one word character, but no more than three of them. On the next slide we'll take a closer look at some examples of using quantifiers. Quantifiers specify how many times to match a character, but not only literal characters. They can also be used to repeat metacharacters, subexpressions, and even character classes. Aside from specifying how many times they can be repeated, they can also make matches optional. The first example I have listed here demonstrates that by allowing a single expression to match both admin and admins. The question mark after the S makes the S character optional in this match. It also limits the S to only one match, meaning that it would not match two S's, if there was a misspelling of the word admins. I used an end of the line anchor here to prevent the word-within-a-word problem we solved in the last module with word boundaries. The next example on this slide demonstrates using a quantifier to repeat a negated character class. The plus sign after the closing square bracket of the negated character class repeats that expression. The plus sign is used to match one or more instances of this negated character class. The last example uses quantifies to accomplish a few things. The first is to repeat a subexpression. I haven't gone over subexpressions quite yet, but they are indicated by the opening and closing parentheses. Because I placed the plus sign outside of those parentheses, it is used to repeat the entire expression inside of it, which is looking for a digit character, repeated exactly three times, followed immediately by either a dash or a period. Now this expression is going to allow that to run as many times as necessary, and in this case, it's going to run that expression twice, to match 402 hyphen 583 and then another hyphen. After that in the expression, I'm using the quantifier to match the digit character exactly four times, to match the rest of the phone number, which is 5366. Quantifiers are an extremely useful metacharacter, but the downside of these expressions are, they can get complex pretty fast. Notice how these expressions shown here look a lot like those crazy expressions that you find on the internet. In my opinion, this is why people stay away from regular expression. Because it appears very complex at first, if you don't have any foundational

knowledge of any of the metacharacters. Luckily, that's not you, because you're watching this course. You can accomplish a great deal by using quantifiers, so let's move on to the demo so you can learn to use these quantifiers in your expressions.

Quantifier Basics

In the upcoming demo, we'll start off by talking about the fundamentals of quantifiers. We'll begin that by talking about repeating metacharacters by using quantifiers, then discuss the pitfalls of using the star quantifier. After that, I will discuss finding one or more matches with the plus sign quantifier, followed by combining anchors and optional matches with the question mark metacharacter. Once I've gone over some examples demonstrating the fundamentals of using quantifiers, we will use them to discover service accounts, within active directory, and to validate phone numbers that are pulled from active directory. The first quantifier that I will demonstrate is the asterisk, commonly referred to as the star in regular expression. On line three, I have a string that contains the word expression, followed by an exclamation mark at the end. To match that string, I have an expression that repeats the period metacharacter 11 times. The period is a metacharacter that will match anything including the special character at the end, and when I run this, not too surprisingly, I do get a match, and if we take a look inside the matches variable, we can see that it matched the entire string, all the way up to the exclamation mark. Now wouldn't it be great if I didn't have to repeat that period 11 times? Well luckily, that is exactly when you would want to use a quantifier. On line five, I have the same string, expression with an exclamation mark. Instead though, in the expression, I have just one period listed, and then I'm using the asterisk or the star metacharacter, that quantifier, to repeat that period zero or more times. Now more times means infinitely, so this would match anything inside the string. And when I run line five, we will get the same result as on line three, so if we take a look inside the matches variable, we'll see that it matched the entire expression. And we reduced the expression from 11 characters down to two. That right there demonstrates the power and the reason for using quantifiers, it alleviates a lot of typing. Now there is one caveat, and I truly believe that the dot star is overused in regular expression, because it matches anything, and it's a quick and easy way to kind of replace a wild card. However, it is dangerous, because if you take a look at line seven, when I run this, it'll return true. Because you're matching anything zero or more times, so it will match absolutely nothing. So you have to be careful when you use the period, which matches anything, with a quantifier. Especially the asterisk. The next quantifier that we'll take a look at in this demo is the plus sign quantifier. Line 12 looks very similar to line three, except for I removed the exclamation mark at the end, and I've changed the expression to use the backslash w metacharacter to match only word characters. So, to no surprise, when I run line 12 I get a match, and when I look inside the

matches variable, it matched the entire word, expression. Now we can use a plus sign quantifier to repeat that w as many times as possible. The only requirement for this plus sign quantifier is that it has to match at least one time, meaning it would have to at least match the e. So when I run this, we'll get the same exact results. So the match will be true, and it will match the entire word, expression. Now unlike the example above, when I run this against an empty string, it will return false. As you can see here. And the reason is, the plus sign quantifier needs to match at least one instance of it, so if there isn't a valid word character within the string, the match will fail. Let's now move on and take a look at how we can make matches optional with the question mark metacharacter. On line 22, I have three words, expression, expressions, and then expressions spelled improperly with three S's. The expression that I'm using to match these words is expressions with a question mark right after the s, making that character optional. The S only impacts the preceding character, the one right before it. I am then anchoring that to prevent the words-within-a-word problem that we took a look at in the previous module. And when I run line 22, we should only get the first two entries returned to us, making that expression and expressions matchable. The one with three S's shouldn't have been matched. And as you can see from the output, the one with three S's wasn't returned, we were able to properly use that expression to make the S optional within that. Now we could have used an alternation and put expression, hype symbol, expressions, but that would have required a lot more typing, and this is where it becomes really nice and easy to use that question mark, to make individual characters optional. Now, you can use that against a character class, or against a subexpression, which we'll take a look at in a later module. Now on line 44, it's exactly the same, you have the expression, expressions, and then expressions with three S's, except for, I'm changing the regular expression used here to include the asterisk at the end instead of the question mark. And when I do that, it's going to match all three of them. Because it doesn't care how many times the S is repeated, it's going to find a match. Now that we have all the somewhat academic examples out of the way, let's move on to some practical examples of using quantifiers to do certain things with inside a PowerShell.

Crafting Expressions with Quantifiers

In a previous module, we used the shorthand metacharacter, backslash w, to find several active directory accounts. Specifically we were looking for a service account, but just like what we ran into a problem with earlier, when I run this expression on line 30, I get two accounts returned to me, I get DefaultAccount and svc_account. Taking a closer look at the expression I used on line 30, I am using the backslash w metacharacter, then the plus sign quantifier, followed by the word account. What the slash w and the plus sign quantifier is doing is it's matching any word character one or more times. That is why both the default account and the svc_account was

found in the results. Remember the underscore is considered a word character. My intent with this expression was to find only service accounts, and if I wanted to do that, I have to make this expression more specific, and that's exactly what I'm going to do on line 32. The expression on line 32 starts with the backslash w metacharacter looking for word characters, it then has an opening curly brace, the number three, and a closing curly brace. This is going to look for a word character exactly three times. No more, or no less. Then, the underscore, I'm getting a little bit more specific here, and I'm declaring only underscore should be in that position. After that, I'm using the backslash w metacharacter again followed by the plus sign quantifier, to look for one or more words, but no limit on the number of word characters found. And when I run line 32, you'll now notice that I only get the service account returned to me. The svc_account returned. In this expression, I used two different types of quantifiers. One was used to specify the exact amount of matches to allow, and the other was used to match one or more instances. There isn't a limit on how many quantifiers you can use within side an expression. I will now move on to the last example in this demo, which takes a look at validating phone numbers and phone extensions. On line 38, I am populating a variable called dollar phone number, with the user J. Green's office phone. I'm pulling that from active directory and then just populating that variable. I will go ahead and run line 38, and then we'll take a look inside that variable, dollar phone number, just to see what the phone number is. So it's 402-583-5366. And then on line 40, I'm taking that variable and I'm running it against the expression backslash d, plus sign, then a hyphen, repeated twice, followed by backslash d, then the plus sign right after that. And to break down this expression, it is looking for any valid digit character, zero through nine, repeated one or more times, followed by a hyphen, then another digit character, one or more times, another hyphen, and then it ends with looking for any digit character one or more times. And when I run line 40, we will get a match. And that match will have found the entire phone number, 402-583-5366. Now we know this isn't the only format a phone number could come in, it could also be an extension or a long distance number, and that's what we'll take a look at next. On line 42 I am populating a variable called dollar extension, with an extension from the user B. Bailey. So I'll go ahead and run that. We'll take a look inside that variable just to confirm that it's an extension. So just a random number, 3956, and then we're also going to populate a long distance variable on line 43, from the user G. Bailey. Now these are both from the office phone property of the active directory user. And when we look inside long distance, we can see that it is prefixed with a one hyphen. Now on line 45, we have both of those variables, and we're going to run them against this very long expression that I will break down for you. Now before I explain that expression, I'll go ahead and clear the screen just so we have some working space. Ah, much better. Alright, back to the expression on line 45. Now, immediately you can recognize that this is way more complex than any of the other expressions you've seen through the course. This probably resembles something that you might see online. But, it isn't that bad if we slowly walk through this expression and break it down piece by piece. I'll start by

re-stating what the intention of this expression is. The intention for this expression is to match both the extension and the long distance numbers that we pulled from active directory. And just so we have a frame of reference, I'm going to run both of these variables so we can see what their contents are, and the numbers that we're trying to match with this expression. Alright, now let's begin walking through what this expression is meaning. It starts off with the backslash d metacharacter, followed by a question mark. Now this is making that character completely optional. And it does the same thing for the next hyphen. This right here is used to match the one hyphen from the long distance number. The next part of the expression is another backslash d metacharacter, looking for exactly three digits. Now this is going to match the 760 part of the long distance number. After those three digits, it's looking for a hyphen, and then another three digits. This section of the expression matches the hyphen, the four, the one and the zero, after the 760 in the long distance number. Right after the second set of three digits, there is another hyphen, and then another backslash d looking for exactly four digits. This part of the expression matches the hyphen 9010 right after 410 in the long distance number. Now if we ended the expression right there, we would match the long distance number with that expression that I've read off so far. However it wouldn't match the extension, and that's where an alternation comes in. So we're using the pipe symbol to indicate that there's going to be an alternation with inside this expression, and then there's a backslash d with the opening curly brace, three comma five closing curly brace. The backslash d in the second part of the alternation is pretty obvious, it's trying to match a digit character. But then it is looking for three to five instances of that digit character, indicated by the three comma five. So this will only validate extensions that have at least three numbers, but no more than five. Now that I've explained that entire expression, I will go ahead and run line 45, and with any luck, the expression will match both of those variables. So as you can see from the results, it did match the extension of 3956, and it also matched the long distance number of 1-760-410-9010. That wraps up the rest of this demo, and this module. Before we head over to the summary slide, I just want to say that even though this expression appears complex, it can be broken down into manageable chunks so you can understand what the expression is doing. So with that said, let's now move on to the summary slide, and review what we've learned. Throughout this module, you learned about quantifiers. You learned how to use them to repeat characters, character classes, and subexpressions. The first quantifier that you were introduced to was the asterisk, or star, quantifier, which is used to match zero or more instances. You then learned about the plus sign quantifier, which is used to match one or more instances. After that, you learned to make matches optional with the question mark quantifier. And the last quantifier that we discussed in this module allowed you to specify minimum and maximum number of matches allowed. After learning about all the different quantifiers, I then demonstrated using several of these quantifiers to discover active directory service accounts, as well as how to validate phone numbers with inside active directory.

Capturing Matches with Subexpressions

Introduction to Subexpressions

Sometimes you'll want to group certain parts of your expressions to either capture it or repeat it. To accomplish this, you'll need to use something called a subexpression. These subexpressions will then create what's called a capture group, which stores the values of that subexpression in a special variable, which we'll talk about on the next slide. Before I show you an example of using a subexpression, I'm first going to show you an example of how to extract a server name and a share name from a UNC path by using the knowledge you've gained throughout this course already. In this first example that I have here, I've chosen to use the split operator to split on the backslash. There are two backslashes here, because the backslash is a metacharacter, and I have to escape it if I want to use the literal character backslash. Now, when I split this UNC path, it will create an array with all the different values in it. Item two will contain the server name, and item three will contain the share name. By using the square brackets and putting two comma three, I'm effectively selecting those two items out of the array, so I would only get returned to me the server name and the share name. You could easily accomplish this with the knowledge that you've already gained throughout this course. In the next example I'm going to accomplish the same exact thing, but I'm going to be using subexpressions to capture those different values instead of using the split operator. In this next example I am using the match operator with an expression that starts with four backslashes followed by an opening parenthesis, then the backslash W metacharacter, followed by a plus sign quantifier, and then a closing parenthesis. In between the parentheses is the very first subexpression, which creates a capture group containing the server name. This first subexpression is then followed by two more backslashes, then another subexpression that captures the share name by using the backslash W metacharacter and the plus sign quantifier again. Because I don't care to match the last part of this UNC path I didn't include any additional things in this expression to match the file.txt. The two subexpressions store their values in special variables, which we'll take a look at in the upcoming demo, but, for now, just know that the very first subexpression stores its values in a dollar one variable, and the second one stores its values in a dollar two variable. Now that you have a better idea of what subexpressions are and how they can be used, let's move on to the next slide. As you just saw, subexpressions can be used to match specific sections of a string or block of text. You can accomplish this but putting parentheses around what you want to capture. In the first example I have listed here on the slide is a phone number and an expression that matches the entire phone number. I've used parentheses to capture the area code of this phone number. When I do that, a new capture group is created, and, in the upcoming demo, you'll notice that a new

item is added to the dollar matches variable, which contains the newly-created capture group. One very useful thing that you can do with these subexpressions is replace the string or sections of the string. In the second example I have the same exact phone number. I am now using the replace operator, not the match, and with inside the expression I've updated it to use the backslash D metacharacter to look for digits and the plus sign quantifier. I've changed the subexpression to not capture the area code anymore. It now captures the phone number, and, as I talked about a few times, that capture group is being stored in a variable called dollar one, so I can use that capture group to replace the entire string with what was captured. If I were to run this, as you'll see in the upcoming demo, it will replace this entire string with the phone number that includes the area code to just a phone number without an area code. The last example I have shown here demonstrates two things: first, how to use multiple subexpressions and, secondly, how to use a non-capturing subexpression. The string I am matching is a UNC path that you saw on the previous slide, and the expression I am using to match this UNC path starts off by matching the two backslashes at the start of the UNC path, and to do that I have to use four backslashes, because I have to, of course, escape those. It then moves into capturing the server name inside the very first subexpression. That first subexpression is using the backslash W metacharacter to match any word characters. It is then followed by a plus sign quantifier to match multiple word characters. In this case, it would match the word server. That very first subexpression is followed by two more backslashes to match the backslash between the server name and the share name, and right after that is another subexpression which uses the backslash W metacharacter and the plus sign quantifier to match the share name, and, in this case, it would match the word share. Right after that is the third and final subexpression with inside this regular expression. If you ignore the colon and the question mark for now, the period and the star quantifier are matching the rest of this UNC path, which is the final backslash followed by file.txt. Going back to that colon, question mark, what this indicates is a non-capturing subexpression, meaning that the values within here are not going to be stored in a dollar something variable, dollar one or dollar two, and, in this case, they would be stored to dollar three, but, because it's non-capturing, it won't store anything in that variable. This becomes more useful when you're wanting to repeat subexpressions, but you don't necessarily want to capture what's inside of them. We'll take a closer look at that in the upcoming demo.

Learning How to Use Subexpressions

In the upcoming module, there are four main concepts you will learn, the first of which is how to capture sections of a string or a block of text using a subexpression. The second concept is how to find and use capture groups that were created by those subexpressions, and the third concept is how to use multiple subexpressions with inside the same, regular expression, and,

finally, we touch on a concept of how to use a non-capturing subexpression, which will prevent that capture from being stored in a capture group. To demonstrate all of these concepts, I will be using subexpressions to accomplish several real world tasks such as capturing a server name from a UNC path, extracting a user's name from a distinguished name in Active Directory, as well as converting a distinguished name to a domain name, and then I will end the demo by demonstrating how to extract event source node names from a windows event forwarding subscription. The very first example in this demo starts off by extracting an area code from a phone number, so, as you see on line five, I have a phone number which includes an area code of 202. The expression I am using is matching that phone number exactly by using the exact same digits. However, in the expression, I am using parentheses to capture the area code, and, when I run line five, we'll get the results of true, which isn't really a surprise. It did match the phone number, but, when we take a look inside the matches variable, it looks a little bit different than what we've seen previously. There are actually two entries inside the dollar matches hash table, one with the name of zero, and one with the name of one. The entry with the name of zero should look familiar to you. It's the one that we've seen throughout this course, and it's always going to be there, because what's stored in the very first entry of matches is the entire match for the regular expression used against the matches operator, and, in this case, it is the entire phone number. What's interesting about this is the entry with the name of one. This is the first supbxpression used in this regular expression, so that value is going to be captured to this entry, and, as you can see over in the value column, it does have a value of 202, which matches the area code captured by the supexpression on line five. The first example was easy enough, so now let's move onto the second example where I'm using the same phone number, except for I've updated the regular expression. Instead of using the exact numbers of the phone number, I've changed the regular expression to use the backslash D metacharacter and the plus sign quantifier. I have also added an additional subexpression. When I run this, it will now capture the area code and it will capture the phone number. It will sort each of those in different capture groups, so I will go ahead and run this line now, and then we'll take a look inside the matches variable, and, as you can see from the output, there are now three entries. There's the entry of zero, which has the entire regular expression capture. Then it has the entry of one, which contains the first subexpression used within this regular expression. That captured the area code. The second subexpression, but the third capture, is the entire phone number, which is stored in the entry with the name of two. What this example demonstrates is with every additional subexpression you put inside your regular expression, another entry will be created with an incremented number, so, if we had another subexpression, it would create an entry with the name of three and so forth. It's also important to know that these capture groups are created from left to right. Now, having just said that, there is one exception to that, and that's when you use a non-capturing subexpression, which we'll take a look at next. Starting on line 13, I'm demonstrating using a non-capturing

subexpression, so I have the same phone number, except for this time I'm using the replace operator, which we'll talk a little bit more about in just a second. The regular expression starts off with this subexpression with a question mark and a colon. This is how you indicate that a subexpression is non-capturing, meaning it will not create or store anything to a capture group. Within that non-capturing subexpression, I have the backslash D metacharacter, and then I'm specifying that it should match exactly three times, followed by a hyphen. If you look outside that first non-capturing subexpression, I have a plus sign quantifier. You can use quantifiers to repeat entire subexpressions. After that I have another subexpression. This time it will capture what's inside, and what's inside is a backslash D metacharacter. We're looking for exactly four digits, and then, if you look right after the regular expression, I am replacing this entire string with what was captured in the dollar one variable, and, in this case, it's going to be 0148, the last four digits of this phone number, so let's go ahead and run line 13 and take a look at what happens. As I predicted, the results were 0148, which is the last four digits of that phone number, so let's now move on to the last example in this demo before moving on to some more practical examples of using subexpressions. On line 17, I have the phone number again. This time I'm using the select string commandlet, and then I'm using a regular expression pattern that captures the backslash D metacharacter in a quantifier. I'm effectively just repeating this expression one or more times, so the hope is it's going to capture the digits for the area code, the middle of the phone number, and then the end of the phone number, effectively just stripping out the hyphens and then capturing each of those different sections into its own capture group. I'll run line 17 and then we'll take a look at the results. The very last capture was 0148, which is the last four of the phone number. That's good. That worked, and the next capture group was the middle of the phone number, 555. If we scroll up just a little bit more, we should see the area code, 202, was captured as well. This is just one way that you can use quantifiers to repeat subexpressions and then capture all of those results. Now that you have a better understanding of how to use subexpressions with inside of a regular expression, let's move on to some more practical examples.

Extracting Matches with Subexpressions

Starting on line three, I will first demonstrate how to pull a server name from a UNC path. The expression that I'm using to do that starts with an anchor character followed by four backslashes to match the backslashes in the UNC path. It then moves into a subexpression that has a character class and a plus sign quantifier in it. The character class is being used to match the following characters: hyphen, A through Z, upper or lowercase, because Powershell is case insensitive, any number zero through nine, and the underscore character, all of which are valid server name characters. Right outside the character class is that plus sign quantifier that I

mentioned, so this is going to be repeating any of those characters found in the character class one or more times until it finds a backslash. There are two backslashes here, because I have to escape it 'cause it is a metacharacter. After that backslash is the period star. This is being used to match the rest of the string, which is the share, the folder, and the powershell.ps1. I previously warned you about using the period star metacharacter sequence, but this is actually a really good place to use it, because I don't care about the rest of the match. I only really care about what's captured in the subexpression. Using the period star here allows me to quickly match the rest of the string without exerting a lot of effort into matching all of the different characters, so I'll go ahead and run line three now, and we get a match within there, but, when we look inside the matches variable, we'll see that there are two capture groups. Capture group zero is the entire expression match, which is the full UNC path, and then capture group one has the server name, which is just server in this case. Let's move on to the next example in this demo where I will take a look at extracting Active Directory data. On line 11, I have the distinguished name for the administrator account. I am then using the replace operator with the expression CN=, then a subexpression that contains the backslash W metacharacter and the plus sign quantifier. After the subexpression is the period star to match the rest of the distinguished name. I am then replacing the entire string with what was captured in the first capture group. When I run this, we should get returned to us just the word administrator, and, as you can see from the output, that's exactly what happened. We now converted the distinguished name to a username. In the next example I will be converting a distinguished name to a domain name. On line 15, I am creating a new variable called dollar distinguished name, and I am populating it with the distinguished name of the user ssmith. I will go ahead and execute line 15 to populate that variable and then, on line 19, I am taking that distinguished name and using the replace operator to convert the distinguished name to the domain name. The regular expression I'm using to accomplish this starts off with a non-capturing subexpression. With inside this non-capturing subexpression is the dot star metacharacter sequence, which will match everything up to the characters DC. This entire subexpression, non-capturing subexpression, is optional. It's optional because there's a question mark outside of the subexpression. Right after that non-capturing, optional subexpression are two literal characters, DC, and then the literal character equal sign. This is going to match exactly those characters. There isn't anything tricky about that. There's no quantifiers, just those literal characters. After those three characters is another subexpression with the period star metacharacter sequence. This is going to match anything zero or more times that follow DC= up until there is a comma. Now, that comma is right outside of that subexpression, so it won't be captured inside that first capture group. Right after the comma there are three more literal characters, DC and then the equal sign. Right after that equal sign is the last subexpression. This is a capturing subexpression, and it has an alternation with inside of it that will match com or net. After that entire expression I have a comma and then what I'm

going to replace this string with. I'm going to replace this entire match, which will be the full distinguished name with whatever was in the first capture group followed by a period and then what is inside the second capture group, so, when I run this, if the expression is correct, I should get globomantics.com, which is the domain for this Active Directory environment, so let's go ahead and run line 19 and see what we get, and, as you can see from the output, we did get the domain name. I can tell you that this, right here, was extremely useful when I was working in a cross force environment where I needed the server, or the domain name, rather, to put to the server parameter of the Active Directory commandlets. Being able to find the domain name from a distinguished name saved me hundreds of hours, if not maybe even thousands of hours, because I had to audit Active Directory groups that had cross force nested groups inside of all of them, and, without this, I wouldn't have been able to query for each of those groups of Powershell. Thousands of hours is probably a slight exaggeration, but it definitely did save me at least a hundred plus hours. All right, it's time to move on to the last section of this demo and this module where we take a look at finding event sources from a windows event log subscription. If you aren't familiar with windows event forwarding, it's just a simple way with inside windows that you can forward events from one server to another.

Capturing Event Sources from Windows Event Log Subscriptions

To my knowledge, there isn't a way natively in PowerShell to query the event log subscription properties. There is however, luckily, a command line utility we can use to query these properties. Before I run line 24 and take a look at this subscription, let me set the stage for what we're trying to accomplish here. The overall goal here is to capture all the event sources from the subscription app events. Event sources is just the term that it uses to define the servers that are forwarding it events with inside a given subscription. With the stage now set, let's take a look at the app events windows event forwarding subscription. To do that I'm going to use the wecutil command line utility followed by gs and then the subscription name, which is appevents. I'll go ahead and run line 24, and we'll take a look at the output. At the very end of this output, we see the event sources. We see the event source number, which is zero, the address, and that it is enabled. However, if you scroll up just a little bit, you'll see a lot of additional information about the subscription. This is returned back to us as an array, and, because it's an array of text, we cannot just simply say, "Give me just the event sources". We can't pipe it to our object and just select that out, and that's where regular expression comes in and really helps us to parse this information. Before I show you how to extract the event sources by using regular expression, I'm first going to show you what I'll call a non-regular expression approach. Starting on line 27 is the non-regular expression approach of how to accomplish this. What I'm doing is I'm taking the output from line 24, I'm piping it to select

string, and then I'm matching it against a simple match of address. I take that output, and then I convert that to a string, and then I take that string and I use the split method to split on the colon, and I grab the second item out of that split array, because the array starts at zero, and then I take that second item and then I use the trim method to trim off any white spaces, so, when I run line 27 and 28, I should get the fully-qualified domain name for the server inside the app event subscriptions. Let's go ahead and run that and take a look at what the output is. The only server inside this event log subscription is GDC01.globomantics.com. What if I didn't want the fully-qualified domain name and I just wanted the server name? I canuse the split method again, as you can see on line 31 and 32. 31 and 32 is the exact same as 27 and 28, except for, at the very end, I'm using the split method again and splitting on the period, and, this time, I'm grabbing the very first item out of that array, which is the server name without the domain name. When I run this, we should just get returned to us GDC01, and, as you can see, GDC01 was returned to us without the domain name this time. When you don't know regular expression this is the typical thing that you choose to do. You choose to split and then replace, and split and trim, and split again until you get what you want. Granted, if you don't know regular expression, this approach is probably easier for you to do and read and understand. However, there is a better way to do it, in my opinion, and that's with regular expression, and, to see how that's done, let's take a look at lines 35 and 36. Line 35 starts off very similar to the previous two examples. By using the wec utility to get the event subscription properties, I then pipe those properties to select string and use a simple match against address and then a colon. I then take that output and then use the two string method. After that I use the replace operator with the following expression: backslash S to match any space character at the beginning, the word address, a colon, and then the backslash S metacharacter again with a plus sign quantifier to match one or more spaces, and then I'm using a subexpression to capture any word character repeated one or more times with the backslash W metacharacter and the plus sign quantifier. Right outside that expression you can see that I'm going to replace everything in this string with whatever was captured in that first subexpression. That subexpression, of course, is stored in a capture group in the dollar one variable. When I run this, I should get the fully-qualified domain name of the server, so let's go ahead and find out what that returns, and, as you can see from the output, it did return the fully-qualified domain name, GDC01.globomantics.com, and now, if I wanted to accomplish the same thing as the second example I showed where we trimmed off the globomantics.com, we can update the expression to include a non-capturing subexpression that will capture anything after repeated word characters. What this is really doing is it's matching right after GDC01, it's matching that period, and then everything after that, which is globomantics.com and then it's going to capture in that non-capturing subexpression, so, when we reference the dollar one variable to replace it, it's going to give us just GDC01. Let's go ahead and run lines 39 and 40 and see what the results are, and, as you can see from the output, GDC01 was returned to us without the domain name.

That's it for this module. Let's now head over to the summary slide and review what you've learned. We started off this module talking about subexpressions. We talked about how they are used to capture sections of an expression, that they create additional capture groups with inside the dollar matches variable These capture groups are stored in special variables starting at dollar one and incrementing by one for each additional subexpression. Subexpressions allow you to easily repeat sections of an expression by repeating the entire subexpression with a quantifier. You can also use non-capturing subexpressions when you want to repeat an expression, but you don't necessarily want to capture it. Throughout this module, we used subexpressions to accomplish a number of things. The first was to capture a server name from a UNC path. I next demonstrated how you can use regular expression to extract information from Active Directory, both how to get a username from a distinguished name and how to convert a distinguished name to a domain name. The last thing that we covered throughout the demos was how to extract event sources from a windows event forwarding subscription.

Capturing Matches with Named Captures and Modifiers

Introduction to Named Captures and Modifiers

You have already learned how to capture matches by using subexpressions. You also know doing so stores the capture in a numbered capture group. It is also possible to store them in what's known as a named capture group. As the name implies, the named capture group gives you the ability to name each of the subexpressions you're capturing. Name captures aren't much different than normal captures. You still have to use a subexpression or parentheses to indicate that it is going to be a capture group. Now, how you name that capture group is by adding a question mark at the very beginning, right inside the parentheses, followed by a less than sign, then the name of the capture, a greater than sign, and then after the greater than sign will be the regular expression used to populate the named capture. In the example that I have here, I have a string that contains four letter characters and four digit characters. In between those is a space, and to match that string, I'm going to use an expression that creates two named captures, one with the name of word and one with the name of num. The one with the name of word will capture all the letter characters, and the one with the name of numb will capture all the digit characters. In an expression as small as this one, it doesn't add a lot of value. It just creates some organization, and I don't have to remember which one's the letter and which one's the word by counting them from left to right. However, they become extremely useful when you have a very large regular expression that has several, if not, you know, 10 or 20, maybe even 30 captures in it. That way you can just reference them by the name, which is much easier to remember than its number. You've probably noticed that the regular expressions are starting to get longer and longer, making it harder to read and

understand. Wouldn't it be nice if we could write the regular expression to span multiple lines? Or what about putting comments in the expression to remind us what we were trying to accomplish at any given point in the expression? Well, you can. You can accomplish both of those things by using what's called a modifier. Modifiers change the way that the regular expression engine interprets characters. By using the x modifier, we could tell the regular expression to ignore un-escaped white spaces in an expression. This would allow us to write the expression across multiple lines and to put comments in it. Modifiers can be used for a number of things, such as turning off case sensitivity or to enable single or multi-line mode. Single-line mode allows the period metacharacter to match new lines, and the multi-line mode character changes the start of the line anchor and the end of the line anchor to match on each line instead of just the start and the end of an input string. Another very useful modifier is the n, which makes all subexpressions non-capturing unless they are named. We'll take a closer look at this in the upcoming demo. To use modifiers, you must place them with inside a subexpression at the very beginning of your regular expression. Right before the modifier character, you must put a question mark to indicate it's a modifier. It is possible to use multiple modifiers at once.

Using Named Captures

In the upcoming demo, you will learn about named captures. Both how to create name captured groups and how to convert those groups to an object. You will also learn about the regular expression .net object, both how to create .net regex objects and how to use their methods. Throughout the upcoming demos, you will also learn how to use modifiers. You will use modifiers to ignore un-escaped whitespaces in your expression. You will learn how to add in-line comments to your regular expressions, as well as preventing captures from non-named subexpressions, and how to enable single-line mode and what that means. The first example that I have for this demo starts on line three. It begins with the string abcd space 1234. The expression that I'm using to match this string starts off with a subexpression. Inside that subexpression is a question mark, then a less than sign, and then the word word. After that is a greater than sign and then the backslash w meta character for word characters, and then a plus sign. The plus sign is of course a quantifier that indicates it should match at least one time, but doesn't have a maximum number of times it can match. All of that combined is the first named capture. Right after the first named capture group is a space. After that space is the next named capture, which starts with the parentheses indicating the subexpression. Inside of that, of course, is the question mark, the less than sign, the word num, representing number, and then a greater than sign. After that greater than sign is the backslash d metacharacter and another plus sign quantifier to match one or more digits. So in this case, it will match 1234. So let's go

ahead and run line three, and then take a look at the matches variable to see what happened with the capture groups. Taking a look at this output, the capture group with the number zero is still there, that captures the entire expression. That's always going to be there. But what's interesting is the capture groups are now named. They are now named word and num. The one with the name of word has the word characters in it, and the one with the name of num has the numbers in it. The next example that I have starts on line five, where I'm populating a variable called $share with the share name of the netlogon path for a GDC01. On line seven, I have the regular expression that I'm going to be using in this example stored in a variable called $regex. Just like the above example, I am going to be using some named capture groups, and by doing that, it made the expression a little long, so I chose to separate out into different lines and just put them into variables instead of one long giant line. So if you look at line seven, we'll take a closer look at the regular expression that I'm going to be using. It starts off with a carrot character, which is the begin of line anchor, matching the very first position, followed by four backslashes that will match the two backslashes of the file share path. Then is our first named capture group. It starts off with a subexpression, then a question mark, a less than sign, and then the word ServerName, which is going to be the name of this first capture group, which will of course capture the server name. After ServerName is a greater than sign followed by the expression used to match the server name from the share path, and for this expression, I chose to use a character class that will match any letter, and because PowerShell is case insensitive, it will match upper and lower case as well as any digit character zero through nine one or more times. The one or more times comes from the plus sign quantifier outside of the character class. Right after that first named capture is two more backslashes to match the slash in between the server name and the share name. Now, after those two backslashes in the expression is the second named capture. For the second named capture, I have updated the name to ShareName, and then I have changed out the expression to match any word character with the backslash w shorthand meta character and then the plus sign quantifier to specify one or more times. So now that we have a better idea of what the regular expression is doing, I will go ahead and run line nine, and then we'll take a look inside the matches variable. And as you can see from the output, we now have a capture group with the name ServerName which has a value of GDC01, and then we have a capture group with the name ShareName which has the value of NETLOGON. Throughout this course, I've been using the match operator in the select string cmdlet. However, there is another way that you can use regular expression with inside PowerShell, and that's to use the .net regex object, which I will show you next.

Creating and Using .Net Regex Objects

Before I begin the regex demonstration, I will go ahead and clear the screen, and starting on line 15 I am populating a variable called $string, and I am populating that variable with abcd space 1234 space efg space 567. Now, the goal of this expression is to capture all of the word characters into the named capture group of word, and all of the digit characters to the named capture group of num, just like above. However, this time we're going to use the regex object. One way of creating a regex object in PowerShell is to typecast it to a variable, as you can see on line 17. Line 17 starts off with a square bracket, then the word regex, then a closing square bracket. After that is the variable name of the regular expression object, which is in this case $regex. You set it equal to the expression that you want to use, so the expression that we have here is very similar to the one that you very first saw at the beginning of the demo, and I'm just using it to capture word characters with A through Z character class into a named capture group of word, and then it has a space, followed by the second capture group, named capture group of num. Which is going to capture any digit characters zero through nine. Now, if you're an operations or sysadmin, this next step might seem a little bit weird, but if you're a developer, it will make total sense. What we need to do now is use the match method off the regex object to compare the stream we have against the expression inside of the object. And that's what happens on line 19. Lien 19 starts off with $regex, then a period, and then the method that we want to call, and in this case, we are going to call the Match method. To use this method, we have to put the input string, which in our case is $string, inside parentheses right after Match. So let's go ahead and run lines 15 through 19 and see what the Match method returns for us. I will scroll up just a little bit to make sure that we're seeing everything. Now, this output doesn't look like the results that we get from the Match operator inside this Matches variable. This looks a lot more like the select string output. This output isn't as obvious to see what matched where and what the capture groups' names were. You actually have to use a different method for that, which we'll talk about in a second. But as we examine this output, we can tell that the regular expression did work by looking at the captures and values. Inside of captures, we have abcd and then 1234, so we know at least the very first word set and the very first digit set did get captured. You can verify that by looking at the value. The values also contain the same thing, abcd 1234. Now, what about efg and 567? Well, because we used the Match method, it's only going to match the very first instance that it found of those. We would need to use the Matches method if we wanted to find more than one match. And that's what we're going to take a look at next. On line 25, I am repopulating the $string variable just so that you can see what the values are as we walk through this example. On line 27, I am using the regex object that we created above, and then I'm calling the Matches method against that string variable we redefined on line 25. And since I'm using the Matches method now, we should get two match objects returned to us when I run line 27. And if we scroll through this, we see that we do get two match objects returned, we get one that matches abcd and 1234, and then the second match object matches the efg 567. One thing that isn't included in this

output is the name of the capture group itself. You know, we name these for a reason, and we should be able to filter by those names. And we can't, so on line 29, I'm doing the same thing as I did on line 27, I am using the Matches method and then putting the string in there, except this time I'm storing the values to a variable called $m. Just go ahead and do that now, and now that we have the match results stored in the $m variable, we can now use the ForEach-Object cmdlet to filter out all of the values for each of the named capture groups. Line 30 will filter the $m object and only return the values that are within the capture group named word. So let's go ahead and run line 30, and as you can see the results that we got returned to us was abcd and efg, which was the capture for the word characters in each of those groups, each of those match objects. If we scroll up to the top a little bit, we can see that efg was the named capture for the word group inside the second capture group. If we scroll up a little bit more, we'll see that abcd was the word capture for the first group. So this is just one of the ways that you can find the values for a particular named capture. Line 31 does the same thing, but now looks for all of the num capture group values. So let's go ahead and run line 31 and take a look at the results. And not too surprisingly, we get the digits returned to us from both of those match objects. So now we've created regex objects, we've used the Match and Matches operator, and we've also filtered out all the values for a particular named capture. But what if you forgot what those named captures are? Well, there is a method that you can use to find all of those from your regex object. Which if you scroll down just a little bit, line 33 demonstrates that. So we have a $regex object, and then we're using the GetGroupNames method from it. And when I run this, we should get returned to us three different capture group names. We see that we have the zero, which is always going to be there, and we have the word and the num named captures. So if you had forgotten the names of your capture groups, you could use this method to retrieve those names. In the past few examples I have been using the regex object by putting it inside a variable. It is possible to use it without storing it in a variable, as you see on line 35. Now to avoid storing it in a variable, you have to put an opening square bracket, the word regex, closing square bracket, two colons, and then the method that you want to use. So in this case, on line 35, I'm going to use the Matches method. Now, after that, I have to put the string that I'm going to run the regular expression against, and then a comma, followed by the regular expression that I want to use against that string to attempt some matches. So when I go ahead and run this, we should get the same results as before. We'll get two different capture groups that collect all the word characters and numbers and then store them into named captures. On lines 37 and 38, I'm expanding what I did on line 35 by piping it to the ForEach-Object and then filtering out all the values that belong to the named capture word. Let's go ahead and run those now. And as you can see, I get the values abcd efg returned to me. Which are the values inside of the named capture word.

Using Modifiers

As I mentioned in the slides, modifiers are used in regular expression to change the way the regex engine interprets characters. The first modifier I will be using is the s modifier, which stands for single-line mode. Also known as dot matches all. By default, the period or dot metacharacter matches every character except for the new line. But by using the s modifier and putting the engine in single-line mode, we can change that behavior. When you use the single-line mode modifier, you allow the period or the dot metacharacter to also match new lines as shown on lines five and six. Starting on line five but continuing on to line six, there is a string that contains three words, dot matches all. I am then using the Match operator followed by the expression opening parenthesis question mark s closing parenthesis, the period or dot metacharacter, and the star quantifier. Now, before I run lines five and six, it's worth noting that if I hadn't used the single-line mode the dot star would only match the word dot, because it does not match the new line characters by default. But however, when I run lines five and six, we'll get that the match was true, and if we look inside the matches variable, we can see that it matched dot and then something else. But if we wanted to see that something else, we can call out the name capture of zero. And see the entire capture, which was dot matches all. The next modifier that I will be demonstrating is the x modifier, which is used on line 10. The x modifier is used to ignore un-escaped white spaces. By doing that, this modifier allows you to put spaces in your expressions as well as comments, making the expressions easier to read. It can even span multiple lines, as you'll see later. On line 10, I have a string that contains a phone number. I am then using the match operator. At the very beginning of the regular expression is the x modifier. It's using the subexpression with a question mark followed by an x to indicate the ignore un-escaped whitespaces modifier. Throughout the rest of the regular expression, I use the backslash d metacharacter, and then I specify how many times it should match. One difference between this expression and ones that you've seen already is that there are several spaces in between each section of this expression, as well as a comment at the end. So by using the ex modifier, we can allow comments with inside our regular expressions and spaces to make it easier to read. The last modifier that we'll take a look at in this module is the n modifier. The n modifier allows you to prevent captures of subexpressions unless they are named. On line 14, I have a string, abc 123, using the match operator again, and then the expression I'm using opens with, of course, with the n modifier with a subexpression question mark and then the n inside to indicate that we're going to be using that n modifier. Now, after the modifier is the first named capture, which creates a named capture of word, and captures all characters A through Z and then uses the plus sign quantifier to say more than one character. After that, there's the backslash s metacharacter with the plus sign quantifier to match spaces, and then there's another subexpression to match all of the digits with the backslash d metacharacter and the plus sign quantifier. Now, if I hadn't used the n modifier, what would happen is I would have two different captures. I would have one named capture of

word and then I would have a capture into a numbered capture with the digits. But because I'm using that modifier, I shouldn't get a capture group for the digits, so let's run line 14 and take a look at the results. So we got a match, now let's take a look inside the matches variable. And as you can see, we only have a named capture of word and then the capture group of zero which is always going to be there. We did not get a capture for the digits subexpression. That wraps up the section of this demo on modifiers. Let's now move on to creating objects from regular expressions.

Creating Objects from Named Captures

In a previous module, I demonstrated how you could get event sources from a Windows Event forwarding subscription. I showed you two different ways of obtaining that information. One was with using regular expression, and the other was by using the split and replace operators. But I didn't really talk about why regular expression is better. The main reason regular expression is better is because it gives you much more flexibility. The following example will demonstrate how you can use regular expression to capture the server name and domain name of the event sources by storing them inside named captures. Starting on line 20, I am populating a variable called $regex with an expression that spans multiple lines. The expression starts off on line 20 with the x and n modifier. The x modifier is what is allowing me to span this expression across multiple lines, and the n modifier will prevent any capture that isn't named, as demonstrated on line 21. Line 21 starts off with a subexpression with the word address in it followed by a colon and a backslash s metacharacter to match spaces. In this expression, I just wanted to group these together, but I didn't want to capture it, and that n modifier allows me to do that. On line 22 is our first named capture. The name of this capture will be ServerName, and it will capture any word characters one or more times. Outside of that first named capture I have a comment reminding me that it is the first named capture of this expression. On line 23, I have an escaped period or dot metacharacter. On line 24, I have another named capture with the name of domain. The expression it will use starts with the backslash w metacharacter and the plus sign quantifier, which is followed by a period, then a c, then an o, and then an m. Outside of that name character is another comment reminding me that this is the second named capture of the expression. That wraps up the rest of the expression. Now moving on to line 26, I am populating another variable called $eventSub. This will contain the Windows Event forwarding subscription information. To get that information, I'm using the command wecutil and then getting the subscription details for the app event subscription and then piping that out to a string. So I will go ahead and run lines 20 through 26 to populate those two variables. And then we'll take a look at the event subscription. So now when I run line 28, what should happen is we should get two named captures, one of server name and one of domain name.

Server name should have a value of GDC01 and domain name should have a value of globalmantics.com. So let's run line 28 and find out what is inside the matches variable. As you can see from the output, we do have a named capture of ServerName with the value of GDC01 as well as a named capture with the name Domain and the value of globalmantics.com. So the expression worked flawlessly. You can see how this gives you additional flexibility over using split and replace methods. You can even name your captures, and you can do it multiple ways, depending on what type of output and what format that output needs to be in. The last example that I have for you before we end this demo is how you can convert named captures to PS custom objects. On line 34, I am creating a regex object called $regexObj, and then I'm going to use the expression we used above inside the regex variable. On line 36, I am populating a variable called $keys, and what this keys variable will contain is all of the named captures with inside this regular expression, and to do that I'm using the regex object and then using the GetGroupNames method and then piping that to where-Object, and basically just matching anything that is a letter character that has at least two letters in it. That way I can avoid getting the named captures or numbered captures of anything else. I will go ahead and run lines 36 and 34. And then we'll take a look inside the keys variable, just to see what it contains. As you can see from the output, keys contains the names of the named capture groups inside of a regular expression. Now comes the fun part. I will go ahead and expand line 38, so we can take a look at all the code required to create PS custom objects from named captures. Line 38 starts off by using the Matches method from the regex object on the eventSub variable. The eventSub variable contains the string output of the event subscription. I then pipe the results of this before each object alias, the percent sign, and then create a $match variable with that input. Within the first for each object, I pipe the $keys variable to another for each object. The second for each object creates a function that has a Begin Process and End section. Inside the Begin section, I create an ordered hash table called hash, then inside the Process section, I use the add method of the hash table to add a key value pair. The $_ is the key which is populated by the $keys variable. This will either be server name or domain name in this example. The value of this key value pair is extracted from the $matches variable created on line 39. The End section outputs the hash table as a PS custom object. So now that we've walked through all of that code, let's run lines 38 through 43 and take a look at the results. As you can see from the output, we have an object with two properties, server name and domain. Server name has the value of GDC-01, and domain has the value of globalmantics.com. Granted, creating this object wasn't super easy, but think of the possibilities. You can now transform any regular expression capture into an object. That's it for this demo. Let's now summarize what was covered throughout this module. Throughout this module, you learned about named captures. You learned how to create named capture groups, as well as how to filter results by those named captures. You also learned how to create objects from named captures. In addition to that, you were introduced to the regex object inside of

PowerShell. You learned how to create those regex objects, as well as how to use some of the methods of those regex objects. This module also covered modifiers. You learned how to use modifiers to add comments and spaces to your expressions, how to enable single line mode and what that means, as well as how to prevent captures from non-named subexpressions.

Matching Positions with Lookarounds

Introduction to Lookarounds

As humans, it's very easy for us to recognize complex patterns. For example, say you were given a large number, and you were told to add commas to it. You'd start by counting every third digit, and write a comma, until you could no longer find three digits without a comma. To accomplish this in regular expression, you would use a lookaround. Lookarounds allow you to either look ahead, or look behind, at any given point in an expression. Lookarounds are unique in that they don't consume characters, rather, they assert whether a match is possible or not. In this first example, I am using a lookbehind to match the word Address followed by a colon, and one or more whitespaces. The neat thing about the lookbehind subexpression is that the characters used within it aren't actually captured in the match. Only the characters after the lookbehind will be captured, and in this case, it would be word characters listed one or more times. For this example, the lookbehind allows me to match the words GDC01 only if it's after Address, colon, space. This allows you to get very specific with your regular expressions. It's also one way that you can match by position, as you can see in the next example. I'm using both a lookbehind and a lookahead in the next example, where I have a string that says, "See Joshs course." I am using the lookbehind and the lookahead to match the position right between the h and the s. That way, I can make the word Josh possessive. Using lookaheads and lookbehinds allow you to craft very specific regular expressions. I previously mentioned the two types of lookarounds, lookahead, and lookbehind, however, there is a positive and negative variant of each. The table shown here lists each of them. Positive lookaheads, normally just called lookaheads, are successful if the regex can match to the right. The regular expression for this is a subexpression with a question mark and an equals sign within it. Negative lookaheads are the opposite. They only match if successful when the regex cannot match to the right. The regex for a negative lookahead is a subexpression with a question mark and an exclamation mark in it. Now on to lookbehinds. Positive lookbehinds are successful if they can match to the left of the expression. The syntax for a positive lookbehind is a subexpression followed by a question mark, a less than sign, and an equals sign. Negative lookbehinds, of course, are the opposite. They are successful only if they cannot match to the left. The syntax for it is a subexpression with a question mark followed by a less than symbol, and an exclamation mark. The logic of all of this is a little mind bending. So let's get started with some demos to see these in action.

Using Lookaheads

Over the course of the next few demos, we will be taking a look at lookarounds. Now as we just discussed, there are several types of lookarounds. The first one we'll be discussing is a positive lookahead. We will use a positive lookahead to add commas to digits. The next is a negative lookahead. We will be using negative lookaheads to validate IP address ranges. After that, we'll take a look at positive lookbehinds, and use them to capture event sources from a Windows event forwarding subscription. After that, we'll take a look at a negative lookbehind. We will use a negative lookbehind to prevent specific branch names from matching in an expression. At the very end of the demo, we will combine a lookahead and a lookbehind to obtain all the local administrator members, displayed from the output of a net local group command. As promised, starting on line three, I will use regular expression in a positive lookahead to add a comma to the number 1000. The objective with this expression is obviously to place a comma between the one, and the very first zero. I chose to use a positive lookahead in this example, as you can see by the subexpression with the question mark and the equals sign in it. Now right after that equals sign is a backslash d metacharacter, and it specifies three matches. After the backslash d metacharacter and the three in the square brackets, there is a dollar sign. This is an end of the line anchor. How this expression works is it evaluates after each position. So after the one, it evaluated that there are three digit characters, and then an end of the line anchor. So that was the position where it needed to match. The results of this is going to be the position right between the one and the zero. Remember that lookarounds do not consume characters. They just evaluate whether the regular expression was true or not. So with this positive lookahead, it looked ahead, to the right, for three digit characters, and then an end of the line anchor, which is found right after the one. And again, because lookarounds match positions, when I use the replace operator, I'm not actually going to replace any characters. I'm just going to replace the position with a comma. So let's go ahead and run this, and see what it looks like. As you can see, for the results, we get the number 1000 with a comma right after the one, as we expected. Let's now move on to negative lookaheads. In previous modules, we've taken a look at how we can validate IP address ranges with regular expression. The IP address range that we used was 192.168.20.0, with a subnet mask of /29. Now the last available IP address in that subnet should be .6. However, when I run the expression that I have on line nine, I will get a match. And as soon as I run this, I'm sure you immediately know why I did. The problem with the expression was that it matched the one inside of the number 11. To avoid this, we can use a negative lookahead, which is shown on line 11 at the very end of the expression. It starts off with a subexpression, the question mark, and then an exclamation mark. Right after the exclamation mark is a backslash d metacharacter, to match any digit character. The negative lookahead will look ahead, to the right, as soon as it matches the last digit in the last octet,

which in this case, is one. And if a number exists after the one, it will fail the expression, because it did find something. This will allow us to prevent matches with numbers that repeat themselves, like one, and two, and all the way through six. Now if it was seven, it wouldn't be an issue, because our character class right before that negative lookahead would have caught it. So when I run line 11, we shouldn't get any values returned to us, because the match should fail. So let's run line 11, and find out what happens. And as you can see from the lack of results, we did not get a match, and nothing was returned, as we expected. Let's now move on and take a look at positive lookbehinds.

Using Lookbehinds

Starting on line 17, I am populating a variable called $t with a here sting. Within the here string is a snippet of code that was extracted from the Windows event forwarding subscription. In this here string, I have added an additional event source, GDC02, that belongs to the globomantics domain. The goal of this expression is to extract out just the server names, GDC01 and GDC02. The expression starting on line 26 uses the regex object, and then uses the matches method from that object. The $t is the here string, which is the input from above. Then is the expression, the expression starts off with a positive lookbehind, which is indicated by the subexpression, with the question mark, the less than sign, and then the equals sign inside of it. Right after the equals sign, inside of the positive lookbehind, is the word Address, followed by a colon, then the backslash s metacharacter, and the plus sign quantifier. Outside of the positive lookbehind is the backslash w metacharacter, and a plus sign quantifier. This is the part of the expression that will actually consume characters, and return results to us. How this expression will work is it will try to match any word character, and as soon as it does, it will look behind that word character to see if it can match Address, colon, and then at least one whitespace. And because line 19 and line 22 are the only two lines in this here string that have Address, colon, and then a space, we should be able to capture and return to us GDC01 and GDC02. It will stop at the period, because that is not considered a word character. I will now run lines 17 through 26, and we should get returned to us the results of those two server names. And as you can see from the results, we do get GDC01 and GDC02. In the next example, we'll take a look at negative lookbehinds. Before I walk through the examples on line 32 and 34, allow me to provide some context. I'm about to write some PowerShell automation for a TeamCity build. I need to parse the output of the build, and get the release name and number, but only if it isn't the master branch. The automation I'm writing should only be applied to the develop and other feature branches. On line 32 I have an expression that will match the release name and version off a branch called develop. The expression starts off with a negative lookbehind, which is indicated by the subexpression, followed by a question mark, a less than sign, and an

exclamation mark. After the exclamation mark is the word master, followed by two backslashes. One backslash is escaping the other, because the backslash is a metacharacter. Now outside the negative lookbehind is the word improvement, and then an escaped hyphen, followed by 1.2.0. The dots are, of course, escaped as well. Now when I run line 32, I should get a match, and I should get the results returned to me of improvement-1.2.0, because the branch that this release is on is the develop branch, not the master. And that negative lookbehind is looking to the left, after it matches improvement-1.2.0, to see if the branch is master, and if it is, it will fail the expression, resulting in no matched values. So let's run line 32, and see what's returned. As you can see from the results, we get returned to us improvement-1.2.0, which is the release and the release version. And just to prove that the negative lookbehind is working, I will run line 34, which I've just updated the branch name to be master. So when I run this, I shouldn't get any values returned to us. And as you can see, when I ran line 35, I didn't get any results returned to us, which is exactly what we wanted to happen. In the very last example in this module, I will demonstrate how to use both a lookahead, and a lookbehind, in the same expression.

Combining Lookaheads and Lookbehinds

In this last example, we will be taking a look at the output of the net localgroup commandlet to get all of the administrators group memberships. I will go ahead and run line 40 to populate the $LocalAdmins variable, and then we'll take a look inside to see what the results look like. This is the raw output from the net localgroup commandlet. As you can see, it gives us very important information. It tells us who the members are, but, it's not very clean. What if we just want to know the names of the groups and users, and not the extra stuff that it provides? Well, one way that you could do that is, of course, with a regular expression. On line 42, I'm using the regex object, and then using the matches method off of that object. I am then putting the localadmins variable in as the input string, next up is the expression. At the very beginning of this expression I am using a modifier, I am putting it in single line mode. After the single line mode modifier, I have a lookbehind. I am using this lookbehind to match the hyphen at the very end of that string of hyphens that you see in the output below, and then using it to match the return character one or more times, and a newline character. This should match the position right before the a character, in the word administrator. Outside of the lookbehind, I have the period star metacharacters, which I will use to match all of the memberships of the administrators group. After the period star, I have a lookahead. This lookahead will look for the word The, and match position right before the T, which should allow me to capture just the users and groups inside the administrators group. I am then extracting out only the value returned from those matches, and then trimming those values to remove any additional whitespaces. So before I run

line 42 and see if the expression works, I will clear the screen, so we can get only the results of the values to return back to us. Now that that's done, let's go ahead and run line 42. And as you can see, the only thing returned to us was the memberships of the administrators group. Lookaheads and lookbehinds can get a little confusing, but they are extremely powerful. They allow you to do things that would otherwise be impossible. That's it for this demo, let's now review and summarize everything we covered throughout the module. Throughout this module, you learned about lookarounds. You learned about positive lookaheads, and used them to add commas to digits. You also learned about negative lookaheads. You used them to validate IP address ranges. After that, you took a look at and learned about positive lookbehinds. You used positive lookbehinds to capture event sources from a Windows event forwarding subscription. After that, you took a look at negative lookbehinds, to prevent specific branch names from matching in an expression. At the very end of the module, you combined a lookahead and a lookbehind to extract a local administrator names from command output.