

Understanding Scripting With Windows PowerShell

Jeffery Hicks
SAPIEN Technologies

Pre-requisites for this presentation:

- 1) Some scripting experience
- 2) Some PowerShell experience

Level: Beginner

Agenda

- What is PowerShell Scripting?
- PowerShell's "Scripting" language
- PowerShell, Functions and the Pipeline



Why Script?

- Long, complex command-lines are tedious to retype
- Copy and paste into a .PS1 file
- More complex tasks may require more than one command

Scripting vs. Interactive

- There is ***no difference*** between scripting and using the shell interactively
- Anything you can do in a script, you could just type into the shell... and vice-versa
- Test in the shell and build a script from there

The Script File

- Plain-text files with a .PS1 extension
- Scripts don't run by default
- Scripts can be digitally signed (Set-AuthenticodeSignature)
- You must specify a path when running a script

Script Editing

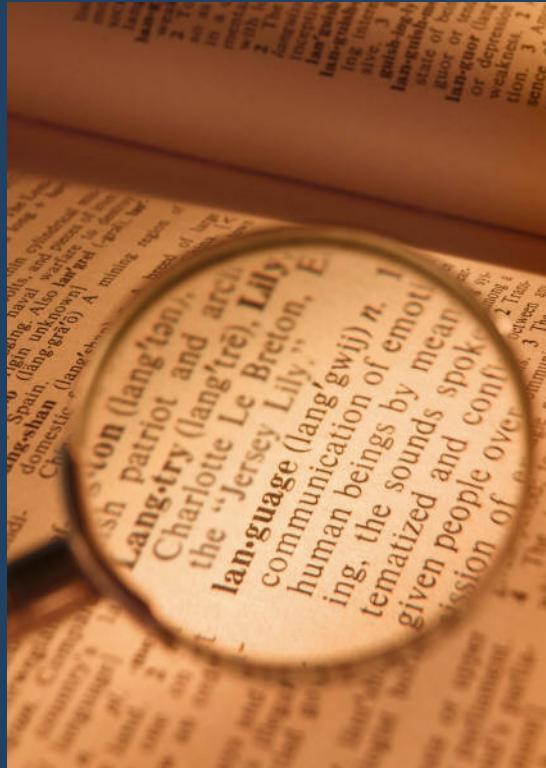
- Notepad is a bare-minimum tool
- Microsoft does not produce a "PowerShell Editor"
- Third-party scripting environments like PrimalScript are designed specifically for working with PowerShell
- Use *something* that fits your budget and needs

Scripting FAQ

- Can scripts have error handling?
- Can scripts accept input arguments?
- Can scripts play in the pipeline?
- Do scripts require PowerShell to be installed in order to run?
- Can scripts be packaged (not "compiled") into standalone executables?
- Can I write a script?
- The answer to all of these is **YES!**



Language





Language

- PowerShell has just over a dozen keywords that are considered its "scripting language"
- This does not mean these elements can only be used in a script...
- ...it's just that they're *most often* used in a script

Decision-making constructs

- If...ElseIf...Else
- Switch

If...Elseif...Else

```
If (expression) {  
    # code A  
} ElseIf (expression) {  
    # code B  
} Else {  
    # code C  
}
```

Notes

- Positioning of { and } not picky
- If is **mandatory**...
- ...ElseIf is optional...
- ...Else is optional
- Only *first matching code block* is executed

If...ElseIf...Else Example

```
If ($var -contains "a") {  
    # contains A  
} ElseIf ($var -contains "b") {  
    # contains B, but not A  
} Else {  
    # contains neither A nor B  
}
```

Switch

- Evaluates a single item against a range of possible values
- Comparable to Select Case in VBScript
- Great for multiple comparisons
- *Every matching condition* is executed!

Switch Example

```
Switch ($var) {  
    "foo" { # code }  
    "*foobar*" { # code }  
    "bar" { # code  
            break }  
    default { # code }  
}
```

Switch Notes

- Values can contain * wildcards
- Supports regular expressions
- "Default" block is executed only if no condition has matched
- Use **break** in a condition's code block to exit the switch construct (preventing further matches)
- Help [about_switch](#) for more information

Looping Constructs

- For
- Do...Until/While
- While

For

```
For ($i=0; $i -lt 5; $i++) {  
    $i  
    # or run some code  
}
```

- Starts with \$i equal to 0
- Continues while \$i is less than 5
- Increments \$i by one each time through

Do...While

```
$i = 10  
Do {  
    # some code  
    $i--  
} While ($i -gt 0)
```

- Block code always executes at least once

Do...Until

```
$i = 0  
Do {  
    # some code  
    $i++  
} Until ($i -gt 10)
```

- Block code always executes at least once

While

```
$i = 10  
While ($i -gt 0) {  
    # some code  
    $i--  
}
```

- Block code does not always execute at least once... as in this case

Enumeration

- Foreach
- Technically, an alias to ForEach-Object
- PowerShell can read your mind

ForEach

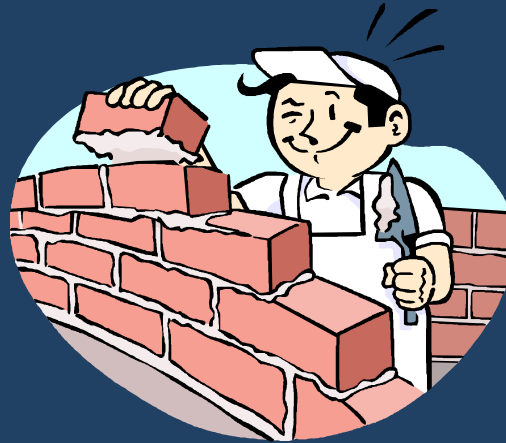
```
$wmi = Get-WMIObject Win32_Process  
ForEach ($proc in $wmi) {  
    $proc.Name  
}
```

- Use the variable you define (\$proc) to represent the current object from the collection (\$wmi)

ForEach-Object

```
PS C:\> get-wmiobject win32_process |  
    foreach {  
        if ($_.workingset -gt 10mb) {  
            write-host $_.name -foreground red  
        }  
        else {  
            write-host $_.name  
        }  
    }
```


Modularization



What Are Functions?

- Self-contained modules of code that typically perform a single task
- Think of them as mini-cmdlets
- Functions can be defined within any scope

Rules

- Functions must be defined *before* they are called
- If a function accepts input, call it like this:
FunctionName Arg1 Arg2
- Functions should not refer to any elements (e.g. variables) from other scopes

Remember: Scope!

- Functions have their own internal scope
- Functions should not use or modify variables from their parent scope
- Always assign a value to a variable, and ensure that arguments have default variables, before using them.



Tip: Nested functions

- You can include functions *within other functions*
- This allows a single function to be completely self-contained – if it relies on other functions, they can be nested within it.

A Basic Function

```
Function SayHello {  
    Write-Host "Hello!"  
}
```

```
PS C:\ SayHello  
Hello!
```

Function Input

- Use Param keyword
- Cast variable to the right type
- Declare a default value

Function Input Example

```
Function SayHello {  
    Param ([int]$arg1=1,[int]$arg2=2)  
    $result = $arg1 + $arg2  
    Write $result  
}
```


Function Input Example

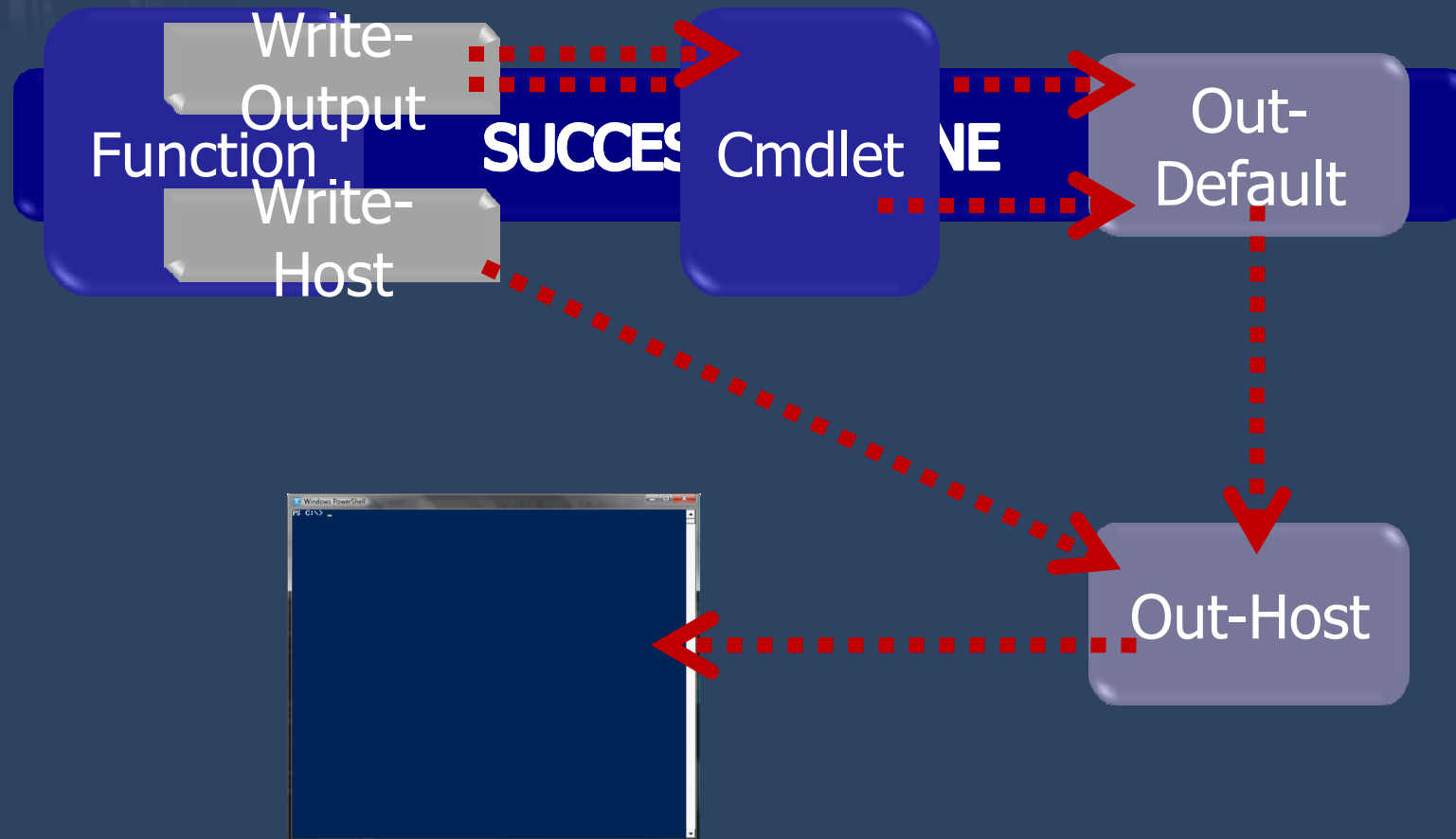
```
Function Foo-Bar {  
    Param ([string]$name=$(Throw "You  
must enter a computername!"),  
        [datetime]$date,  
        [int]$offset=3  
    )  
    #do something  
}
```

```
PS C:\ Foo-Bar -name "DC01" -date  
3/4/08 -offset 5
```

Function Output

- Write-Host writes to the console window
- Write-Output writes to the pipeline
- Return writes one item to the pipeline and exits the function immediately

Function output, illustrated



Function Output Summary

- Use Write-Host to write directly to the console
- Use Write-Output to write the function's "output"
- No real need to use the Return keyword

Functions In the Pipeline

- Functions can accept pipeline input and create pipeline output
- A function can use three specially-named ScriptBlocks to process pipeline input:
 - BEGIN: Executes once before any pipeline input is handled
 - PROCESS: Executes once for each pipeline input object, refer to each object with \$_
 - END: Executes once after all pipeline input objects have been processed
- Use Write-Output to send down the pipeline

Demos



Functions as Tools

- If a function is defined in a script...
- ...then it exists only while that script runs
- If you want a function to be available at the command-line...
- ...either define the function in your profile, or *dot source* a script that contains the function

Script Best Practices

- Name scripts using a *Verb-Noun* naming pattern (like PowerShell cmdlets)
- Comments and more comments
- Keep each script focused on a single task
- Add scripts to source control for long-term storage and safekeeping



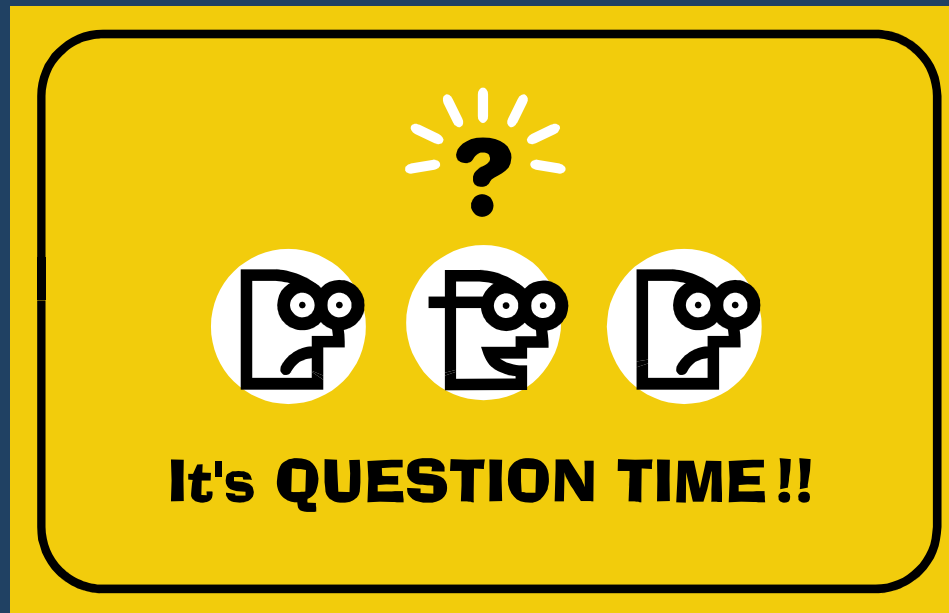
Resources

- blog.SAPIEN.com
- www.ScriptingAnswers.com
- www.PowerShellCommunity.org
- www.ThePowerShellGuy.com

Resources

- *Windows PowerShell v1.0: TFM 2nd edition* (Don Jones & Jeffery Hicks)
- *Windows PowerShell Cookbook* (Lee Holmes)
- *Windows PowerShell in Action* (Bruce Payette)

Questions



Summary

- Scripting and interactive use = same thing
- Scripts live in .PS1 files
- Scripts make complex tasks easier to perform, debug, and repeat
- PowerShell's "scripting language" is short and simple (easier to learn)

Thank you

- jhicks@sapien.com
- Follow me: twitter.com/JeffHicks

