Ref: <http://powershell.com/cs/blogs/ebookv2/default.aspx>

 <http://www.techrepublic.com/blog/10things/10-powershell-commands-every-windows-admin-should-know/2052>

Goals:

* Understand basic Powershell command line usage.
* Understand how objects are processed through the pipeline
* Create a script that can be scheduled

**Deliverables:**

You have two documents that you must complete that should be placed together in a single zip. The first is to show that you worked through the initial basic tutorial. The second will be a powershelll script that utilizes a tool of your choice. For both assignments, you may consult with others, and you should partner with one or two people (no more than three in a group). Those who took IS271 will be familiar with the first part of this assignment, and those people should work with those who have not had IS271.

**Part 1 – Warm Up To Powershell**

Welcome to Powershell! Before anything else, you should know that Powershell is based on Microsoft .Net technology. What that means is that all data types are objects – there are no primitives. Knowing that will be helpful as you go through this warm up exercise.

As your work through the exercises, record your responses in a flat text document. You can use notepad. To start notepad, first open up the Powershell command line interpreter. Then enter a command similar to the following (use the drive letter and folder appropriate for you) (Of course you can use Notepad++ as long as the file created ends with .ps1):

 C:\>notepad P:\IS271\myscript.ps1

You may get a dialog box saying the file does not exist and asking you if it should be created. If so, click on the “Yes” button.

Enter your names at the top of the notepad document.

GOING FORWARD, ENTER ALL THE RESPONSES ABOUT OBSERVATIONS OR SOLUTIONS WHEN REQUESTED INTO THE NOTEPAD FILE. PLEASE PROVIDE THE NUMBER OF THE RESPONSE THAT APPLIES.

**Practice on the Command Line**

Powershell ‘s default behavior is to display values of expressions. So at the command prompt for Powershell, simply enter the number 46 and hit enter.

 C:\> 46

You will see the result is a display of the number itself.

Now enter a simple expression to multiply your age by 3.14.

 C:\>56 \* 3.14

We also have strings. Enter the following at the command line and observe the results:

 C:\> “Hello, Pat”

 C:\> "Hello, Pat. " + "And everyone else, too."

 C:\> "Pat" \* 10

**Response 1**: Explain what the plus (+) and the multiplication (\*) signs did.

So far, there are no surprises. Powershell is more than a simple calculator, it also has commands built into it. These are called cmdlets (pronounced command-lets). One cmdlet lets you list the other cmdlets. To get a list of all the cmdlets available, enter the following:

 C:\>get-command -type cmdlet

Notice that cmdlets have a pattern of verb-noun. Another cmdlet is get-member. It reports on the class type of any data (objects) provided to it as input. Enter the following:

 C:\> 338.019 | get-member

**Response 2: Research the get-member cmdlet to determine its purpose. Then note the Type Name given:**

Now enter the following:

 C:\> "Friends are great" | get-member

You can see that this is a string, and it has the typical methods normally associated with a string. Now enter the following :

 C:\> "Friends are great".ToUpper()

To reuse this string several times and save some typing, enter the following:

 C:\> $x = "Friends are great"

This creates a variable that holds that value. To test that, enter the following:

 C:\> $x

After hitting return you see the original string. Now do the following:

 C:\> $x.ToUpper()

Again, the results are similar. Now, do this again:

 C:\> $x

**Response 3a:** Huh? It doesn’t have the upper case letters anymore. Explain why?:

That was an example of a method. To use a method requires parenthesis behind the name of the method. If any arguments are needed, they go inside the method – again, similar to what you may have experienced in other OO languages.

**Response 3b:** Is $x an object or is it a class? Explain.

**Response 4:** Now, do the following :

Open a browser and go to the following URL:

<http://msdn.microsoft.com/en-us/library/system.string.aspx>

You see that it is the reference documentation for the string type and its class members. Review the methods available. Then, find the methods that will accomplish each requirement listed below. In your notepad document, note the command you had to use to implement the functionality using the $x variable created above (it should still have the assigned value of “Friends are great”.) For instance, the answer to problem **a** is: $x.Contains("great") (Note: to see a detailed explanation of a method you see on the webpage, click on it) MAKE SURE THAT WHAT IS ENTERED IN NOTEPAD CAN BE COPIED AND PASTED BY THE INSTRUCTOR.

|  |
| --- |
| Functionality |
| 1. Determine if the string includes the word “great”
 |
| 1. Pad it with five exclamation marks (!) at the end without using the plus sign – use an appropriate method.
 |
| 1. Print out only the six letters after the third one.
 |
| 1. Determine how many characters are in the string
 |

Now enter the following:

 C:\>get-date

**Response 5:** What did you observe?

Now enter the following commands and observe what is taking place:

C:\> get-date

C:\> $dateVariable = get-date

C:\> $dateVariable

C:\> get-date | get-member

**Response 6a:** You will have noticed several members when piping a datetime object into get-member. The documentation for using those methods is at: <http://msdn.microsoft.com/en-us/library/system.datetime.aspx> Take time to review that page and the slides presented in class. Then write the commands that will accomplish the following:

1. Add three days to the value in $dateVariable with the results assigned to a new variable named $datePlus
2. Subtract three years from the value in $dateVariable with the results assigned to a new variable named $datePast
3. Write the command to print out only the hour of the day of the date in $dateVariable

It’s been a while since you created $dateVariable. We can determine how long it’s been with the following command:

C:\> (get-date) - $dateVariable

There are a couple of things to note. The parenthesis is needed around get-date so Powershell would realize you only want the value and not to see the minus sign (-) as a parameter flag to the command. Now, try this (don’t panic, it won’t work):

C:\> $dateVariable - "10/01/2009"

The problem is the $dateVariable is a datetime data type and what appears in quotes is a literal string (meant only to be read as a string of characters.” To tell Powershell to treat “10/01/2009” like a date, there are two ways to do this. One is through casting as below:

C:\> $dateVariable - [datetime]"10/01/2009"

**RESPONSE 6b:** EXPLAIN WHAT IS MEANT BY CASTING.

The other way now introduces the use of a static member of a Class Type. Most of the value types have a static class member called Parse. Static members are not meant to be used with a specific instance of the class in an object. A static member is meant to provide an all-purpose piece of functionality. To use a static member, you have to make a reference to the class using the namespace convention. So to convert a string that “looks like a date” into an actual date can be done with the following:

[System.DateTime]::Parse("10/01/2009")

You can see it created a datetime object and Powershell displayed it in that format.

**RESPONSE 7:** Using the static method above, write a command that will assign the conversion of the string to a variable name $dateFromString. Write another command that will verify for you visually that $dateFromString is of type System.DateTime. Then write another command that subtracts $dateFromString from the current date. Make sure the commands used are recorded into the notepad document.

**Working More With Cmdlets**

We’ll wrap this up by looking at the get-process cmdlet. Enter the following on the command line:

get-process

Very likely you saw several things scroll by. This is information about all the applications currently running on your desktop. You can search for the existence of a specific program with the following command:

get-process notepad

If you are running only the instance of notepad you started to record your commands and comments you should see only one listed. Start up another notepad session either from the command line or through the standard Windows interface. Then, run the command above again.

You should have notice two processes. Now, save your work in the notepad instance you use to record your responses. **I REALLY DO MEAN FOR YOU TO SAVE YOUR WORK RIGHT NOW, OR YOU’LL BE SORRY.** Now enter the following command:

help stop-process

Read the documentation and determine how to stop both of your notepad instances running.

**RESPONSE 8:** Make sure you find a way to hold on to what you did. You will have to restart notepad in a moment and then put in the command you had used to stop it. NOW: RUN THE COMMAND YOU WOULD USE TO STOP THE NOTEPAD PROCESSES YOU CREATED.

Now we need to start notepad up again, this time, we’ll do it differently. Enter the following at the command line:

$myProcess = [System.Diagnostics.Process]::Start("notepad")

$myProcess

Take time to observe what is taking place. We are referring to a specific class of System.Diagnostics.Process and we are using a static method named “Start”. As you can see, it finds an application named “notepad” and starts it up. When you entered in $myProcess, you saw that it displayed what was normally the result of the get-process command.

Now, go into the notepad program you just started and open the file you were listing your commands in. Enter the command you used originally to kill it. Then, make sure you save it.

One more thing, since $myProcess is an object, it has information about that specific notepad process you just started. Enter the following command:

C:\> $myProcess | get-member

Again, get-member is describing the data type. Look at the top and verify that $myProcess is an object of type System.Diagnostics.Process (Keep in mind $myProcess is not the process itself. It’s an object that has information about the process.)

Now, go to the Microsoft documentation at <http://msdn.microsoft.com/en-us/library/system.diagnostics.process.aspx> and read the first two paragraphs of the Remarks section. Then, scroll up to read about the methods and properties of the Process class. Read about the member “HasExited” (it’s a property). Then read about the member Kill(). On your own, write commands to accomplish the following using the $myProcess variable:

* Determine if the notepad application you started has exited (we’re assuming it hasn’t)
* Stop notepad from running
* Verify that notepad has stopped running
* Print out the time that notepad stopped running

**RESPONSE 9: Reopen your notepad file and enter the commands you used to accomplish the above.**

**Regular Expressions:** As discussed, regular expressions help find patterns. More specifics on this are at <https://technet.microsoft.com/en-us/magazine/2007.11.powershell.aspx> . For now, observe that there are log files in the C:\windows folder. Enter the following command:

get-content C:\windows\WindowsUpdate.log

Observe the contents of this file. The following command will display only those lines with the word “WARNING:” with “URL” appearing sometime later.

get-content C:\windows\WindowsUpdate.log | %{ if ( $\_ -match 'WARNING.\*URL') { $\_ } }

Do some research and modify this so that only lines that end with “.dll” are displayed. Enter this into your flat text file of commands.

Hold on to your flat text file. You will make it part of a zip file that includes your script from the next exercise.

**PART 2 – Create a script**

For this assignment, you will write a powershell script that will utilize any command or cmdlet that allows you to examine the status of a network component or log file. It is incumbent upon you to research the command or cmdlet and determine with your partners how to utilize the command. You will also be given a sample.

First, make sure your environment is set up so you can run a script. To do this, make sure you run PowerShell as Administrator. Once PowerShell starts, enter the following command:

 Set-ExecutionPolicy Unrestricted

Respond to the prompt with “Y”

Next, download the sample.ps1 script to a folder. Then in the PowerShell command window, change the current working directory to that folder. (Similar to Unix, you can use the **cd** command. For example: cd S:\IS333\Powershell )

Once in that folder, enter the name of the sample.ps1 PowerShell script at the command prompt. Once you verify you can run it, you can use it as a starting point to write your own script. For your script, determine with your partner what PowerShell cmdlet, logfile or WMI resource you would like to explore (note Chapter 18 of the tutorial for some examples of using WMI). Write a script that you would run daily to create a report. It can be related to measuring network connectivity, the health of the system etc. All your output, similar to the example, should end up in an HTML file.

To have your script automatically run, you need to add it to the Task Scheduler. Start the Task Scheduler by selecting the Windows Start button. In the command text box, entry “Task Scheduler” As you enter it, you should see the Task Scheduler application listed. Right click on it to “Run as Administrator”

It will look similar to the following:



On the right hand side, you will see the option for “Create Basic Task.” You will be prompted for the name of the Action. This is arbitrary. Enter “Sample Service”. Don’t worry about the description. Click Next. On the list of options, select “one time” for the start time of this task. Click Next. The next dialog display is requesting a specific time. Set it up to run five minutes (or less if you prefer) from the current time. Click Next. Select the radio button “Start a Program.” Click Next. In the next display enter “Powershell.exe” in the text box labeled “Program/Script” in the Add Arguments put in the name of the script followed by the appropriate flags.

Wait for the time and check to see if it ran.

When things are working, place the script in a folder with the text file from the first part of the assignment. Make a zip file and submit it to Moodle.