Lecture 8: Text and Document Management

Administrative work is simpler with PowerShell's ability to work with objects. However, every competent administrator must learn to work with text. The latter is an unstructured text stream called a string. PowerShell has many types of objects, such as arrays and hash tables, but when text is to be written to a file or displayed on the console, all objects must be converted to strings. Text streams are made up of printable characters such as letters and numbers, and non-printable characters, such as space and new line

A Attributes of Strings: Null Empty Literal

1 Null or Empty

There are "null" strings and "empty" strings. The reason for the two types of strings is to be able to differentiate between a string that has not been initiated from one that has been initiated. A null string is a special string which has not been defined and does not contain anything. For example $a = $null; the $null value is a special automatic variable for undefined objects. An empty string is a defined object with a length of "0" length. For example, $a = ""; this code declares a string object of zero length. PowerShell treats both situations as the same. In Figure 2, we create a variable $PS and set its value to "PowerShell". Using the length property we can see there are 10 characters in the string. $emptystr is a declared string object and the length property shows zero length. Since $nullstr is not a declared object, you would think that the length property would not work, but it does work because PowerShell treats both situations as the same. It is important when testing if a string is null or empty to use the static method IsNullOrEmpty. We will learn how to use this method when we discuss the IF ELSE clause.

```
Windows PowerShell
Windows PowerShell
Copyright (C) 2013 Microsoft Corporation. All rights reserved.

PS C:\Users\dhr> $PS = "PowerShell"
PS C:\Users\dhr> $PS.length
10
PS C:\Users\dhr> $emptystr = ""
PS C:\Users\dhr> $emptystr.length
0
PS C:\Users\dhr> $nullstr = $null
PS C:\Users\dhr> $nullstr.length
0
PS C:\Users\dhr>
```

Figure 1: Comparison $null and empty string

2 Literal Strings

A literal string is defined using double or single quotes. In other words, you are declaring what the string should be. For example, $text = "PowerShell is the Harry Potter of system administration". To see the difference in between single and double quotes. Type the following:

$HP = "Harry Potter"
$text = "PowerShell is the $HP of system administration"
$text1 = 'PowerShell is the $HP of system administration'

```
PS C:\Windows\system32> $text
PowerShell is the Harry Potter of system administration

PS C:\Windows\system32> $text1
PowerShell is the $HP of system administration
```

Figure 2: Difference between double and single quotes

Double quotes create a literal string and tell PowerShell to replace any variables and special characters. Single quotes create a literal text string where the variables and special characters are not replaced and displayed as text.

3 Common String Conversion Mistake

One of the most common types of mistakes is type conversion, such as forgetting to cast or declare variables as a string. For example, when a string and another data types are combined using the "+" operator, or the pipeline, the other datatype is automatically converted to a string using the object's ToString() method.

Type: $ver = 4 # this is an integer
Type: $text = "Windows PowerShell " #notice the space before the double quotes
Type: Write-host "$text + $ver"

PowerShell implicitly changes the datatype of $ver from number to string, because the first variable was a string. However, if we change the order of $ver and $text we will get an error, because PowerShell

```
PS C:\Windows\system32> $str
Windows PowerShell 4

PS C:\Windows\system32> $str = $ver + $text
Cannot convert value "Windows PowerShell" to type "System.Int32". Error: "Input string was not in a correct format."
At line:1 char:1
+ $str = $ver + $text
+ ~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

Figure 3:Forgetting to Declare Variable as a String

can't convert the string "Windows PowerShell" to a number. This error could be avoided by remembering to type **$ver = "4"** or **[string] $ver = 4** to explicitly create a string instead of a number.

B Working with Strings

1 Comparing Strings

Type: **$text = "PowerShell is the Harry Potter of system administration."**
Type: **$text1 = "POWERSHELL IS THE HARRY POTTER OF SYSTEM ADMINISTRATION."**
Type: **$text.CompareTo($text1)**

When comparing strings you are comparing the total of the Unicode values. Since uppercase letters have a lower value than lowercase, the output, "-1" indicating that string $text2 is less in value than string $text. A "1" would indicate that $text2 is greater than $text and a "0" would indicate a perfect match.

Alternative comparison methods are:

- the Equals method, which always also does a case-sensitive comparison, and provides a boolean value. Type: **$text.Equals($text1)**
- -match parameter which does a case insensitive comparison, but adding the –cmatch will do a case sensitive comparison like equals and CompareTo.
    - Type: $text –match $text1 #notice the result is true because match uses a case insensitive comparison by default.
    - Type: $text –cmatch $text1 #result is same as with Equals because if is comparing case.

```
PS C:\Windows\system32> $text.CompareTo($text1)
-1

PS C:\Windows\system32> $text.Equals($text1)
False

PS C:\Windows\system32> $text -match $text1
True

PS C:\Windows\system32> $text -cmatch $text1
False
```

Figure 4: Comparing Strings

You can also use the Contains method to compare if a character or group of characters exists in the string. The methods StartsWith and EndsWith are also very useful and like the CompareTo method they are all case sensitive by default.

Type**: $text.Contains("SYS")**

The Contains method returns True if the target text can be found and False if the target text cannot.

.2 Combining Strings

Combining strings is one of the most common operations in administration (technically called concatenation). A typical use is when you want to output some information to the console by combining some literal strings with variable values. To combine strings you can use the "+" or the join operators to combine multiple strings into one string

JoiningString1.ps1 | Untitled2.ps1

```
1 $text = "Windows PowerShell"
2 $ver = "4.0"
3 $text1 = "the Harry Potter of system administration"
4 $string = "$text" + " " + "$ver" + " " + "$text1"
5 Write-host $string
6
```

```
PS C:\Windows\system32> C:\Users\dhr\Documents\Win213_Revised\wk8\JoiningString1.ps1
Windows PowerShell 4.0 the Harry Potter of system administration
```

Figure 5: Combining Strings with the "+" Operator

The second method uses the join operator. The strings to be joined are to the left of the join operator and the delimiter, in this case a space, is to the right of the operator. The strings are appended in the order that they

appear, separated by the delimiter.

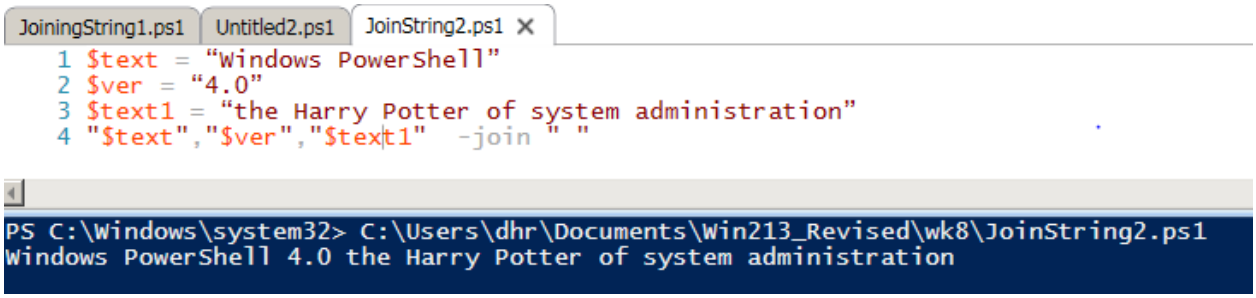
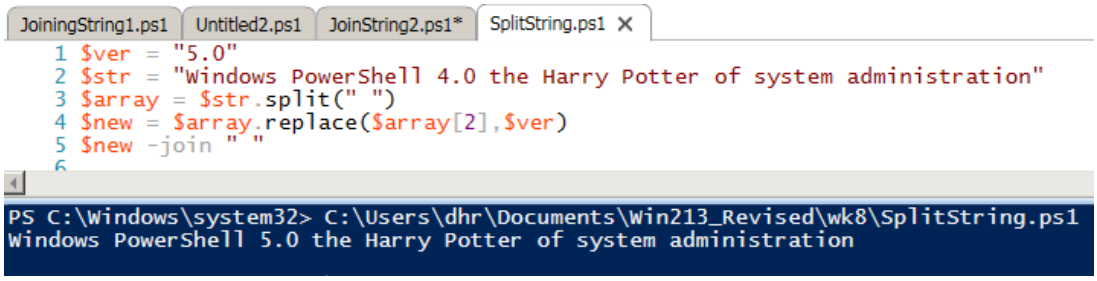


Figure 6: Using the Join Operator to combine strings

3 Splitting Strings

Sometimes strings need to be split apart and converted to an array. Splitting strings into an array makes it easier to substitute strings. The split method will split a string based on the defined delimiter.



4

Figure 7: Splitting strings into array for string substitution

Changing the Case of Strings and Title Case

Type: **$text.ToUpper()**
Type: **$text**

Notice that the variable $text has not been overridden, only modified by the ToUpper method. To overwrite the variable. If we want to convert a string to upper and lower case, where the beginning of a sentence or word is capitalized (called Title case) there are two ways we can do it. First, we can split the string into an array and use the ToUpper method to capitalize the first letter of each word, appending the rest of the word to the substring. This new array will have to be saved in a variable which needs to be casted as a string so that PowerShell rebuilds the array elements as a single string

Type: **[string] $title = foreach($word in $text.Split()){$word.Substring(0,1).ToUpper()+$word.Substring(1)}**

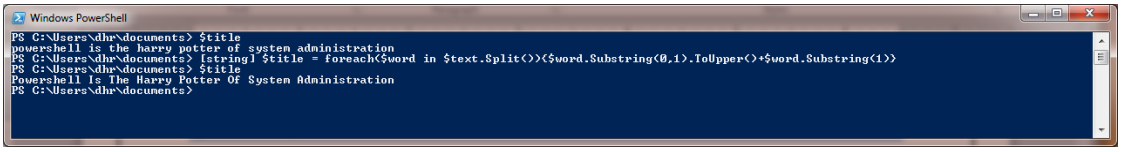


Figure 8: Changing to Title Case using Split and Substring Methods

Or, we will need to use the ToTitleCase method of the TextInfo class which is part of System.Globalization. We use the Get-Culture cmdlet (the string object does not have a ToTitleCase method).

Type: **(Get-Culture).TextInfo.ToTitleCase**
Type: **$text = (Get-Culture).TextInfo.ToTitleCase($text)**

Figure 9: Changing to Title Case using ToTitleCase Method of .NET

5 Extracting a Portion of a String

Sometimes you will need to extract a portion of a string from within another string. For this you use the substring method. This method has two parameters, starting position in the string and the length of the extraction.

In our example I could easily count the starting position, but what if I was not able to, how can I determine the beginning of the substring. This is the purpose of the IndexOf method. It gives the character position of the substring starting from the beginning of the string.

Type: **$text.IndexOf("harry")**
Type: **$d = $text.Substring(18,12)**

Here we are saying start extracting characters at the 18 character in the string (remember to begin counting from 0) and continue for 12 characters and place that substring in a variable $d. If you do not use the second parameter, which is optional, the substring method will extract all characters, from the starting position to the end of the string.

Figure 10: Using the SubString Method

6 String Substitutions

To replace a portion of a string, you would use the Replace method. The latter uses two parameters, the character(s) to be replaced, and the new character(s).

Type: $text.Replace("h","l")

Figure 11: Using the Replace Method

The replace method is case sensitive and would not find an "H", if that was the letter we were looking to replace. Also, notice what happened. It changed harry to larry, which is what we intended, but it also changed the "the" to "tle", and "powershell" to "powerslell" which was unintended. The Replace method will change all occurrences of the letter in the string. If we only wanted to change one occurrence, we would first use the IndexOf to find its character position in the string, and then use the Replace method to change the text at that position.

Type: **$text.Replace("powerslell is tle la","powershell is the ha")**

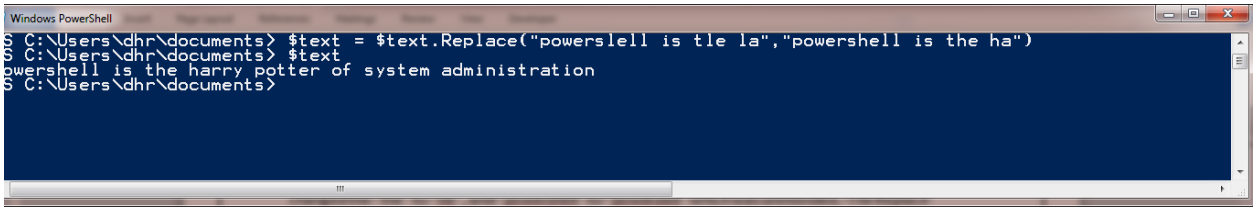


Figure 12: Replacing substrings with the Replace Method

C Here-String

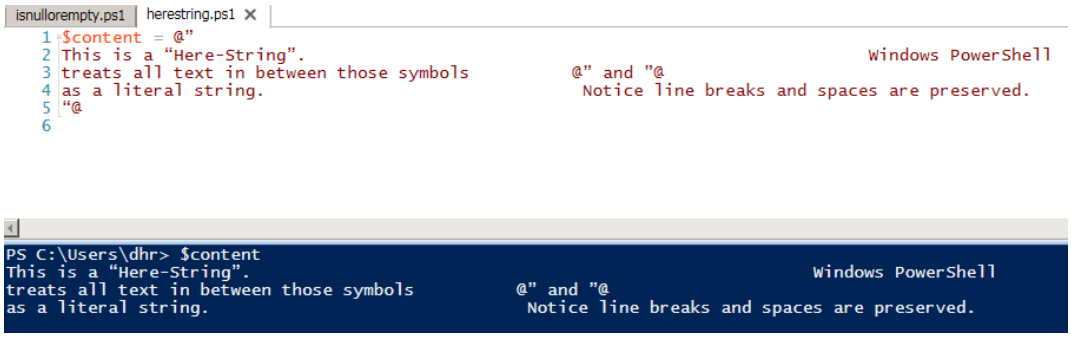


Figure 13: Here-string

You create a Here-string by starting it with an @ symbol mark, followed by a double quote and then a new line. You end a Here-string by using a double quotation mark followed by the @ symbol which also must be on its own line.

D Document Management

Files and directories are stored on the file system drive in a hierarchical manner using paths to identify the location of the file or folder. PowerShell has cmdlets designed to work with files and folders. Paths are strings, but PowerShell is object-oriented so if we convert a string to a System.IO.FileInfo object we can use special properties to extract path information rather than using the substring method.

1 File Properties and Resolve-Path

JoiningString1.ps1 | Untitled2.ps1 | JoinString2.ps1* | SplitString.ps1* | Untitled6.ps1 | pathnames.ps1

```
1  # use resolve-path to fill in the wildcards of a path and store in variable
2  $files = "$home\documents\win213??\*\*" | Resolve-path
3
4  #select only the last path and store in variable
5  $one = $files | Select-object -last 1
6  (Get-childitem $one).fullname              #full absolute path to file
7  (Get-Childitem $one).DirectoryName         #full absolute path to folder
8  (Get-Childitem $one).Directory.Name        #name of folder
9  (Get-childitem $one).name                  #name of file and extension
10 (Get-childitem $one).basename              #file name only
11 (Get-childitem $one).extension             #extension only
12 (Get-childitem $one).PSdrive.name          #drive name
```

Figure 14: System.IO.FileInfo Properties

For example:

In the above script, we use the Resolve-Path cmdlet to complete the path identified by the string on the left of the pipe using wildcards. This string wants PowerShell to resolve paths recursively for all Win213?? Directories (there is only one on the system). All of the file paths are stored in the variable files. Then we pipe the output of the files variable to select-object to select only the last path and store the result in a variable called one. Then we convert the string path to a FileInfo object so we can use the specialized properties to extract path information:

- Fullname – provides a complete absolute path to the file
- DirectoryName –provides the full absolute path to the folder
- Directory.Name –provides the folder name where the file resides
- Name –provides the name of the file with the file extension
- BaseName –provides only the file name
- Extension – provides only the file extension
- PSdrive.Name –provides only the system drive name

Let's suppose you wanted to create a new path of C:\Win213_welcome.ps1 from the original path. You can use properties and is Figure 18 and save the result to a variable and use the Join-Path cmdlet to create the new path.

```
PS C:\Users\dhr\Documents> $Drive = (gci $one).PSDrive.Name

PS C:\Users\dhr\Documents> $File = (gci $one).Name

PS C:\Users\dhr\Documents> Join-Path $Drive -ChildPath $file
C\Win213_Welcome.ps1
```

Figure 15: Using Properties to Modify a Path

a Alternate Method Using the Split Method

Another method to modify paths is to first convert them to an array using the split method. For example, suppose we had a path like the following: **c:\users\<loginName>\documents\win213\wk6\Lab6_Webpage.html**. And, we wanted to create a new path combining the drive "c:" with the file name "Lab6_Webpage.html" to create a new path c:\Lab6_Webpage.html. How can we do this?

Type: **$Path = "c:\users\<loginname>\documents\win213\wk6\Lab6_Webpage.html**
Type: **$array = $Path.Split("\")**

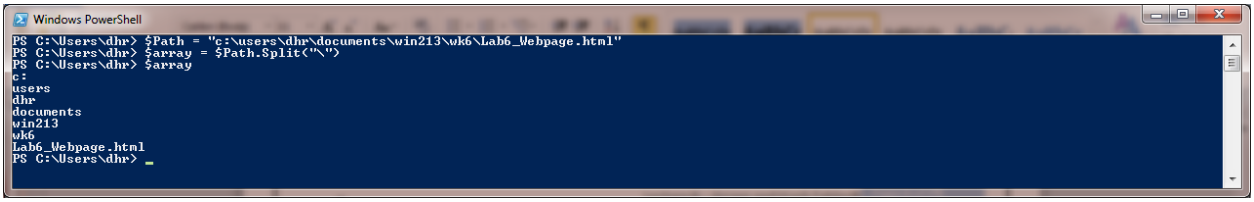


Figure 16: Splitting Paths Using the Split Method

With each element separated on a new line it is easy to create a new path.

Type: **$file = $array[-1]**
Type: **$Drive = $array[0]**
Type: **Join-Path –path $Drive –childpath $file**

Figure 17: Creating a New Path with Join-Path

2 Inserting a File Tag

A common administrative function is to rename a batch of files by inserting some file tag to identify the group of files. Suppose we had a 6 files named: Budget1.2012.xlsx, Budget2.2012.xlsx, up to Budget6.2012.xlsx, How can we rename the group of files Budget1.Q1.2012.xlsx? Open up PowerShell ISE. Create a for loop to create 6 files called Budget1.2012.xlsx to Budget6.2012.xlsx. Make sure you are in the documents folder.

Type: **For($i=1;$i –le 6; $i++){New-Item –path . –Itemtype file –Name "Budget$i.2012.xlsx}**

Type **$files = gci *.2012***

Run the code to generate the files.

The files returned are an array. We need to convert them to a string so we can use the string methods.

Type: **[string] $strFiles = gci *.2012***

Type: **$strFiles**

Notice the difference between to a structured order, such as an array to an unstructured order of text, like the string.

Type: **$strpath = $strFiles.Split(" ")**

Type: **$strpath.Gettype()**

Using the split method we can separate the file paths into an array of strings, separated by a new line character. Now we can use the pipeline Foreach-Object and use the Insert command to change the file name. The insert command takes two parameters, the first is the insertion point, and the second is the letter to be inserted. Notice you can only insert one character at a time. But you can chain the commands together using dot notation to add multiple characters.

Type**: $strpath | Foreach-Object{$_.Insert(31,"Q").Insert(32,"1").Insert(33,".")}**

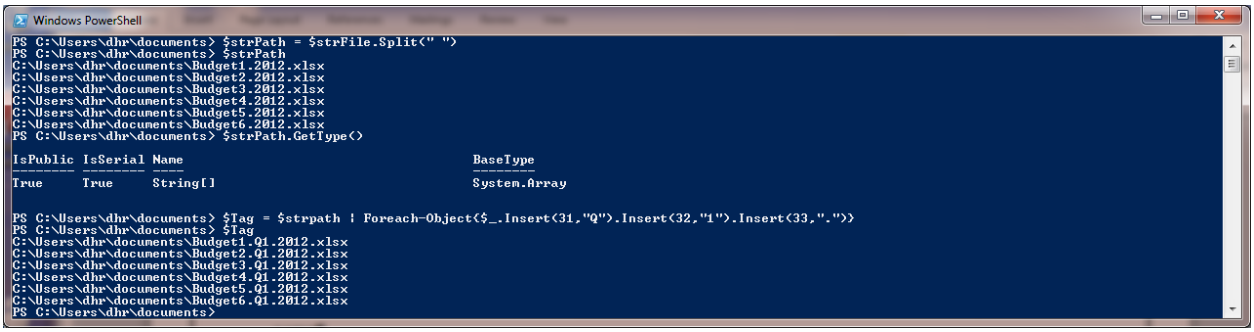


Figure 18: Using the Insert Method

b. Converting to a Character Array

When working with strings, there are times when it is easier to convert the string to an array which gives you an index to reference each element. Suppose you had a list of part numbers, such as “129BX123”, which represents the month of manufacture, 12, the plant number, 9, inventory lot number, BX123 the item number. Restructuring of the inventory requires changing BX to BY. Converting the string to a character array of individual characters will make the change easier.

Type: **$PartID = 129BX123**
Type: **$PartArray = $PartID.ToCharArray()**

Type: $**PartArray**. You can see the each character is an element of an array. We can now count from 0 and change the element.

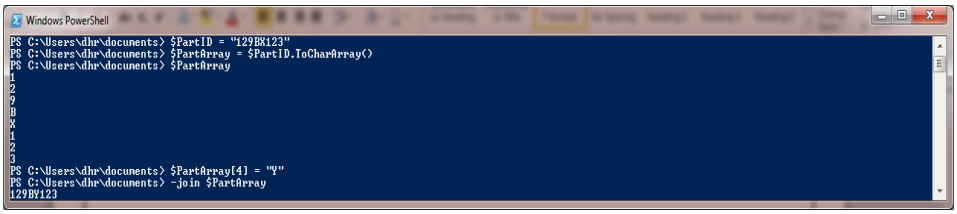


Figure 169: Converting to a Character Array

Type**:$PartArray[4] = “Y”**
Type: **-join $PartArray** to convert the array elements back into a string.

c Regular Expressions

Everything we have discussed so far, in finding and replacing text can also be done using Regular Expressions. Regular expressions are very powerful and very accurate in describing text patterns. Administrators often use regular expressions to validate user input such as, date and time, IP address,

postal code or email address. There are excellent resources on how to use regular expressions on the web such as RegExLib, RegExTester and RegBuddy. RegExLib is a web site with free information and examples of regular expressions, RegExTester is an online regular expression tester and RegBuddy is a GUI tool to create and test regular expressions.

Using regular expressions in PowerShell requires one of two methods: The -match operator, or the Select-String cmdlet (other commands, such as the Switch construct, also accept regular expressions). PowerShell will return True or False if there's a match. In using the match operator the text to be texted is to the left and the regex expression is to the right of the operator.

Regular expression patterns basically contain three elements: **anchors, placeholders, and quantifiers** which are used to test a string. Suppose we wanted to confirm that the partID "129BY123" was entered correctly, we would use a regular expression as follows:

Type: **$pattern = '^\d{3}[A-Z]+\d{3}$'**

This expression contains all of the above elements:

- "^"is an anchor to match the pattern at the beginning of the string.
- \d is a placeholder. It represents a digit 0-9.
- {3} is a quantifier. It expects the placeholder to occur 3 times
- [A-Z} is a placeholder for capitalized letters A-Z
- "+" is a qualifer and matches repeating incidences of previous pattern.
- "$" is an anchor to ensure that the previous pattern of digits -0-9 is at the end of the string

By default, regular expression searches are case insensitive. To include case, we use the "C" modifier of match operator. When a match is found an automatic variable called $matches stores the result.

Type: $matches

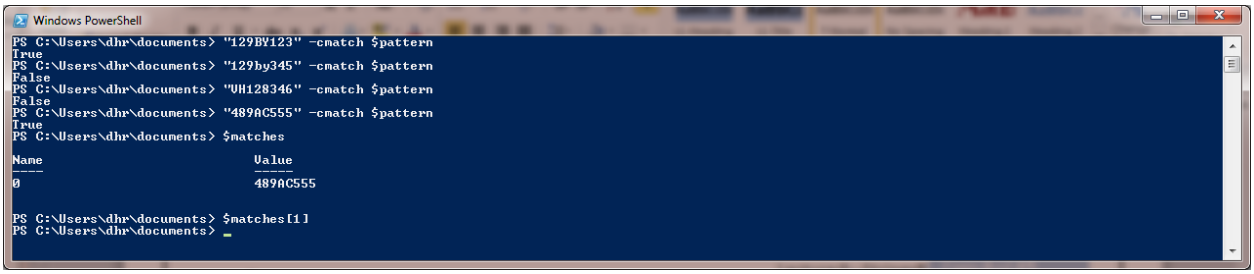


Figure 20: Regular Expressions and $Matches Variable

We can create groups of the string. Suppose we wanted to separate the numbered portion from the letter portion. The first set of 3 digits could refer to the bin number, the letters, refer to the inventory lot category and the last 3 digits are the item number in the bin.

Type: **$pattern = '^(\d{3})([A-Z]+)(\d{3})$'**
Type: **$matches**

Figure 21: Using Regular Expressions to Find Matches

The matches variable now holds all of the grouped strings. Grouped in this way, the strings can be referenced using the index number and modified using the –replace operator.

3 Converting to CVS

CVS files are "comma delimited files" which are text files that can be used to separate information in the file, such as name, address, etc.. A CSV file begins with a header row which identifies the name of each column. Then each row after the header file is one record with the corresponding information for each column. We will create a short CSV file using a Here-String. Type the following:

$cvs = @"
"Name","Email"
"Jerry Sienfeld","jsienfeld@cbs.com"
"Danny Roy","danny.roy@senecacollege.ca"
"Ataur Rahman","ataur.rahman@senecacollege.ca"
"@

After creating the content of the CSV file, we need to create a file:
Type: $csv > c:\users\<longinname>\mycsv.csv

Then we need to load the file into memory and store the contents to a variable so we can use it

Type: $mycsv = Import-CSV "c:\users\<loginName>\mycsv.csv"

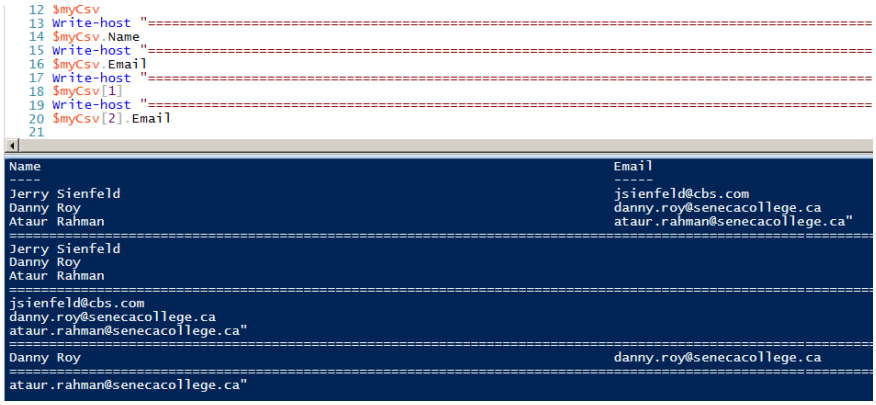


Figure 22: Accessing a CSV file

CVS files are the most commonly used files in network administration because they work on all platforms. It is easy to access the information in the CSV file by using array and dot notation. For example, to access the all of the names in the file, type $myCsv.Name, to access all of the emails, type $myCsv.Email. Individual rows of information such as the 2nd item in the file, $myCsv[1]. To access only the name of the second item, type $myCsv[1].name, etc. The important point is that the CSV file must be loaded into memory using Import-CSV before PowerShell can work with it.

4 Converting to HTML

PowerShell makes it easy to post information on the web using the ConvertTo-HTML cmdlet. Managers like using the web because the information easily accessible using a web browser, and is a convenient method to update information. You only update in one place and everyone can see the new information. For example, to post the CSV file to the web

Type: $myCsv = Import-CSV " c:\users\<loginName>\mycsv.csv"
Type: $myCsv | ConvertTo-HTML | Set-Content "c:\users\<loginname>\mycsv.html"
Type: Invoke-item "c:\users\<loginname>\mycsv.html" # like double-clicking on an application

This creates a web site, but it is not very pretty. ConvertTo-HTML takes 3 parameters, title, head and body. We can make the table more interesting by creating a style file using a Here-String and passing the content to the head parameter

```
$ToWeb = @"
<style type=text/css>
Table{margin:auto;width:50%;border-width:4px;border-style:solid;background-color:yellow}

TH{height:50px;color:blue}
TR{height:40px; text-align:center}

</style>
"@
$body = "<center><H1>My CSV List</H1></center>"
```

Figure 173: Creating a Style Sheet

Your output should be similar to the screen shot to the left.

My CSV List

Name	Email
Jerry Sienfeld	jsienfeld@cbs.com
Danny Roy	danny.roy@senecacollege.ca
Ataur Rahman	ataur.rahman@senecacollege.ca"

Figure 24: Using ConvertTo-HTML parameters to pretty web output

We've Learned

1. Strings are unstructured text streams and are commonly manipulated by administrators, to prepare reports or modify paths. Strings in PowerShell have properties and methods. Strings typically have three attributes, empty null and literal.

2. Working with strings involves proficiency comparing strings and searching for substrings and extracting a portion of a string, changing the case of strings, splitting strings into an array, and joining strings.

3. Text strings can be converted to arrays using the split or ToCharArray methods to make substitution easier. The Join operator is used to combine multiple strings into one string, and set the delimiter.

4. Regular expressions can be used for the searching, replacing and formatting of string data. PowerShell has good regular expression support for complex pattern matching using anchors, placeholders and qualifiers.

5. Document management is working with strings and paths. PowerShell provides powerful cmdlets such as Resolve-Path and Join-path to make working with paths easier. Paths can be modified using the build-in properties of the System.IO.FileInfo object, or by splitting it into an array and using array notation.

6. PowerShell provides excellent tools to convert text information into CSV and HTML files.