# Preface

This is Studly McBeefy’s base PowerShell workshop. The aim of this workshop is to get you competent in PowerShell quickly. There are no quirky gimmicks or “master class” shenaniganry here. My approach is simple. We’ll walk through a bunch of things you should know and see what happens. If you’re the curious type you’ll dig deeper in each area on your own. If not, you should still be able to fumble though quite a bit. At the end of it all you should be able to read a lot of the PowerShell out there, and be able to churn out simple code to accomplish a wide range of tasks for better or for worse.

Other, more focused bolt-ons to the workshop may exist. Also, they may not.

Ideally, you’ll be taking this workshop with me. The content herein has been created to facilitate guided exploration. If you’re doing this on your own a few things may not be so clear. Still, in the interests of saving your from skiddiedom, open materials are provided. There is enough here to provide the “what”, but the “why”, “when” and “how” may require some additional instruction.

So, what’s in the box?

* PowerShell Workshop.ps1 – The curriculum. This is the meat.
* examples/\* – Simple example code to provide somewhat realistic examples of PowerShell usages. Delicious potatoes, of the Yukon Gold variety.
* Vagrantfile – A simple Vagrant configuration to allow you stand up a quick and easy sandbox for learning. This isn’t necessary, but its advised you use it. Its fresh sausage gravy on top.

Enjoy!

--

Here is the gist: If you got this file from the workshop GitHub you’re probably fine. If not, this might not make a lot of sense. Check out the readme in the repository root. The short-short is that this is a guide to \*your\* exploration. Poke, play, prod, learn, break, hack, drink, fun!

--

## Topics

1. Introduction and PowerShell Basics – What and Why, Basic Syntax, and Getting Help
2. Basic Workflow – Pipeline, Filtering, File I/O, Filesystem and Registry
3. Scripting – Execution Policy, Conditionals and Flow Control, Functions, Splatting, Custom Objects and ArrayLists.
4. Modules – Finding, loading, and authoring modules.
5. Code Execution – Packing and Execution of code.
6. Web Interaction – Basic web-based communication.
7. Remoting – PowerShell and CIM Remoting.
8. .NET I/O – Use .NET I/O directly for speed and flexibility.

## Prerequisites

1. Basic scripting or programming experience.
2. GUI Windows Operating System with PowerShell 5.0 or later.
3. Local or remote system with WinRM enabled and usable. (For Remoting section only.)

# Outline

1. Introduction and Basics
   1. Introduction
      1. What is PowerShell?
         1. Task-based command line shell.
         2. Scriptable.
         3. Object Oriented.
         4. Extensible though PowerShell-specific add-ons, .NET Framework, and native platform APIs.
      2. Why
         1. Functional intersection of other tools which did not necessarily work well together.
            1. Command Prompt / cmd.exe – Command line interface. Easily scriptable.
            2. Windows Scripting Host / cscript.exe – VBScript, Jscript, etc… Native, object-oriented Windows scripting.
            3. .NET – Re-use investment in the .NET ecosystem, including integration into .NET based-software.
   2. Syntax Basics
      1. Before We Begin – Verbosity and Aliases
         1. Aliases will be avoided.
         2. In most cases, commands will have all parameters qualified with their names.
         3. Flowery, verbose coding style.
      2. PowerShell Prompt
         1. PowerShell is mostly the same at the interactive prompt and in scripts.
         2. There are a few exceptions which will be noted as they come up.
      3. Variables
         1. Assignment – $Variable = value
         2. Quotes – $Variable = “value”, $Variable = ‘value’, with and without interpolation.
      4. Objects
         1. Properties – $Variable.Property
         2. Everything is an object. – $Variable.GetType, $Variable.GetType(), (7).GetType()
         3. Introspection is great but may catch you off guard due to PowerShell syntax.
      5. Commands – Cmdlets and Advanced Functions
         1. About
            1. PowerShell uses Spaces and dash-prefixed parameter names to define Cmdlet parameters.
         2. Examples
            1. Verb-Noun
            2. Verb-Noun Parameter
            3. Verb-Noun -ParameterName Data
            4. Verb-Noun()
         3. Additional Comments
            1. Cmdlets are not functions, but can look similar.
            2. Use spaces and explicit parameters whenever possible to avoid confusion and unexpected behaviors. Failure to use the preferred convention can lead to unexpected results.
            3. Functions (discussed later) allow function-like syntax, but Cmdlets don’t.
            4. Sometimes parameters may be auto-converted to a collection (discussed later.)
      6. Compound Statements
         1. About
            1. PowerShell expressions can be evaluated individually and incrementally, but the aim is to chain items to complete tasks.
            2. It is necessary to understand and use additional elements to properly control code execution.
         2. Examples
            1. () – Grouping operator. Execute this first.
            2. $() – Sub-expression operator. Execute this first in a sub-shell. Allows for complex statements, including multiple semicolon-separated statements.
            3. {} – Script Block.
         3. Additional Comments
            1. Many other language features are based on these basic elements, such as control structures.
      7. Types and Casting
         1. About
            1. PowerShell is dynamically, implicitly, weakly typed.
            2. Variables can (usually) change type. Type is figured out via context (usually). Types are automatically converted as needed at run-time (usually).
            3. Casts (usually) apply to assigned variables, causing right-hand operands to be converted to the now explicitly declared type.
         2. Examples
            1. Cast to an Integer.
            2. Cast between strings and integers.
         3. Additional Comments
            1. Dynamic typing can lead to awkward issues, or convenience, depending on the situation. Empty strings meaning false, etc…
            2. This is important when looking at Function parameters and reference to .NET types.
      8. Collections
         1. About
            1. PowerShell (and .NET) have several types of Collections. The most commonly used are the Array and the Hashtable.
         2. Examples
            1. Array Use
            2. Array immutability.
            3. Hashtable Use
            4. Hashtable appends.
         3. Additional Comments
            1. Other Collection types can be instantiated by referencing .NET .
      9. Comparators
         1. About
            1. For consistency and cleanliness PowerShell comparators look and work like Cmdlet parameters.
         2. Examples
            1. Mathematical symbols don’t work.
            2. Numerical Comparators
            3. Text Comparators
            4. Comparators and Collections
         3. Additional Comments
            1. Be wary of Collection placement in comparisons. Left-hand versus right-hand can make a difference. @( 1, 2, 3 ) -ge 2 = @( 2, 3 ) versus 2 -ge @( 1, 2, 3 ) = ERROR
            2. -contains versus -in is essentially the same thing with the left- and right-hand values reversed. -in was introduced in PowerShell 3.0, so -contains is more compatible.
            3. The -match operator is very useful (regular expression matching) but isn’t covered here. See issues with automatic variables and Select-String.
   3. Getting Help
      1. About
         1. PowerShell help is generally good.
         2. Examples
            1. Tab Completion
            2. Get-Help Verb-Noun
            3. Get-Help about\_\*
            4. Update-Help
            5. Microsoft Developer Websites.
         3. Additional Comments
            1. When Googling look for docs.microsoft.com over technet.microsoft.com .
            2. Be wary of content-farm websites and lazy example code.
            3. PowerShell was designed to be clear, consistent, and expressive. Avoid help material that teaches you bad habits. See Perl.
2. Basic Workflow
   1. Pipelining
      1. About
         1. PowerShell can be executed sequentially, or through the use of common control stuctures.
         2. Being workflow oriented, the KillerApp approach is to set up workflow Pipelines, moving objects through in one go.
         3. PowerShell pipelines pass objects to different commands, not plain text. (Though text can be an object.)
         4. OBJECTlivey better than other languages since PowerShell passes objects rather than text.
         5. Can stream data out in real time.
         6. Not all Cmdlets accept Pipeline input.
      2. Examples
         1. Use Select-Object to alter returned data via Pipeline.
      3. Additional Comments
         1. Pipeline is how PowerShell is designed to be used.
         2. Pipelines generally process in a stream.
         3. Continuous stream of data means lower memory requirements, but often slower execution.
         4. PowerShell sometimes lies. About\_Format.ps1xml .
         5. Default display can execute code, so consider performance implications.
   2. Filtering
      1. About
         1. Pipeline with Where-Object Cmdlet.
         2. $\_ / $PSItem automatic variable can be used in Pipelines to refer to the “current” object.
      2. Examples
         1. Get processes and filter for PowerShell.
         2. $PSItem vs $\_
         3. FilterScript vs Paramaeter -comparison Value
      3. Additional Comments
         1. Comparison Statement syntax (no Script Block) introduced in PowerShell 3.0.
         2. Comparison Statement syntax reads better but is limited.
         3. Multiple comparators and inline calculations require the full Script Block syntax.
   3. File Data
      1. About
         1. PowerShell provides several built-in Cmdlets for handling file I/O.
         2. They work fairly well if paradigms are respected.
      2. Examples
         1. Write a list of PowerShell processes to a file.
         2. Write a list of processes to a file, read back, and filter for PowerShell.
         3. Write a list of processes to a structured file, read back, and filter for PowerShell.
         4. Serialize a list of process to a file, read back, and filter for PowerShell.
         5. Create a Lookup Map from file data using a Hashtable.
      3. Additional Comments
         1. PowerShell types have default views which determine how they are output to a console by default. Be aware.
         2. Built-in Cmdlets have built-in magic that sometimes creates unexpected results. A key example is plain text files line to array conversion.
         3. Built-in magic adds overhead to native file I/O Cmdlets.
         4. Using .NET I/O API is faster (discussed later.)
         5. Deserialized objects may behave inconsistently.
   4. Filesystem
      1. About
         1. File and Directory manipulation is done though PowerShell’s generic Data Store Provider / Item model.
         2. Providers provide access to data stores, such as the file system, in a somewhat standard way.
      2. Examples
         1. Create, get, and delete a file.
         2. Create a file in the current directory, then list the contents of the current directory.
         3. Get an object reference to a file, then use the object to delete the file.
         4. List the contents of the current directory as file names, not full objects.
         5. List all files into a variable.
      3. Additional Comments
         1. PowerShell makes Filesystem manipulation very easy.
         2. Be wary of performance and memory usage since everything is object-based. A lot more than a simple “dir” is going on. Even if managed correctly Garbage Collection cycles may mean long stretches of high memory utilization.
         3. Be mindful of object and backing item linkages.
   5. Registry
      1. About
         1. Registry manipulation also uses the Provider / Item model.
         2. Registry Keys are Items.
         3. Registry Values are Properties of Keys.
      2. Examples
         1. Read Current User Auto-Run entries.
         2. Create an Auto-Run entry for Notepad.exe.
         3. Navigate the Registry like the filesystem.
      3. Additional Comments
         1. Only HKCU and HKLM are provided by default.
         2. Interactive, complex registry manipulation can be challenging due to the intersection of child objects and properties.
3. Scripting
   1. Execution Policy
      1. About
         1. Execution Policy defines when PowerShell will execute a script or block execution.
         2. Scopes are LocalMachine, CurrentUser, Process.
         3. The most specific scope’s policy wins.
      2. Examples
         1. Get and set the Execution Policy.
      3. Additional Comments
         1. Despite Microsoft’s claims Execution Policy isn’t a “security” feature. “Safety” is more apt.
         2. Article “15 Ways to Bypass the PowerShell Execution Policy” exists, so clearly it is not a real security feature.
   2. Conditionals and Flow Control
      1. About
         1. If / Then / Else
         2. Switch
         3. For / ForEach
         4. While / Do While
         5. Try / Catch / Finally
      2. Examples
         1. (Examples of each structure.)
      3. Additional Comments
         1. Try, Catch, Finally does support multiple Catch blocks.
         2. ForEach vs ForEachObject
            1. ForEach-Object is generally for Pipeline customization such as wrapping Cmdlets that do not accept Pipeline input.
            2. ForEach-Object allows for multiple statements per input object in a Pipeline.
            3. ForEach is an alias for ForEach-Object in interactive mode. There is no difference.
            4. ForEach in scripts behaves differently. Pipeline is broken but can still output.
            5. ForEach results in higher memory utilization than ForEach-Object (caching during broken Pipeline) but can be faster.
   3. Functions and Advanced Functions and Cmdlets
      1. About
         1. Functions allow compartmentalization of code for re-use.
         2. Advanced Functions mimic Cmdlets.
      2. Examples
         1. Basic Function
         2. Function with a Parameter (Inline declaration)
         3. Function with a Parameter (Block declaration)
         4. Function with Pipeline Input
         5. Advanced Function / Script Cmdlet
      3. Additional Comments
         1. Data written out during a function is the function’s output.
         2. Normal Functions are basic.
         3. Advanced Functions can be created by either using CmdletBinding or by using Parameter attributes.
         4. Begin, Process, and End blocks are required for pipeline input.
         5. Pipeline input can be very tricky. When in doubt, fall back to ValueFromPipelineByPropertyName .
         6. Get-Help about\_Functions\_Advanced\_Parameters
   4. Splatting
      1. About
         1. Pass a Hashtable of parameters instead of or in addition to explicitly defined parameters.
         2. Allows for run-time parameterization. We can define not just what the value of passed parameters are, but what parameters are passed.
         3. Avoids messy scripts with large sets If/Then statement blocks for different parameter sets to the same command.
         4. Main uses are accounting for Switch parameters and parameter sets.
      2. Examples
         1. Passing a parameter via Splatting.
         2. Adding a Switch parameter to an existing splatting Hashtable.
      3. Additional Comments
         1. $PSBoundParameters can be used with wrapper scripts for an intercept-and-pass-through model.
         2. Even though PowerShell passes by value be careful about Hashtable reuse in loops.
   5. Custom Objects
      1. About
         1. Custom objects can be created and used for better control over output.
         2. Custom objects can be streamed out of the pipeline in a collection.
         3. Outputting Hashtables yields a collection of Hashtables with keys which may match, but each Hashtable is still a rendered separately.
         4. Outputting a collection of PSObject yields clean, consistent PowerShell output.
      2. Examples
         1. Create a basic custom object with two properties.
      3. Additional Comments
         1. It is possible to update the Default Display Property Set to control what is shown on the console by default. (Remember Get-Process.)
         2. Additional member types including methods can be added if necessary.
   6. ArrayLists
      1. About
         1. PowerShell Arrays are immutable by default.
         2. Appending data to an Array means creating a third array from the two given ones. (Original, Addition, Target.)
         3. ArrayLists solve this problem.
      2. Examples
         1. Array versus ArrayList
      3. Additional Comments
         1. Scripts should try to continue moving objects through the pipeline, streaming them until the end.
         2. When that’s not desirable or possible, use ArrayLists.
         3. Be sure to consume or discard the output from [ArrayList].Add(). Output is the index of the element added.
4. Modules
   1. Finding, Loading and Using Modules
      1. About
         1. PowerShell code can be packaged and distributed in Modules.
         2. All Cmdlets are implemented in a Module.
         3. Snap-Ins are similar to Modules, but they are compiled .NET assemblies. Snap-Ins are deprecated.
         4. As of PowerShell 3.0 Modules will be auto-loaded.
      2. Examples
         1. List modules and module contents.
         2. Force Load and Unload a module.
         3. Use a Module Qualified Name.
      3. Additional Comments
         1. Combine Aliases, Module Qualified Names, and Splatting to make Cmdlet “hooks”. These can be used to implement utilities such as pluggable logging modules.
         2. Override or intercept modules by installing them in higher precedence locations. User Directory > Program Files > System .
         3. Module auto-loading can be disabled by setting $PSModuleAutoloadingPreference to ‘None’ .
   2. Authoring Modules
      1. About
         1. Authoring modules allows for clean, compatible distribution of related code.
      2. Examples
         1. Simple one-file module.
         2. Full module.
      3. Additional Comments
         1. PowerShell has easy support for multi-line string literals.
         2. Using Module Manifests is preferred as it allows for versioning.
         3. Use #Requires -Modules in scripts to force dependency on a module. Specific versions can be specified.
5. Code Execution
   1. Packaging and Execution
      1. About
         1. PowerShell code can be executed with similar means one would use for any other program or script.
         2. Special care should be taken to account for PowerShell’s feature set.
      2. Examples
         1. Basic invocation of code with powershell.exe.
         2. Execution of Base64 encoded scripts.
      3. Additional Comments
         1. Be aware of PowerShell output console rendering when invoking scripts though powershell.exe .
         2. Output often must be captured and converted to plain text before final output. See Out-String.
         3. EncodedCommand lets you execute code in-line via any job scheduling engine such as Task Scheduler, GPO Scripts, etc…
         4. For both patterns of invocation there is a limit of 8191 characters that can be passed to a new process. This includes any passed parameters and environment variables.
   2. Jobs
      1. About
         1. Jobs can be used to run tasks in the background.
      2. Examples
         1. Start, receive and remove jobs.
      3. Additional Comments
         1. Jobs spawn in a new child process.
         2. Jobs are still attached to the invoking PowerShell, so terminating it terminates the job.
         3. Jobs don’t have access to any data not explicitly passed to them.
6. Web Interaction
   1. Web Communication Cmdlets and Classes
      1. About
         1. PowerShell has several network communication Cmdlets.
         2. Mainly Invoke-WebRequest is used.
         3. .NET Web Classes can be used as well.
      2. Examples
         1. Perform a Google search with IWR.
         2. Download a file with Net.WebClient.
      3. Additional Comments
         1. Invoke-RestMethod is similar to Invoke-WebRequest, but deserializes the response differently. In-built support for XML and JSON.
         2. Systems without Internet Explorer (running Server Core) need to pass -UseBasicParsing in Invoke-WebRequest.
         3. Built-in Cmdlets don’t always perform well.
         4. When working with .NET Classes some PowerShell magic is absent (such as working directories.)
   2. Web Service Proxy
      1. About
         1. Easy access to WSDL-described Web Services
         2. Usually SOAP, often ASP.net, but not necessarily either.
      2. Examples
         1. GeoIP lookup.
      3. Additional Comments
         1. Good discoverability, generally easy use.
         2. Very low barrier to entry.
   3. Internet Explorer COM Object
      1. About
         1. Its possible to run and control Internet Explorer and other COM applications via Powershell.
         2. If you need to use a “real” browser instead of the Web cmdlets you can.
      2. Examples
         1. Navigate to Google and inspect data.
      3. Additional Comments
         1. Fat-clients with web-based interactive logins (SAML, 2FA, etc…) can be authenticated this way. Log in and retrieve OAuth token, for example.
7. Remoting
   1. PowerShell Remoting
      1. About
         1. PowerShell remoting uses Windows Remote Management to remotely instantiate PowerShells on other systems.
         2. Remote sessions are nearly indistinguishable from local sessions.
         3. Windows Remote Management (Web Services Management (SOAP over HTTP(S), XML)) -> WMI -> COM -> PowerShell Session
      2. Examples
         1. Create and enter a session interactively using the PSSesion Cmdlets.
         2. Execute a command remotely with Invoke-Command.
      3. Additional Comments
         1. Invoke-Command can be given an array for the ComputerName parameter.
         2. PowerShell Remoting may or may not be configured by default. Its variable based on operating system.
         3. In a Windows Domain, default non-TLS WinRM over HTTP connections are in fact encrypted with Kerberos tickets.
         4. Created Sessions can be passed to Invoke-Command.
         5. Session creation brings a lot of overhead. If you are sure there will be only one execution, use Invoke-Command. Otherwise, use New-PSSession.
   2. CIM
      1. About
         1. Not actually PowerShell.
         2. Used to interact with Windows Management Instrumentation.
         3. Interaction over WMI is lighter weight than full blown PowerShell sessions.
      2. Examples
         1. Remotely query services via CIM.
      3. Additional Comments
         1. Note the added PSComputerName property added with remote command invocation.
         2. CIM should be used over PowerShell for simple tasks at scale due to the lower overhead.
         3. CIM Cmdlets can be used locally to accomplish things PowerShell Cmdlets can’t do directly.
         4. CIM / WMI has been around on Windows since 1996 (in contrast to PowerShell 1.0’s 2006) so there is a lot of history and solved problems to take advantage of.
         5. Don’t use deprecated WMI Cmdlets. CIM Cmdlets can do the same and more.
8. .NET IO
   1. File
      1. About
         1. PowerShell file I/O cmdlets often try to be helpful.
         2. Unexpected behavior and poor performance may be noticed.
         3. Using .NET file I/O directly gives great performance and plenty of control.
      2. Examples
         1. .NET File I/O
         2. Random file I/O
         3. Read a file into memory and compress using streams.
      3. Additional Comments
         1. PowerShell “magic” might not be available. Provide full paths, etc…