CSC 368 – System Programming Languages
Spring 2017

Homework 04 – PowerShell

For this homework we will investigate how to reproduce our batch file for resizing images with a PowerShell script. Follow the steps below in the PowerShell ISE to create a script.

Any time you create a script you will likely have to edit it later or move it to another platform… or in this case, modify it for the same platform using another scripting language.

To proceed through this homework, we will look at the concepts we used in the batch file project and see how those are implemented in PowerShell. To do that, we will need to go back to our method of creating a solution to any problem… We start with the problem statement. The problem statement from the batch file project was:

> "…write (and test) a utility batch file to resize all the images in a directory to 640 pixels wide (web size), placing the resized images into a sub-directory called "web-img" (the original pictures will be unchanged)."

We also had some requirements:

> Your batch file should meet the following criteria:
>
> 1. Name your batch file websize.bat
> 2. Your batch file should take the name of the directory containing the images as an argument
> 3. Your batch file should take the file extension of the extension of the image type to be resized as an argument.

Your batch file probably looked something like this:

```
if not exist %1 goto DIRERROR

if not exist "%1\web-img" md %1\web-img

for %%I in ("%1\*.%2") do (
    ECHO resizing %%I
    convert %%I -strip -adaptive-resize 640x "%1\web-img\%%~nI.%2"
)

EXIT

:DIRERROR
ECHO Error, specified directory does not exist
```

There were also various decisions that could have been made, for example, What happens if the command is not provided with parameters? Should you check for existence of the directory? What about the sub-directory? You may address these in your final homework turn in, but for this homework, we will use the pseudocode description that follows:

```
<# PowerShell Homework 04
 This script repeats the project for the Batch file in order to show parameter passing
 and control flow of PowerShell scripts.

 Pseudocode
     Define parameter names

     If <directory parameter> does not specify a directory
         display an error message
         exit the script
     If the sub-directory <directory parameter>\bob does not exist
         create the sub-directory

     For each <file> in the directory <directory parameter>
         If <file> is an image of type <extension parameter>
             convert <file> -resize 640x <directory parameter>\bob\sm_<file>
#>
```

There are always multiple ways to solve problems with computers. For the purposes of this homework, this is the method we will plan on using. Keep in mind that starting out with one solution in mind does not mean that you won't change. The other thing to note is that formal algorithms and pseudocode are not always necessary when creating a program. For most of the programs you have written for school, you have been asked to simply create the solution, not to provide some form of documentation. When programming for yourself, this is the norm. When programming on a project, you will probably be required to provide your design before you write your program and you will need to include documentation, by way of comments, in the code you do end up writing.

Use the pseudocode above as both our design and our initial comments to start our PowerShell script. Be sure to insert your name inside the comments. Note that a multiline comment in PowerShell begins with a **<#** and ends with a **#>**.

First things first. How are parameters passed in PowerShell???

If you recall, in batch files we could have up to 255 command line parameters, and they were accessed by special variables. **%0** was the batch file invocation (i.e., what name was used on the command line to make the batch file execute). We could set up shortcuts to the same batch file and then change the operation of that file based on what shortcut/name was used to call it. You might think this unimportant, but it is done more frequently than you might guess. For example, a Zip archive program might use its invocation name to determine whether it is zipping or unzipping a file (i.e., "zip" would attempt to compress a file while "unzip" would attempt to extract files).

The next nine batch file parameters could be accessed directly using **%1, %2, %3, … %9**. Remaining parameters required operations on the parameter string to assign them to one of the **%1 – %9** variables. The biggest problem with this method of accessing the parameters is that the numbers have no symbolic meaning that makes is clear what they are being used for. It can get confusing to remember what **%1** is supposed to be. The order of parameters is then also fixed and you can't skip any of them, even if they are not needed for the current execution of the script.

PowerShell uses named parameters to solve these issues, though you can use parameter position as "shortcut" to assigning values to the parameters. We've seen this with some of the PowerShell commands we've already used. For example, try the following:

```
Get-WmiObject -Class Win32_OperatingSystem
```

and compare the output to the output of:

```
Get-WmiObject Win32_OperatingSystem
```

There should be no difference because the parameter position assigns `Win32_OperatingSystem` to the required parameter `-Class`, if the parameter name "`Class`" is not included in the command line invocation of `Get-WmiObject`.

What this means is, once we have declared our parameters, we can use any of the following as an invocation of our hw4.ps1:

```
.\hw4.ps1 <directory name> <file extension>
.\hw4.ps1 -filePath <directory name> -fileExt <file extension>
.\hw4.ps1 -fileExt <file extension> -filePath <directory name>
```

These assume our parameters are named filePath and fileExt. If they are, all three invocations will result in exactly the same functionality. We can also set other properties for the parameters, including default values and whether they are optional or mandatory, etc. For example, in our homework, it would be possible to make the script act on all the files in the current directory if the file path was excluded or perform some other action. For now, let's look at how we add parameters and how we get access to them inside our script.

NOTE: If you are familiar with C, you can also use the parsed string array of the command line to access command line values: $args is the array, with element 0 as the script name, 1 as the first parameter, 2 as the second, etc. You can assign variables directly using something like, `$filePath=$args[1]` in your script file. **This is not how we will pass parameters in this homework.**

For references on how to use parameters in PowerShell, you can perform a Google search. Here are a few references that will appear:

https://ss64.com/ps/syntax-args.html Yes, ss64.com has information on PowerShell 😉
http://windowsitpro.com/blog/making-powershell-params
http://www.powertheshell.com/bp_function_parameters/
https://technet.microsoft.com/en-us/library/jj554301.aspx
https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_functions_advanced_parameters

To use parameters with a cmdlet, script, or function, we need to declare them at the beginning of each one. They should be the very first lines other than comments in our script. The simplest syntax is:

```
Param (
[<type>] $parametername1,
[<type>] $parametername2,
…
[<type>] $parameternameN
)
```

For example, if we have a script that takes the first and last names of someone as the parameters we might have:

```
Param (
[string] $firstName,
[string] $lastName
)
```

Make sure you notice the comma between the parameters. They do not have to be on different lines. We could have entered:

```
Param ([string] $firstName, [string] $lastName)
```

which would have worked just as well.

For our example, we are using a path for our files and an extension to indicate the type of image file to resize. Please declare two parameters for your script now (they will be strings) and call them anything you want to call them. Remember, this must be the first PowerShell code in your file. The only thing that can come before it is comments or blank lines.

How do we check if the file path parameter is a directory??

Yes, Google can be your friend here too (also an enemy, providing information overload or links to useless sites). Here are some links before I walk you through this…

https://ss64.com/ps/test-path.html
https://blogs.technet.microsoft.com/heyscriptingguy/2014/08/31/powertip-using-powershell-to-determine-if-path-is-to-file-or-folder/
http://www.powershelladmin.com/wiki/Use_test-path_with_powershell_to_check_if_a_file_exists
http://www.workingsysadmin.com/quick-tip-use-powershell-to-detect-if-a-location-is-a-directory-or-a-symlink/

For our homework, let's use the **`Test-Path`** cmdlet. For more information on **`Test-Path`** you can use the links above or go to the PowerShell online help here. We will use the same basic logic as the second link above.

Test-Path allows us to check for existence and other properties of a named location in the filesystem. The basic form check to see if a name exists as a location on the computer. For example,

```
Test-Path "C:\Program Files\Microsoft Office\root\Office16"
```

will return TRUE if the file exists or not, but will not tell you if it is a file or a directory. Try it. (You can also try it without the quotes…)

We can see what kind of object it is by checking property "PathType" for the object as well as the actual location of the object. The options available are "Any", "Container" and "Leaf" (without the quotes). Any will match anything and is the default test. Container is used for objects that can contain other objects, such as directories. Finally, Leaf is the type for all files.

For example:

```
Test-Path "C:\Program Files\Microsoft Office\root\Office16" -PathType Container
```

If we try these three options for the location above we will get the following results:

Any	True
Container	True
Leaf	False

If we try these three options for the executable file WinWord.exe in this location

```
Test-Path "C:\Program Files\Microsoft Office\root\Office16\WINWORD.exe" -PathType Container
```

above we will get the following results:

Any	True
Container	False
Leaf	True

If we check a non-existent location all three will return False.

It looks like we can check for whether the parameter is a directory, but not if it is NOT a directory. This is where we need the logical operators. There isn't a great deal in the get-help about_operators page, but we can find what we need. It is exactly what we would use for many programming languages, the '!' operator (exclamation point). We net something that is True only for containers, which is the opposite of the last table above. So our conditional test should be the not of that:

```
(!(Test-Path <your path parameter> -PathType Container))
```

Now that we can perform the test, we need to look at the IF statement syntax and logical operators to make sure we get the result we want. The syntax for the If is:

```
if (<test1>)
    {<statement list 1>}
[elseif (<test2>)
    {<statement list 2>}]
[else
    {<statement list 3>}]
```

Both of our tests are the IF part only, without an else, but we need to know how to exit the script… Hhhhhhmmmmmmmmm, back to Google… how to exit a PowerShell script

Google is not your friend here… There are a lot of links, but they don't all tell us what we need. We have to read a lot or read between the lines (or remember some other language and try something…). Our best options here are either the Break command (because we are not in a loop) or the Exit command. Both will completely stop all processing in the script and exit.

> https://ss64.com/ps/break.html
> http://stackoverflow.com/questions/2022326/terminating-a-script-in-powershell
> https://www.pluralsight.com/blog/it-ops/powershell-terminating-code-execution

You can use either one in your script. I used Break.

The other thing we want to do is display a message to the screen about the bad parameter.

Back to Google… how to write a message to the screen in a PowerShell script

We get a couple of interesting results… not too many to track down. Google is a friend again.

> http://windowsitpro.com/windows/write-output-or-write-host-powershell
> https://blogs.technet.microsoft.com/heyscriptingguy/2011/05/17/writing-output-with-powershell/
> http://stackoverflow.com/questions/2038181/how-to-output-something-in-powershell

Looks like "Write-Host" is our answer, the only question is whether or not we want to do colors with our error message… Check out the help page here.

Putting it all together we get:

```
if (!(Test-Path  $<parameter name> -PathType Container)) {
    Write-Host "ERROR  ERROR  <put something meaningful here>"
    Break
}
```

So far so good. Now we have to look at the sub-directory, but this should be pretty easy, right? We need to check for existence and create the sub-directory, if it doesn't exist. First thing we need to do is get a concatenated string with the directory above and web-img added to it. We could try doing this everywhere we need it (we'll need it for checking for existence, creating the directory, and saving the resized images), but it is probably easier to save it to a variable. So how do we concatenate a string in PowerShell???

> https://blogs.technet.microsoft.com/heyscriptingguy/2014/07/15/keep-your-hands-clean-use-powershell-to-glue-strings-together/
> http://stackoverflow.com/questions/15113413/how-to-concatenate-strings-and-variables-in-powershell

This is not one of the places where PowerShell shines, or at least the documentation doesn't. '+' should be the operator, but it works in strange ways with objects… Luckily, if we are assigning

to a variable rather than putting using in an expression we won't get lost with quotes and parenthesis. We can just put everything in double quotes and the work will be done for us:

```
$imageDir = "$pathParameter\web-img"
```

When we use double quotes the value of the variable/parameter is used rather than the object so this gives us exactly what we were hoping for.

Now that we have the directory name, we just need to check for existence and if it doesn't exist, create it. Since the command prompt commands from batch files work here, we can use:

```
md $imageDir
```

This will work, but the PowerShell for this is:

```
New-Item -ItemType directory -Path $imageDir
```

You can check this out with get-help.

NOTE: I am oversimplifying here, we have not considered whether "web-img" exists, but is not a directory. We would need an **Else** to cover this possible error. Feel free to add this, but it is not a requirement (5% extra credit if implemented properly).

We're on the home stretch now. We need to replace our loop. There are several loops available, but the Foreach loop would seem to be the most appropriate. Here is our Batch file loop:

```
for %%I in ("%1\*.%2") do (
    ECHO resizing %%I
    convert %%I -strip -adaptive-resize 640x "%1\web-img\%%~nI.%2"
    )
```

Let's take a quick look here and see if we can't find our PowerShell equivalents. If we look at the help for PowerShell we find that the loop is **`ForEach`**, which is different from the **`ForEach-Object`** cmd-let because we are not required to use piping and is therefore a little more flexible, especially for scripts. Here is the syntax:

```
foreach ($<item> in $<collection>){<statement list>}
```

You can find information on it directly from Microsoft here.

`$<item>` is just the loop variable that takes on the values in the collection, just as **`%%I`** did in our Batch file. Note that this must be inside parenthesis. The collection is an object because **`$<collection>`** begins with a '**`$`**' in the syntax listing. This is important because we need to know that we must **enclose any cmd-let calls inside parenthesis as well**. So now we just need to know how to get the listing of files…

The command we used in our batch file was **dir**, but **dir** is an alias for **Get-ChildItem**, so let's take a look at the help page for that. You'll notice that the first parameter in the syntax is the path:

```
[[-Path] <String[]>]
```

the parameter is optional (indicated by the square brackets around it) and the parameter name can also be left out, even if the string for the path is provided (the brackets around **-Path**). We do this all the time with the dir command, so let's give it a try in our script file.

```
Foreach ($imgFile in (Get-ChildItem $filePath\*.$fileExt))
{

}
```

Try running your script, **make sure there is an image file in the path parameter you use**. It seems to work okay, but what value does $imgFile contain? Do we need to worry about separating the path from the file so we can save it to a sub-directory as we did in our Batch file? Let's find out. Type the following inside your loop (on the blank line):

```
Write-Host "Working on image: $imgFile"
```

Now run your script again…. Sure enough, we've got to separate the path from the file name.

Back to Google. The search I tried was: how to separate the path from filename in Powershell

The first result is in the PowerShell reference (i.e., the online help). It looks like Split-Path will work for us. Example 2 shows what we need, we need to get the filename portion of the full path so we can put the resized file in the sub-directory. To do this we will need to build the new file path, just as we did in the batch file. The -Leaf parameter switch gives us the value we need. Let's try it out by putting it in a variable and printing it out as our progress message.

We will now have the following two lines inside our loop:

```
$currentImg = (Split-Path -Path $imgFile -Leaf)
Write-Host "Working on image: $currentImg"
```

This should display a listing of all the files in the given directory with the correct file extension. Now we need to build the rest of the destination file name and path for the convert program.

The pieces we have:

The sub-directory:	**$imageDir**
The file name:	**$currentImg**

Assuming the convert command is in our PATH environment variable, we just need the command we had before with the source file and destination file changed to the new variables.

```
convert $imgFile  -resize 640x $imageDir\$currentImg
```

Try it…. What's wrong???? Darn PowerShell uses Objects, if we want the values we need to put them in double quotes. Try:

```
convert $imgFile  -resize 640x "$imageDir\$currentImg"
```

Now try it out….

Make sure to include your name at the beginning of the comments at the top of the file and turn in your resulting PowerShell script.