Start at the beginning.  Powershell can be run via:

**start->run->powershell**

Say hello to your new blue and white friend.  My hope is to build from post to post and teach what I learn day-to-day as I attempt to migrate from perl scripts to powershell.

Powershell retains the commands that are available in most shells like csh, bash, korn, or DOS.  In an effort to attract both \*Nix and Windows users powershell is chock full of aliases that will be familiar to both flavors of the IT world.  Commands like **dir** and **ls** produce the same results, but in reality they are just aliases to  a cmdlet called**Get-ChildItem**.  Try typing all three commands in powershell, and you will see that they produce the same results.  In order to see a list of all the available aliases try:

**get-Alias**

This is a great place to start and look at some of the more common commands, but there’s an even more important resource that you need to become familiar with immediately: **get-help**, **man**, or just **help**(mind you help is a function that is equal to get-help, while man is an alias to help).  Let’s look up the help page for get-alias:

**help get-Alias**

Using this we can see everything we need to know about how to look up, and get information about configured aliases in your environment:

**get-Alias dir**

**get-Alias -definition get-ChildItem**

**get-Alias -definition get\***

You’ll notice that one lesson Microsoft learned is to provide proper inline documentation the way \*Nix and Perl do.  This beginning should provide as a confidence booster that they may have done some things right.  I can tell you for certain that there are a lot more surprises in store for you, but we’ll save that for another day.  At least you now have a few resources you can use to start learning the ins and outs of Powershell.

I have found that a great place to start learning a scripting language is by writing scripts to solve word problems.  File input and output operations, text manipulation, variables, arrays, and hashes all need to be understood in order to accomplish a goal like, “find all the 5-letter words in the English language that are palindromes”.  Perhaps the only reason I have found that this is a good place to start is because it is how I started learning Perl.  Regardless, that is where we are going to start today.

The first thing a good puzzle scriptor needs is a decent dictionary file.  I have one that I’ve used for a long time that has been compiled from a few public and private lists.  Sun has one available [here](http://java.sun.com/docs/books/tutorial/collections/interfaces/examples/dictionary.txt).  For this example we’re going to look at ways to open a txt file for reading, read the contents of a file, use a method to go through each line in the file one at a time, and output the results to the screen.

Copy the dictionary file into your working directory or cd to the directory where your dictionary is saved.  get-Content is the method used to read files.  If you look at the aliases available on the definition of get-Alias (see [Part 1](http://powertoe.wordpress.com/2009/11/10/powershell-part-1/)) you will see some familiar friends: cat, gc, and type.  Try the following (feel free to replace get-Content with your favorite alias):

**get-Content dictionary.txt**

Before we begin looping through each line of the file we should probably learn the method of writing output to the screen.  **Write-Output** is the cmdlet that will accomplish this.  It too has some familiar aliases: **write** and**echo**.  Try the following:

**Write-Output blah blah blah Write-Output "blah blah blah"**

When you look at the help page for **Write-Output** you see that the cmdlet has two functions.  The first is to pass the parameters to the next command in the pipeline.  The second is to output to the console if it is the last command in the pipeline.  This brings us to an extremely important concept within powershell that truly makes it a powerful and simple scripting language: **the pipeline**.

In DOS or \*Nix you may have had to pipe a command at one time or another.  For example a long directory listing would require dir |more in order to stop at every page.  Powershell uses the pipeline to pass objects or collections of objects from one command to the next.

Using this pipeline concept we can take the contents of the get-content cmdlet and pass it straight into a foreach loop so that we can iterate over each object in the collection:

**Get-Content dictionary.txt | foreach {Write-Output $\_}**

Perl developers will immediately notice that Microsoft has stolen one of the greatest features within perl.  The special variable **$\_** has an almost identical purpose in powershell.  In powershell the **$\_** variable represents the current object when using specific types of loops or blocks of code.  The following table shows a list of all of the special variables available in powershell.  It comes from an [article that digests and discusses variables in great detail](http://www.computerperformance.co.uk/powershell/powershell_variables.htm):

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| $\_ | The current pipeline object; used in script blocks, filters, the process clause of functions, where-object, foreach-object and switch |
| $^ | contains the first token of the last line input into the shell |
| $$ | contains the last token of last line input into the shell |
| $? | Contains the success/fail status of the last statement |
| $Args | Used in creating functions that require parameters |
| $Error | If an error occurred, the object is saved in the $error PowerShell variable |
| $foreach | Refers to the enumerator in a foreach loop. |
| $HOME | The user’s home directory; set to %HOMEDRIVE%\%HOMEPATH% |
| $Input | Input piped to a function or code block |
| $Match | A hash table consisting of items found by the –match operator. |
| $MyInvocation | Information about the currently script or command-line |
| $Host | Information about the currently executing host |
| $LastExitCode | The exit code of the last native application to run |
| $true | Boolean TRUE |
| $false | Boolean FALSE |
| $null | A null object |
| $OFS | Output Field Separator, used when converting an array to a string. By default, this is set to the space character. |
| $ShellID | The identifier for the shell.  This value is used by the shell to determine the ExecutionPolicy and what profiles are run at startup. |
| $StackTrace | contains detailed stack trace information about the last error |

In addition to using some great features from Perl, Microsoft has also kept some of the best features from vbscript/c#.  For example Powershell uses dot-notation to further explore the properties of objects stored in variables.  This can best be seen with the following line of code:

**Get-Content dictionary.txt | foreach {Write-Output $\_.length}**

In the above example we are getting the length of the string that is currently in the **$\_** special variable.  Since the object is currently a string the available properties of the object come straight from the .Net string object.  You can use any of the [string properties or methods available within .Net](http://msdn.microsoft.com/en-us/library/system.string_members.aspx) straight from Powershell:

**Get-Content dictionary.txt | foreach {Write-Output $\_.toupper()}**

**Get-Content dictionary.txt |foreach {Write-Output $\_.chars(0)**

One final note.  We’ve been using Write-Output to display data and pass it through the pipeline, but this is optional.  Write-Output is implied.  A variable or a string on a line by itself is really all you need to get the same effect as Write-Output.

**Get-Content dictionary.txt | foreach {$\_.length}**

**"Hello World"**

We’ve hit a lot of topics in this post that show some of the power behind Powershell.  The ability to pass objects via the pipeline, special variables, and the interoperability between Powershell and .Net make this simple scripting language a very complete and robust programming language as well.  In future posts you will begin to see that Powershell has a certain flexibility that allows users to write scripts their own way.  The perl motto, “There’s more than one way to do it” holds true with Powershell.

Time to learn some more basics.  In our [last tutorial](http://powertoe.wordpress.com/2009/11/16/powershell-part-2-opening-files-writing-to-the-screen-the-pipeline-and-dot-values/) we looked at some of the special variables given to us within Powershell, but now it’s time to discuss how user variables work.  Let’s start off simply.

**$var1 = 'blahblahblah'**

**$var1**

By using the $ before the word var1 we’ve created a variable.  Using the equal sign we are able to set this variable to the string ‘blahblahblahblah’.  Powershell is a very dynamic language in that it allows you to create and assign variables without casting them to a data type.  This can be a double-edged sword so it’s a good idea to understand how this can cause problems.  Take the following for example:

**$var1 = 5**

**$var2 = "blah"**

**$var1 + $var2**

Here we are trying to use a number (it’s a number because there are no quotes) and a string (it’s a string because it has quotes) with the + operator.  If they were both strings this would concatenate them.  If they were both Int32 they would add them together.  Because they are different we get an error that “blah” cannot be converted to Int32.  Powershell is extremely flexible with its data types.  Since the .net implementation of the Int32 class and the string class have Converto() functions the script will attempt to automatically convert the variable to the type expected by the operation.  In the above example the first variable is an Int32 so it assumes you are trying to add another number to this one.  If we reverse this and start with the string and use the + sign the compiler sees the first variable as a string and expects to concatenate.  Since an Int32 can easily be converted to a string reversing the operation works without failure:

**$var1 = 5**

**$var2 = "blah"**

**$var2 + $var1**

You can also call a conversion through the .Net methods available to the data type.  For example returning to our original problem code you can successfully do the following to ensure that the Int32 is interpreted as a string:

**$var1 = 5**

**$var2 = "blah"**

**$var1.tostring() + $var2**

Another way to get around this problem is to explicitly cast the variable to what you want it to be.  This is done with [] brackets before the variable:

**[string]$var1 = 5**

**$var2 = "blah"**

**$var1 + $var2**

Before we end this portion of the tutorial it’s important to discuss one more item, interpolation.  Powershell uses the same methods for interpolation that Perl uses.  If it’s in a double quote it will be interpolated.  If it’s in a single quote it is a string literal.  In both cases it is a string, but it’s how the string handles variables inside its quotes that makes them different.  The following illustrates this:

**$var1 = 'blahblahblah'**

**"This is what my variable is set to: $var1"**

**'This will not work: $var1'**

We have now learned the basics of what Perl calls a scalar variable.  In the next tutorial we will discuss arrays.

It’s easy to see how you can apply a loop to an array to iterate through each element of the array in order if only there was a way to test for how many elements are in the array.  We can do this by using a member of the array object called Length.  The thing to note here is that you want to initialize your index variable with 0 since that is the first element in your array, but you want to test that your index is less than the array length since the last element of the array will be one less than the number of elements in the array.   Here is the process in practice:

**$array = @("test1", "test2", "test3")**

**for ($i=0; $i -lt $array.length; $i++) {**

**$array[$i]**

**}**

An even faster way to do a for loop is by using a special foreach loop.  The foreach loop will set a variable to each item in a list with a much simpler construct:

**$array = @("test1", "test2", "test3")**

**foreach ($element in $array) {**

**$element**

**}**

The final thing to note with the foreach loop is that it can be accessed via the pipeline.  The foreach that is used in the pipeline is actually an alias to ForEach-Object. Using the pipeline this way is a faster way to write a foreach, but it can incur some overhead when dealing with large data sets. When you use foreach through the pipeline you are given a special variable to represent the element in your code block, $\_:

**$array = ("test1", "test2", "test3")**

**$array |foreach {**

**$\_**

**}**

We have already seen the foreach loop in action in the pipeline during [Part 2](http://powertoe.wordpress.com/2009/11/16/powershell-part-2-opening-files-writing-to-the-screen-the-pipeline-and-dot-values/) of this series when we looked at how to read files.  Here’s a snippit of that code so you can see how a cmdlet can pass a list through a foreach loop:

**Get-Content dictionary.txt | foreach {$\_.toupper()}**

Lists and arrays are very important to programming and scripting.  There is rarely a script that does not use an array or a list in some form or another.  Armed with this knowledge you can start looping away

Arrays can also be accessed linearly through the help of for loops.  A for loop generally has 3 bits of information in their declaration:

1. Initialization
2. Condition to continue the loop
3. A repeating occurrence – This is generally used to bring your loop closer to not meeting the condition for you loop so that the loop will eventually end.

This is best seen by looking at a simple loop.  The following will initialize by setting $i to 1.  It will then test that $i is less than 6 and increase $i by one on each pass of the loop:

**for ($i=1;$i -lt 6; $i++) {**

**"This is line number " + $i**

**}**

In the above example we use the -lt comparison operator to test that the value of $i is less than 6.  There are many other comparison operators available like -gt (greater than), -le (less than or equal to), -ge (greater than or equal to), -eq (equal), and -ne (not equal).  To learn about the comparison operators available to you can use the following command:

**help about\_comparison\_operators**

**Arrays**

For those that have never worked with arrays here’s a great way to understand them:  If a variable is a piece of paper then the stack of papers is an array.  It’s a list of variables or objects, and every programming/scripting language has ways to store these variables or objects linearly so you can access them later via a number of different methods.

So let’s look at how we can create an array of string objects in powershell:

**$array = @("test1", "test2", "test3")**

**$array**

You can also add an element to the end of an array:

**$array = @("test1", "test2", "test3")**

**$array += "test4"**

**$array**

You can also add arrays together:

**$array = @("test1", "test2", "test3")**

**$array2 = @("test4", "test5")**

**$array = $array + $array2**

**$array**

You can access an element of an array if you know the index number of the element you want.  Arrays are indexed by integers starting with 0.  This can be seen with the following code:

**$array = @("test1", "test2", "test3")**

**"First array value: " + $array[0]**

**"Second array value: " + $array[1]**

**"Third array value: " + $array[2]**

You can use that element as if it’s a regular variable at this point.  Since our array is an array of strings we can use string functions as if this was a regular variable set to that string value when we call the element of this array.  e.g.:

**$array = @("test1", "test2", "test3")**

**$array[1].ToUpper()**

A lot of people get intimidated by Regular Expressions, and maybe you should.  However, if you plan on doing any type of text manipulation or searching you will need to learn the basics.  Administrator tasks are littered with getting dumps of data into text form and having to extract, reformat, or find a way to make the data useful.  Regular Expressions do just that.

Before I continue I must mention that the be-all-end-all source of information for regular expressions is [“Chapter 5 – Pattern Matching” in the Perl Bible, “Programming Perl” (The Camel Book)](http://books.google.com/books?id=ezqe1hh91q4C&lpg=PR3&dq=programming%20perl&pg=PA139#v=onepage&q=&f=false) from O’Reily.  I highly recommend reading this chapter a few times (God knows I must have read it over 50 times).  It contains everything you need to know about regular expressions.  I don’t plan on discussing how to form regular expressions in this tutorial, but I want to show how Powershell implements regular expressions.

Now let’s get dirty.  Powershell implements two different methods of regular expressions.  There is a built-in method using comparison operators and there is the method of using a .Net Regex.

**Comparison Operator Implementation**

**get-help about\_Comparison\_Operators get-help about\_regular\_expressions**

**Comparison operators are things like -eq (equal), -gt (greater than), -ne (not equal).  The regular expression operators are** **-match,-notmatch, and** **-replace.  There are also two additional operators not listed in the above helps that do the exact same thing as their counterparts, but they are case sensitive:** **-cmatch** and **-creplace**.

**'test' -match '^t'**

The above returns true because ‘test’ contains a ‘t’ as its first character.  If we apply the -match above to the code from [Part 2](http://powertoe.wordpress.com/2009/11/16/powershell-part-2-opening-files-writing-to-the-screen-the-pipeline-and-dot-values/) of this series of tutorials that reads our dictionary file we can create a script that will list all words that start with the letter ‘t’:

**Get-Content dictionary.txt | foreach { if ($\_ -match '^t') { Write-Output $\_ } }**

In addition to using matches to determine if something is true you can also use them to capture groups within your matches.

**'Powertoe is the best' -match '(\w+) is the (\w+)' foreach ($match in $matches) {write-output $match}**

The $matches special variable is used to display the contents of the match.  $match[0] will always show the entire regex match (as it does in a .net match capture), while $match[1] will show the contents captured by your first parenthesis pair, $match[2] will show the contents of the second pair, etc.

-replace is very similar to -match, but there is one very important thing to note.  -replace does not interpret a regular expression the way a normal regular expression works.  It will always do a greedy match (perl equivalent: s///g):

**'test' -replace 't', 'w'**

The output from the above operator is ‘wesw’ as opposed to ‘west’ like you would expect with a normal regular expression replace.  In order to have -replace work this way is to use the .Net implementation of regular expressions.

**.Net Implementation**

[System.Text.RegularExpressions.Regex](http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex(VS.80).aspx) is the name of the class that implements regular expressions in .Net.  Here is how you would normally create an object of the Regex type:

**$regex = new-object System.Text.RegularExpressions.Regex('^test$',[System.Text.RegularExpressions.RegexOptions]::MultiLine)**

All methods and properties available to this class are available to powershell.  However Powershell also gives a nice shortcut to creating regex objects.  The above command can also be written as:

**$regex = [regex] '(?m)^test$'**

As mentioned we can now use this implementation of .Net’s Regex to do a non-greedy replace:

**$regex = [regex]'t' $regex.Replace("test","w",1)**

That’s the down and dirty of it.  I apologize if this topic is a little bit more advanced than my others have been. I know it expects you to have an understanding of regular expressions.  I highly suggest reading both the [pattern matching chapter of the Camel book](http://books.google.com/books?id=ezqe1hh91q4C&lpg=PR3&dq=programming%20perl&pg=PA139#v=onepage&q=&f=false) as well as the supporting documentation about the [Regex class](http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex(VS.80).aspx) to ensure that you learn all of the power available to you within regular expressions.  After that it’s just a matter of setting yourself to tasks that do string manipulation.  After the next couple of tutorials we should be able to start tackling real puzzles where you can challenge yourself to implement your own regular expressions so that you can start learning how to form them.

**======Update======**

I gave a very in-depth talk on Regular Expressions to the UK user group in March 2011.  The live meeting recording of the presentation is online to view [here](http://richardspowershellblog.wordpress.com/2011/03/23/regular-expression-recording/).