PowerShell Crib Notes

By George Squillace

See SolarWinds webinar notes at the end of this.

Activity {Help and Searching for cmdlets}	**Code and Notes**
Determine version, edition (i.e., Core, Desktop) of Windows PowerShell on a computer.	**`$PSVersionTable`**
Installing various versions of PowerShell (product documentation)	https://aka.ms/pscore6get The traditional PowerShell executable is **`PowerShell.exe`**. The PowerShell Core executable is **`Pwsh.exe`**. PowerShell and Core can co-exist.
Getting help.	**`Get-Help`** displays all help content. **`Get-Help Get-ChildItem`** displays all help content **`Get-Help *process*`** It accepts wildcards for cmdlet names. **`Get-Help Stop-Process -Examples`** **`Get-Help Stop-Process -Full`** to see the most complete help information for that cmdlet **`-Show Window`** help shows up in a separate window **`-Online`** Online version of the Help topic **`-Parameter ParameterName`** **`-Category`** Displays help only for certain categories of commands **`Get-Help about*`** Provides a huge list of "about_" help topics regarding the PowerShell scripting language, operators and more (global shell techniques and features). You can then reference one of those help topics, such as: **`Get-Help about_Workflows`**
Update help	**`Update-Help`**
Download help to an alternate, user-specified location	**`Save-Help`**
Getting the name of cmdlet.	**`Get-Command`** Lists every installed cmdlet, alias, function, filter, script, and application installed on the computer, and retrieves information such as name, category, version, and even the module that contains it.

	You can also reverse this and find out what *Module* a cmdlet is hosted within. **`Get-Command -Type cmdlet`** ***`CmdletName`*** **`Get-Command`** ***`SomeCommandName`*** imports the module that contains the command. **`Get-Command *process*`** Wildcards also supported. Lists every cmdlet with “process” in the name. Also: **`Find-Command`**

Filter portions of the cmdlet name using based on an attribute of the cmdlet	**`-Module`** ***`ModuleName`*** Lists only cmdlets in the specified Module
	`-Noun`
	`-Verb`
	`Get-Command -Noun event* -Verb Get`

Comment code	Use **`#`** for a single line comment and **`<# stuff #>`** as a block comment

Activity {Modules}	Code and Notes
Modules are groups of cmdlets that are packaged together, a container of sorts.	To use a cmdlet within a Module the Module must be *imported* (loaded)

Retrieve list of imported Modules in the current session or that can be imported from the PSModulePath	**`Get-Module`**
	`Get-Module -ListAvailable`
	`Get-Module -Name` ***`ModuleName`***

Load a Module into memory (after it has been installed)	**`Import-Module`** ***`Modulename`***
Update a Module	**`Update-Module -Name`** ***`ModuleName`*** A bunch of other parameters exist.
The PowerShell Gallery	The central repository for sharing and acquiring PowerShell code including PowerShell modules, scripts, and DSC resources. **https://powershellgallery.com** The PowerShell Gallery uses **`PowerShellGet`** Module which contains cmdlets for finding and installing modules, scripts and commands. Use **`Get-InstalledModule`** to get a list of modules on the computer that were installed by **`PowerShellGet`**. You may have to run **`Set-PSRepository`** to trust **`PowerShellGet`** as a repository. Another source of code is the Script Center
To search for modules within the PowerShell Gallery	**`Find-Module`**
To search for scripts within the PowerShell Gallery	**`Find-Script`**
Install a Module You can also uninstall a Module, but not referenced in this document.	**`Install-Module -Name`** ***`ModuleName`***
A number of cmdlets have *aliases*.	**`Get-Alias`** Returns all defined aliases. **`Get-Alias di*`** Wildcards are available.

Aliases can be added and removed, but, they are not saved between PowerShell sessions.cls	**`New-Alias -Name “`*`AliasName`*`”`**
	`morestuff_as_necessary`
	
	`Get-Alias -definition Remove-Item` reverses the process and find an alias (or multiple aliases) assigned to a cmdlet.

To open a window that displays either a list of commands or the parameters for a specific command.	**`Show-Command -Name cmdlet`*`Name`*** You can then supply any values in the form that apply and/or are required and click on the “Run” or “Copy: buttons.

Activity {Module Discovery}	Code and Notes
This lists some of the Module categories for Windows administration	
Active Directory	• User Management • Group Management • Computer Object Management • OU Object Management • GPO Management • Various AD object Management areas o Move / New / Set / Remove / Sync…
Networking	• Managing IP Addresses • Managing Routing • Managing DNS Clients • Managing Windows Firewall
Server Manager	Windows Features
Hyper-V	Many cmdlets available
IIS Management	

Activity {Pipelines}	Code and Notes
A pipeline is one or more commands. Consider that each cmdlet/command is its own pipeline.	Typical patterns are: **`Get | Set`**, **`Get | Where`**, or **`Select | Set`** "Where" is an alias for **`Where-Object`** and "Select" is an alias for **`Select-Object`**.
The output of a cmdlet is a collection of objects, or *Object* for short.	The structure of an Object is sort of like a table. Each row is an "Object". Each column is a "Property" of the Object. The rows are called a "Collection".
Members are the various components of an Object and include:	• Properties • Methods (perform some kind of action) • Events The default on-screen output does not include all of an Object's properties as some Objects have *hundreds*.
To list the Members of an Object:	**`Get-Member`** Lists all Properties, Methods, and Events (alias, **`gm`**) **`Get-Service | Get-Member`**
To page the output of a cmdlet:	Use **`| More`** which shows only one page of output, which does not work within the PowerShell ISE, only the console. A substitute to **`| More`** is **`Out-Host`**. From there hit: Spacebar pages through the output. Enter progresses through the output one line at a time.
Formatting Pipeline output. Cmdlets have a default output and these cmdlets can override the default output.	**`| Format-List`** (alias, **`fl`**), each Property on a separate line for each Object. Each object will have its own list. Particularly useful when a command returns a large number of Properties that would be hard to read in table format. **`| Format-Table`** Useful for displaying Properties of many Objects at the same time and comparing the Properties. Also provides **`-AutoSize`**, **`-HideTableHeaders`**, and **`-Wrap`**. **`| Format-Wide`** two column, single Property output. (alias, **`fw-all`**), **`| Format-Custom`** requires creation of custom XML files

	Each of these accepts the **`-Property`** parameter where one supplies a comma-separated list of property names.

Activity {Selecting, Sorting, Measuring}	**Code and Notes**

Override the default sort output	**`| Sort-Object`** two column, single Property output. (alias, **`sort`**,
	
	`Get-Service | Sort-Object -Property Name - Descending`
	
	`Get-Service | Sort Name – Desc`
	
	`Get-Service | Sort Status, Name`
	
	`Get-Service | Sort-Object Status, Name |fw -GroupBy Status`
	
	See also **`Group-Object`** which gives more grouping control.
	
	Other parameters apply.

Aggregations can be applied to output	**`Get-ChildItem -File  | Measure -Property Length –Sum –Average -Minimum -Max`**

Select a *subset* of the Objects {**rows**} passed along the pipeline using **`Select-Object`**	**`Get-Process | Sort-Object -Property VM | Select-Object -First 10`**
	
	`Get-Process | Sort-Object -Property CPU -Descending | Select-Object -First 5 -Skip 1`
	
	`First | Last | Unique | Skip | Index | Unique`
	
	(alias, **`select`**)
	
	Sort-Object is used to control the object order *prior* to object selection.

Select a specific set of Object *Properties* {**columns**} passed along the pipeline using **`Select-Object`** which requires **`-Property`**	**`Get-Process | Select-Object -Property Name, ID, VM, PM, CPU | Format-Table`** comma delimited list of Properties **`Get-Process | Sort-Object -Property CPU -Descending | Select-Object -Property Name, CPU -First 10`** the ten processes using the most CPU.
When using the PowerShell console you can break up a command over multiple lines and obtain an "extended prompt"	...then hit ENTER when the code is complete.
Calculated columns through hash tables	

Activity {Controlling Pipeline output with Comparison Operators}	Code and Notes
	The operators are case insensitive with strings. Prefix each of the operators below with "**c**", like **`-ceq`** for a case sensitive version.
Equal to	**`-eq`**
Not equal to	**`-ne`**
Greater than	**`-gt`**
Less than	**`-lt`**
Less than or equal to	**`-le`**
Greater than or equal to	**`-ge`**
For pattern matching use the **`-Like`** (or **`-cLike`**) operator with wildcards **`?`** and **`*`**	

There are other, more advanced operators	**`-in`**
	`-contains`
	`-as`
	`-match and -cmatch`

Another type of filtering can be accomplished with **`Where-Object`** (alias **`Where`**) There's a basic (single comparison) and advanced form of usage. There's an extensive parameter list.	**`Get-Service | Where Status -eq Running`**

The advanced usage requires a *filter script*.
 	
The filter script runs one time for each Object that is piped into the command.
	`$PSItem` (or **`$_`** earlier versions of PowerShell) is a special variable created by Windows PowerShell. It represents whatever Object is piped into the **`Select-Object`** command.
	
	The following are four equivalent commands:
	
	`Get-Service | Where Status -eq Running`
	
	`Get-Service | Where-Object -FilterScript { $PSItem.Status -eq 'Running' }`
	
	`Get-Service | Where { $PSItem.Status -eq 'Running' }`
	`Get-Service | ? { $_.Status -eq 'Running' }`
	
	Combining criteria (each expression criterion must be complete):
	
	`Get-EventLog -LogName Security -Newest 100 |`
	`Where { $PItem.EventID -eq 4672 -and $PSItem.EntryType -eq 'SuccessAudit' }`
	
	If the Property interrogated is already Boolean other operators are available.
	
	`Get-Process | Where { $PItem.Responding -eq $True }` or
	`Get-Process | Where { $PItem.Responding }`
	`Get-Process | Where { -not $PItem.Responding }` negation

Activity {Enumerating Objects in the Pipeline}	**Code and Notes**
Definition: *Enumeration* is the process of performing a task on each Object, one at a time, in a collection.	Enumeration is not always required, as seen below… **`Get-Process -Name Notepad | Stop-Process`** or **`Stop-Process -Name Notepad`**

The **`ForEach-Object`** command performs enumeration.
 	
 There is a basic and advanced syntax. Basic syntax can only access a single property.
 	
 The requirement for enumeration seems to be reduced with newer PowerShell commands
	Two common aliases are **`ForEach`** and `%`.
	
	 Three equivalent commands…(The **`-File`** Object type, System.IO.FileInfo has a method named **`Encrypt`**)
	
	 `Get-ChildItem -Path E:\Data -File | ForEach-Object -MemberName Encrypt`
	
	 `Get-ChildItem -Path E:\Data -File | ForEach Encrypt`
	
	 `Get-ChildItem -Path E:\Data -File | % Encrypt`

The output of a cmdlet is a collection of objects, or *Object* for short.	The structure of an Object is sort of like a table. Each row is an “Object”. Each column is a “Property” of the Object. The rows are called a “Collection”.

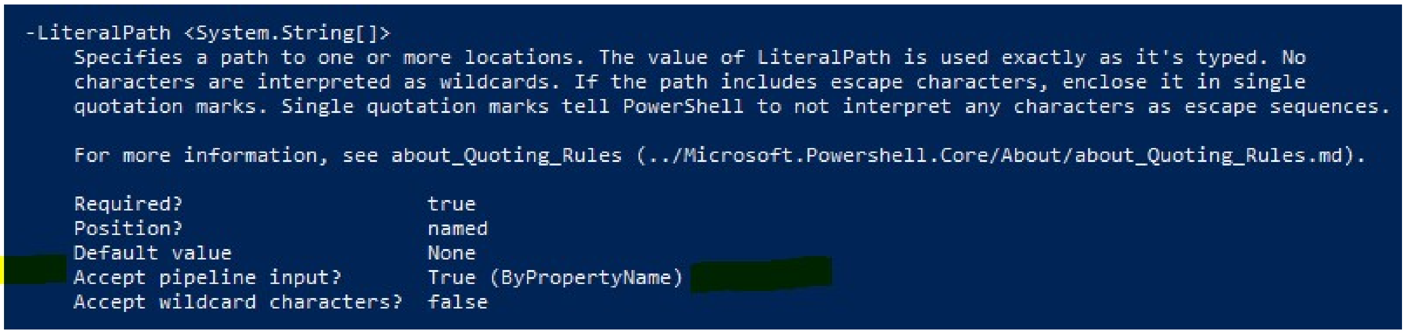
Activity {Sending Pipeline data as output}	Code and Notes

Output to a File	**`Out-File`** (aliases **>** and, **>>**) accepts input from the pipeline. The output appears just as it would on screen. This is not the same as converting or exporting objects. This option is mostly for human consumption and *not* for reading back into PowerShell.
	
	This terminates the pipeline.
	
	> directs to a file, *overwriting* if it already exists
	>> *appends* to an existing file.
	
	`Get-Service |`
	`Sort-Object -Property Status, Name |`
	`Select-Object -Property DisplayName, Status |`
	`Out-File -FilePath ServiceList.csv`

Convert Output to CSV	Converts the output to CSV and the data *remains* in the pipeline. CSV output produces column headers. There is also an **`Import-CSV`** cmdlet.
	
	`Get-Service | ConvertTo-CSV | Out-File Services.csv`
	
	`Export` commands, like **`Export-CSV`**, perform two operations: converting the data and then writing to external storage.
	
	`Get-Service | Export-CSV Services.csv`

Convert Output to XML	**`ConvertTo-CliXML`** .The data *remains* in the pipeline. **`Export-CliXML`**
Convert Output to JSON	**`ConvertTo-JSON`** .The data *remains* in the pipeline.
Convert Output to HTML	**`ConvertTo-HTML`** .The data *remains* in the pipeline. Output can be controlled along with: **`-Head`** **`-Title`** **`-PreContent`** content to appear before the table or list **`-PostContent`** content to appear after the table or list
Additional output options	**`Out-Host`** more output control such as with **`-Paging`** **`Out-Printer`** sends output to your default printer. **`Out-Gridview`** interactive window that allows you to sort, filter and copy (but not save).

Activity {Understanding How the Pipeline Works}	Code and Notes
Pipeline parameter binding	This command… **`Get-ADUser -Filter {Name -eq 'Dan DeLion'} | Set-ADUser -City Seattle`** …the **`Set-ADUser`** cmdlet actually has two parameters passed to it, the output of Get-ADUser and **`-City`**. "When you connect two commands in the pipeline, pipeline parameter binding takes the output of the first command and decides what to do with it. The process selects one of the parameters of the second command to receive that output. Windows PowerShell has two techniques that it uses to make that decision. **`ByValue`** always attempted first. **`ByPropertyName`** used only when **`ByValue`** fails. The ability to accept pipeline output is part of the definition of the parameter within the cmdlet code. Use **`Get-Help`** ***`whatevercmdlet`*** **`-Full`** …to determine the pipeline input capability of each parameter. See screenshot below. A single command can have more tha one parameter accepting pipeline input x but each parameter must accept a different kind of object.
	There's more content to this module.



```
-LiteralPath <System.String[]>
    Specifies a path to one or more locations. The value of LiteralPath is used exactly as it's typed. No
    characters are interpreted as wildcards. If the path includes escape characters, enclose it in single
    quotation marks. Single quotation marks tell PowerShell to not interpret any characters as escape sequences.

    For more information, see about_Quoting_Rules (../Microsoft.Powershell.Core/About/about_Quoting_Rules.md).

    Required?                    true
    Position?                    named
    Default value                None
    Accept pipeline input?       True (ByPropertyName)
    Accept wildcard characters?  false
```

Activity {Using PSProviders and PSDrives}	Code and Notes
A PSProvider, or *Provider* presents data as a hierarchical store.	In managing IIS I could use an IIS specific cmdlet, like **`Get-WebSite`** or a more generic command like **`Get-ChildItem IIS:\Sites`**
To list the available providers, and, the capabilities of each provider	**`Get-PSProvider`** lists the providers in the current session, their capabilities, and their "drives". **`Get-PSProvider`** ***`SomeProvider`*** lists the above details for the specified Provider.
Viewing help of a Provider	**`Get-Help`** ***`ProviderName`***
A *PSDrive*, or *Drive* is a connection to a data store	**`Get-PSDrive`** to see a list of available drives. **`New-PSDrive`** used to create temporary and persistent mapped network drives. Drive names do not include a colon Drives contain *Items*. Items contain child items and properties.
These drives are *always available*	Registry drives **`HKLM`** and **`HKCU`** Local hard drives like **`C`** PowerShell storage drives **`Variable`**, **`Function`**, and **`Alias`** Web Services for Management (WS-Management) **`WSMan`** Environment Variables **`Env`** Certificate Store **`Cert`**
Verbs associated with **`PSDrive`** cmdlets	**`New`** **`Set`** **`Get`** **`Clear`** **`Copy`** **`Move`** **`Remove`** **`Rename`** **`Invoke`**
Commands that manage PSDrive locations	**`Get-Location`** displays the current working location **`Set-Location`** sets the current working location **`Push-Location`** adds a location to the top of a location stack **`Pop-Location`** changes the current location to the location at the top of a location stack

To determine the alias mappings one would translate from a command prompt to PowerShell use **`Get-Alias`** or **`Get-Command`**.
 	
PowerShell accepts both **`\`** and **`/`** as path separators, so beware that **`Dir /s`** would not recurse
	`Dir   Dir /s = Get-ChildItem -Recurse`
	`Move`
	`Ren`
	`RmDir`
	`Del`
	`Copy`
	`MkDir`
	`CD`

Relevant drive commands	**`New-Item`**
	`Remove-Item`
	`Get-Item   Get-Item * = Get-ChildItem`

Activity {Using CIM and WMI}	Code and Notes

	`Get-WMIObject`
	`Get-WMIMethod`
	`Get-CIMClass`
	`Get-CIMInstance`

	`Invoke-WMIMethod` `Invoke-CIMMethod`

Activity {Using Variables}	**Code and Notes**
The main limitation of the pipeline is that the process flows only in one direction and it is diffult to perform complex operations. Variables solve this problem.	Ideas: • Store the name of a log file that you write data to multiple times • Derive and store an email address based on the name of a user account • Calculate and store the date of a day 30 days prior to the current day, to identify whether computer accounts have signed in during the last 30 days.
Variables can hold simple data types like strings and numbers and also *Objects*	There is **`PSDrive`** called **`Variable`** and use that with **`Get-ChildItem`** like: **`Get-ChildItem`** ***`VariableName:`*** to view Variable names, or, **`Get-Variable`**
Variable name rules and conventions	It's common to prefix PowerShell variable names with "**`$`**", but only to make them easier to identify. If variable names include a space the variable name must be enclosed in braces, like **`${Log File}`**

Assign a value to a variable with the = operator and you can also assign command output (single or multiple values) to a variable.
 	
You can also use **`Set-Variable`**
	`$Num = 10`
	`$LogFile = "C:\Logs\log.text"`
	`$User = Get-ADUser Administrator`
	`$Service = Get-Service W32Time`

Show the value of a variable by typing its name and hitting ENTER	**`$User`** ENTER
To empty a variable	***`$SomeVariable`*** **`= $null`**
Variable data types determine available manipulations	Most of the time PowerShell automatically assigns a data type and that mostly works. Some available data types are: **`String`** **`Int`** 32 bit integer **`Double`** 64-bit floating point value, for decimals **`DateTime`** **`Bool`** stores values of **`$true`** and **`$false`**.

Force a variable to accept only specific type of content (limitations apply)	**`[Int]$Num2 = "5"`**
	`[DateTime]$date = "January 5, 2020 11:49AM"`

View a variables' type	**`$date.GetType()`**

Activity {Manipulating Variables}	**Code and Notes**
Variables, just as objects, have properties and methods	
The simplest method for identifying the properties and methods available for a variable is to pipe the variable to **`Get-Member`**	**`$LogFile | Get-Member`**
To browse through the properties and methods for a variable, you can use tab completion by typing the name of the variable appended with a dot. When you press Tab, the properties and methods available for the variable display.	For more information on .NET Framework variable types, refer to "System Namespace" at: **`https://aka.ms/krlgav`**
String variables have only one property, **`Length`**	**`$LogFile.Length`**

The following are *some* of the string variable *methods*.	**`Contains(string value)`**
	`Insert(int startindex, string value)`
	`Remove(int startindex, int count)`
	`Replace(String value, string value)`
	`Split(Char separator)`
	`ToLower()`
	`ToUpper()`

Date Properties	**`Hour`** **`Minute`** **`Second`** **`TimeOfDay`** **`Date`** **`DayOfWeek`** **`Month`** **`Year`**

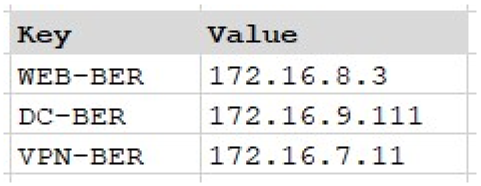
Date Methods	**`AddDays(double value)`**
	`AddHours(double value)`
	`AddMinutes(double value)`
	`AddMonths(int months)`
	`AddYears(int value)`
	`ToLongDateString()`
	`ToShortDateString()`
	`<and more>`
	
	`$date = Get-Date`
	`$date`
	`$date.DayOfWeek`

Activity {Arrays and Has Tables}	**Code and Notes**
Think of an Array as a Variable that contains multiple values or objects.	A Variable has one value; an Array has more than one value.

One way to create an Array is to create a comma-separated list.	**`$Computer = "DC-BER", "WEB-BER", "VPN-BER"`**
	`$Numbers = 787, 2.71, 3.14`

Another way to create an Array is from the output from a command.	**`$Users = Get-ADUser -Filter *`**
	`$Files = Get-ChildItem Z:\`

Verify whether a variable is an Array by using the **`GetType()`** method	If true the **`BaseType`** listed will be **`System.Array`**.
You can create an empty Array for when you have a loop that adds items to the Array	**`$NewUsers = @()`**
You can also force an Array to be created when adding a single value to a variable.	**`[array]$Computers = "WEB-BER"`**
Working with Arrays	**`$Databases`** (displays all array items) **`$Databases[0]`** (displays an item by its index number, starting at 0 **`$Databases = $Databases + $SomeOtherDB`** (adds a new item to the array) To identify what you can do with the content in an array use **`Get-Member`** such as: **`$SQLAgentJobs | Get-Member`** To view the properties and methods available for an array rather than its items **`Get-Member -InputObject  $SQLServers`**
Working with Array Lists (for when you have to manipulate an array with a large amount of members)	**`[System.Collections.ArrayList]$computers = "Srv1", "Srv2"`** <There are more details to this topic.>

Working with Hash Tables	Similar to an Array, as it stores multiple items. *Unlike* an Array a *Hash Table* uses a unique key for each item. With a Hash Table as follows (which stores *both* computer names and IP addresses):  Key — Value WEB-BER — 172.16.8.3 DC-BER — 172.16.9.111 VPN-BER — 172.16.7.11 …you access the first item in the hash table either of the two following ways: **`$servers.'VPN-BER'`** (the hyphen is a special character, which requires single quotes around the key) **`$servers['VPN-BER']`**
Define a Hash Table	Similar to an Array, but you need both the key *and* the value. • Begins with @ • Keys and associated values are enclosed in braces • Items separated by a semicolon (when multiple items are on the same line) **`$servers =@{"VPN-BER" = "172.16.7.11"; "DC-BER" = "172.16.9.111"}`**

Adding and removing items is similar to an Array List	**`$servers.Add("CertSrv-BER", "172.16.5.5")`**
	`$servers.Remove("VPN-BER")`

You can also update the value for a key	**`$servers.'Web-BER'= "172.16.3.71")`**
To view all properties and methods available for a Hash Table	**`$server | Get-Member`**

Activity {Basic Scripting}	**Code and Notes**
	Script files have a .ps1 extension
You have three options when you right click on a PowerShell script:	• Open (in Notepad) • Run with PowerShell (the PowerShell prompt closes on completion) • Edit (opens the script in the PowerShell ISE)
You have three options when you want to run a PowerShell script from the PowerShell prompt	• **`E:\Scripts\Coolscript.ps1`** full path to the script • **`\Scripts\Coolscript.ps1`** relative path to the script • **`.\Coolscript.ps1`** reference the current directory
To control whether or not PowerShell scripts can be run on a computer you set the *Execution Policy* with these options:	• **`Restricted`** no scripts can be run • **`AllSigned`** scripts will only if they're digitally signed • **`RemoteSigned`** downloaded scripts will run only if digitally signed • **`Unrestricted`** all scripts run but with a confirmation prompt • **`ByPass`** all scripts are run without prompts

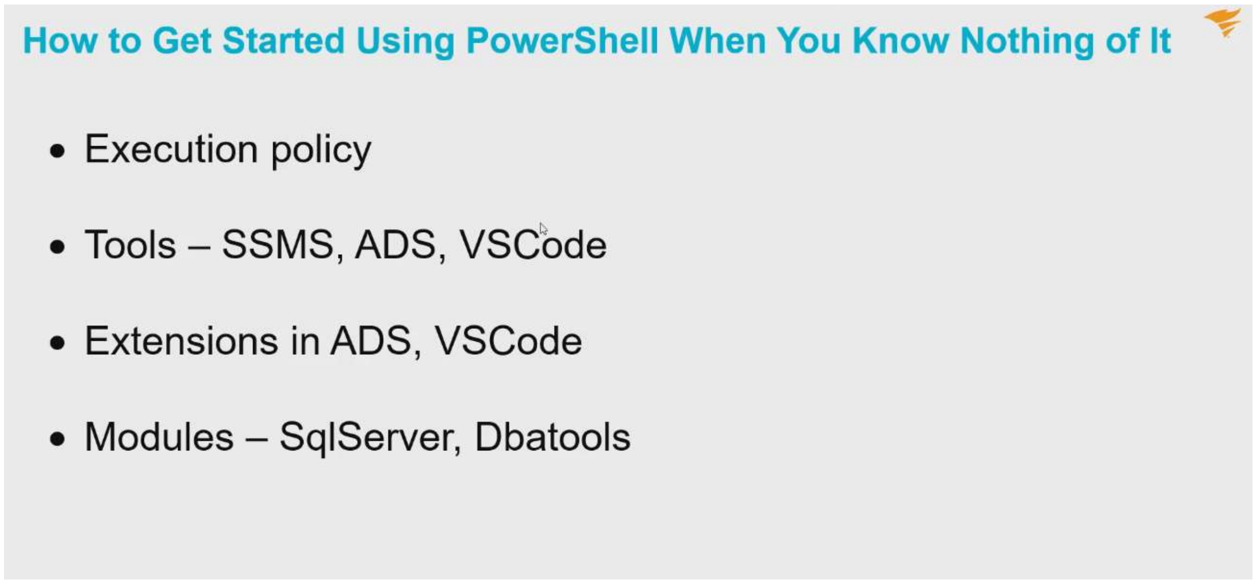
Activity {Advanced Scripting}	Code and Notes

Activity {Using Background Jobs and Scheduled Jobs}	Code and Notes

Activity {Using Advanced Windows PowerShell Techniques}	Code and Notes

SolarWinds Webcast on 2022-08-17

Ben Miller's blog post is at DBADuck.com

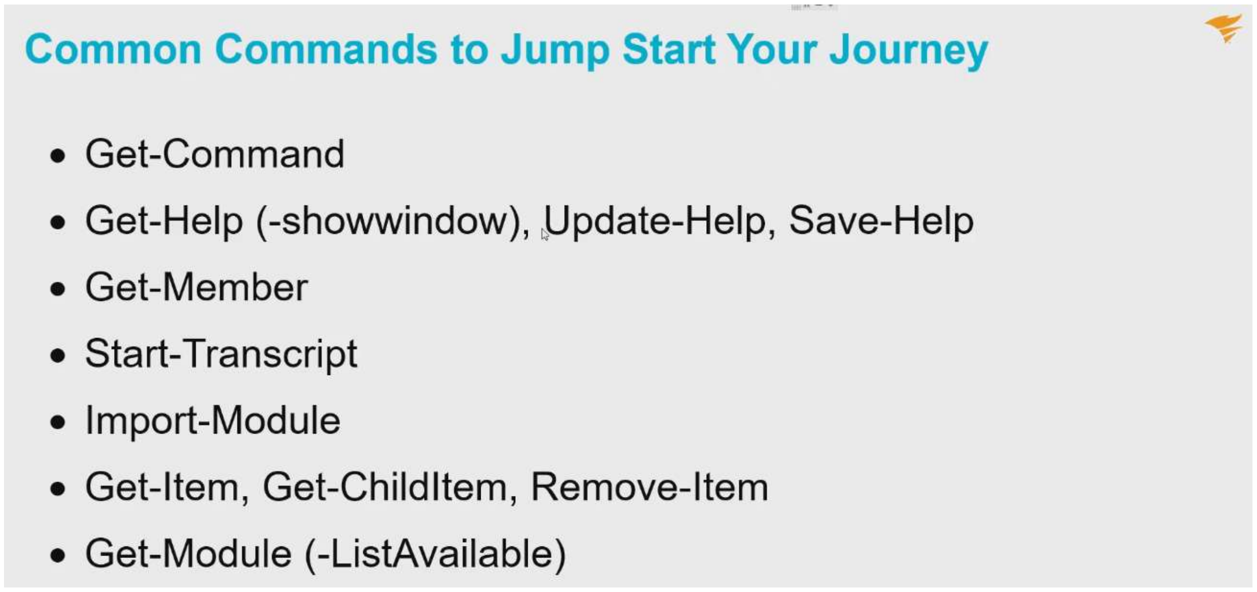


How to Get Started Using PowerShell When You Know Nothing of It

- Execution policy
- Tools – SSMS, ADS, VSCode
- Extensions in ADS, VSCode
- Modules – SqlServer, Dbatools

We'll also add the "importexcel" module.

For Execution Policy you will want "Remote Signed".

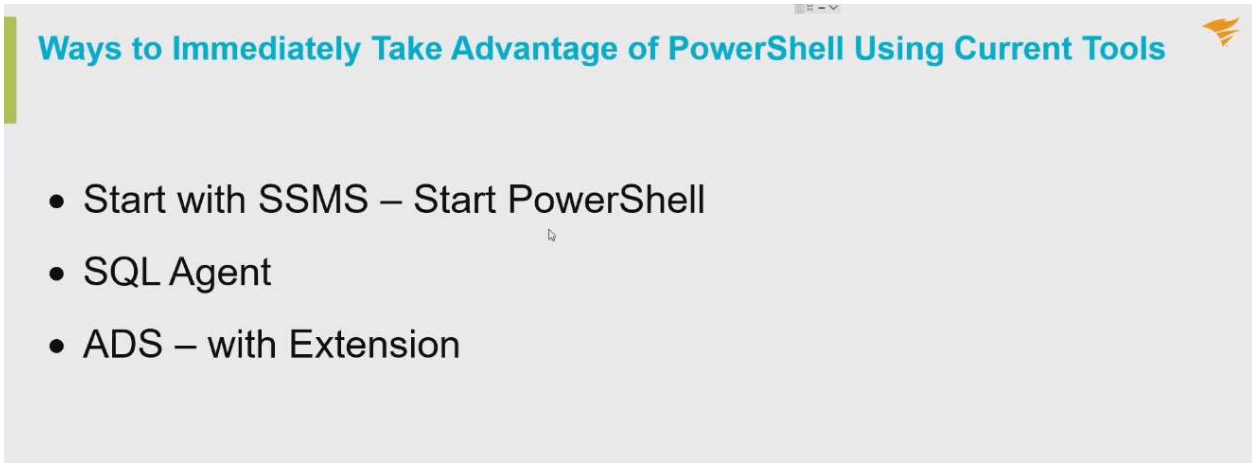


Common Commands to Jump Start Your Journey

- Get-Command
- Get-Help (-showwindow), Update-Help, Save-Help
- Get-Member
- Start-Transcript
- Import-Module
- Get-Item, Get-ChildItem, Remove-Item
- Get-Module (-ListAvailable)

You can save help files to a folder and then transfer that content.

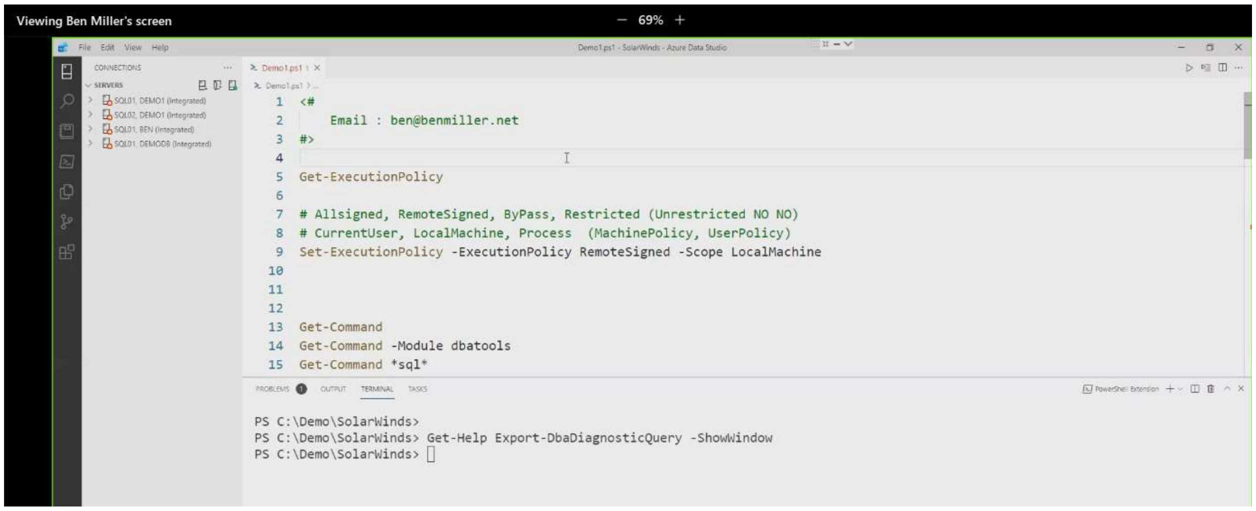
Transcripts allow you to go "headless". The opposite is Stop-Transcript. An input parameter to Start-Transcript is a path.



Ways to Immediately Take Advantage of PowerShell Using Current Tools

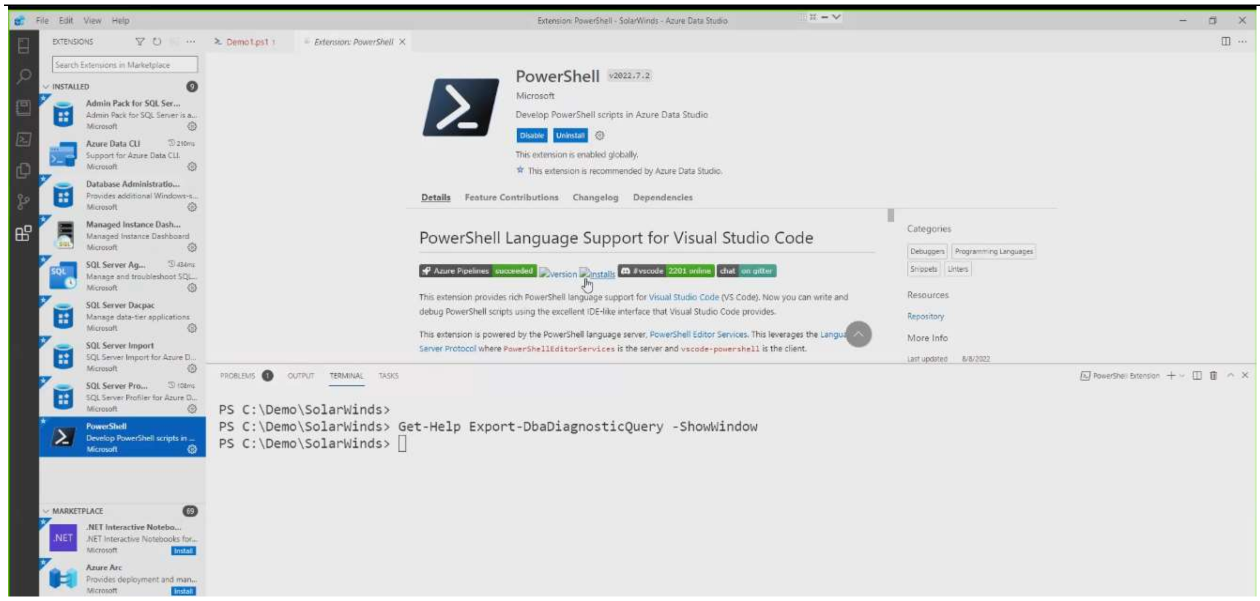
- Start with SSMS – Start PowerShell
- SQL Agent
- ADS – with Extension

Below – Azure Data Studio


Viewing Ben Miller's screen
69%
Demo1.ps1 - SolarWinds - Azure Data Studio
CONNECTIONS
SERVERS
SQL01, DEMO1 (Integrated)
SQL02, DEMO1 (Integrated)
SQL01, BEN (Integrated)
SQL01, DEMODB (Integrated)
Demo1.ps1
PROBLEMS
OUTPUT
TERMINAL
TASKS
PowerShell Extension

```
<#
    Email : ben@benmiller.net
#>

Get-ExecutionPolicy

# Allsigned, RemoteSigned, ByPass, Restricted (Unrestricted NO NO)
# CurrentUser, LocalMachine, Process  (MachinePolicy, UserPolicy)
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine



Get-Command
Get-Command -Module dbatools
Get-Command *sql*
```

```
PS C:\Demo\SolarWinds>
PS C:\Demo\SolarWinds> Get-Help Export-DbaDiagnosticQuery -ShowWindow
PS C:\Demo\SolarWinds>
```



PowerShell v2022.7.2

Microsoft

Develop PowerShell scripts in Azure Data Studio

This extension is enabled globally.

This extension is recommended by Azure Data Studio.

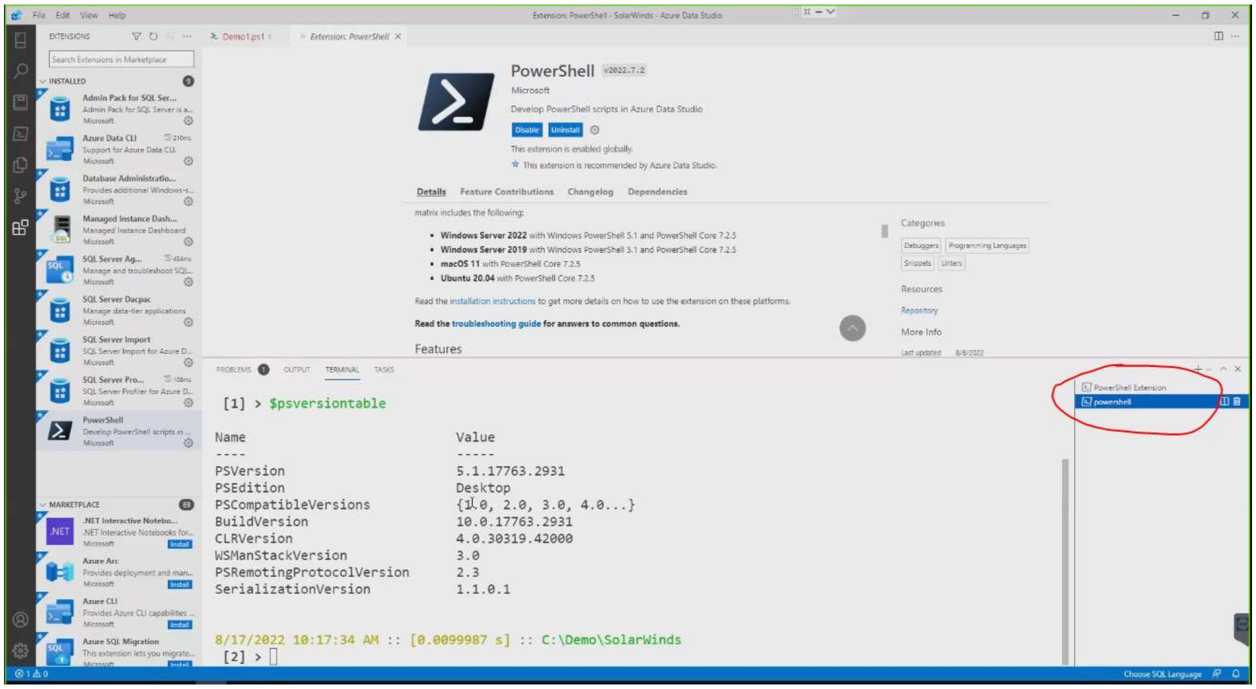
PowerShell Language Support for Visual Studio Code

This extension provides rich PowerShell language support for Visual Studio Code (VS Code). Now you can write and debug PowerShell scripts using the excellent IDE-like interface that Visual Studio Code provides.

This extension is powered by the PowerShell language server, PowerShell Editor Services. This leverages the Langua Server Protocol where `PowerShellEditorServices` is the server and `vscode-powershell` is the client.

```
PS C:\Demo\SolarWinds>
PS C:\Demo\SolarWinds> Get-Help Export-DbaDiagnosticQuery -ShowWindow
PS C:\Demo\SolarWinds>
```

Different shells available [below]



matrix includes the following:

- **Windows Server 2022** with Windows PowerShell 5.1 and PowerShell Core 7.2.5
- **Windows Server 2019** with Windows PowerShell 5.1 and PowerShell Core 7.2.5
- **macOS 11** with PowerShell Core 7.2.5
- **Ubuntu 20.04** with PowerShell Core 7.2.5

Read the installation instructions to get more details on how to use the extension on these platforms.

Read the troubleshooting guide for answers to common questions.

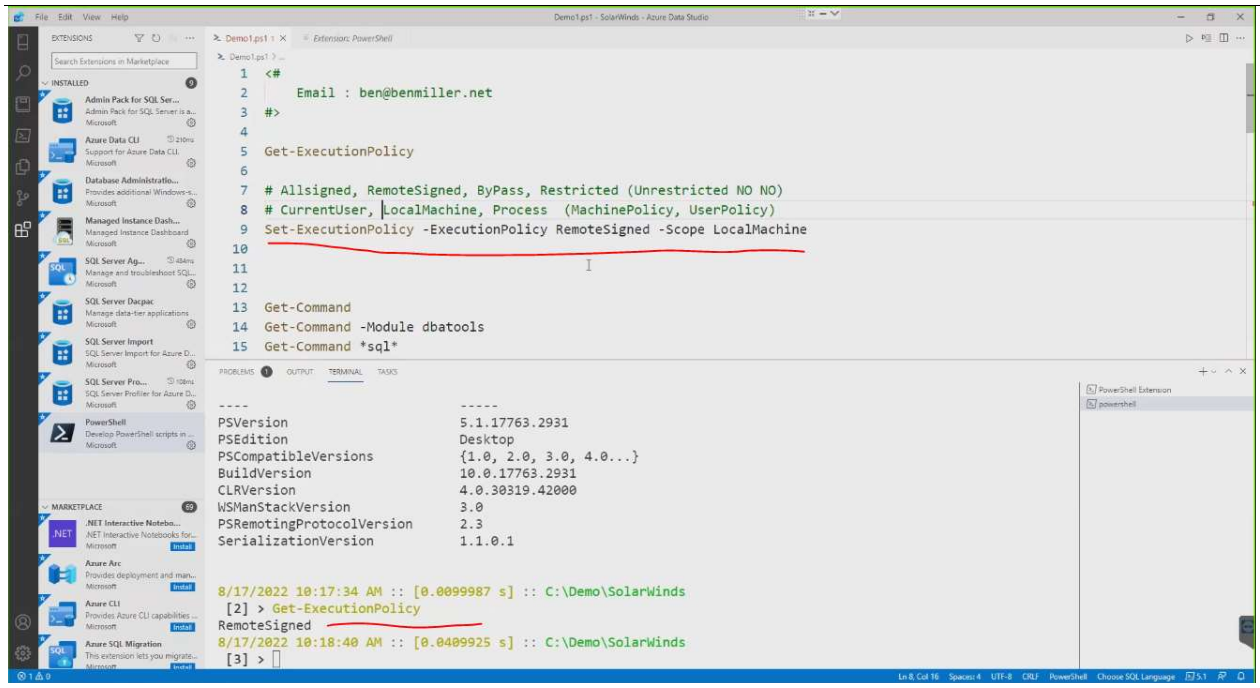
Features

```
[1] > $psversiontable

Name                           Value
----                           -----
PSVersion                      5.1.17763.2931
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.17763.2931
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1


8/17/2022 10:17:34 AM :: [0.0099987 s] :: C:\Demo\SolarWinds
[2] >
```

PowerShell ISE is deprecated…memory leaks, etc.

PowerShell Crib Notes

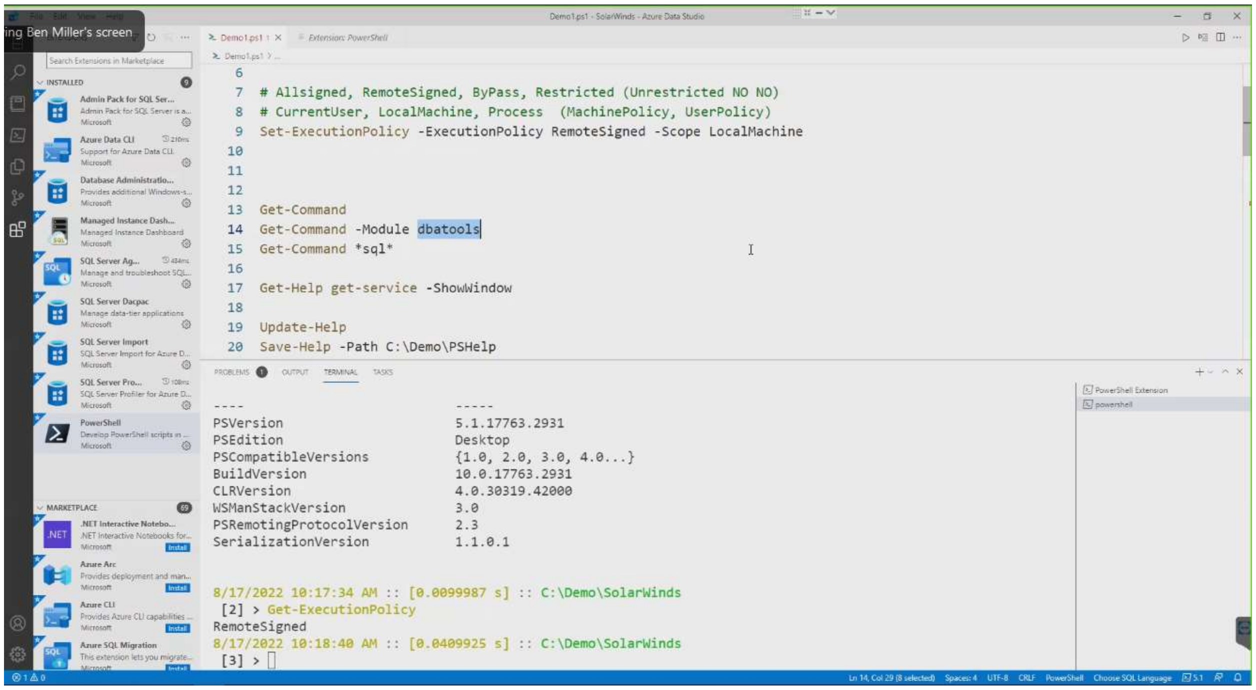
By George Squillace



```
<#
    Email : ben@benmiller.net
#>

Get-ExecutionPolicy

# Allsigned, RemoteSigned, ByPass, Restricted (Unrestricted NO NO)
# CurrentUser, LocalMachine, Process  (MachinePolicy, UserPolicy)
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine



Get-Command
Get-Command -Module dbatools
Get-Command *sql*
```

```
----                           -----
PSVersion                      5.1.17763.2931
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.17763.2931
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1


8/17/2022 10:17:34 AM :: [0.0099987 s] :: C:\Demo\SolarWinds
 [2] > Get-ExecutionPolicy
RemoteSigned
8/17/2022 10:18:40 AM :: [0.0409925 s] :: C:\Demo\SolarWinds
 [3] >
```

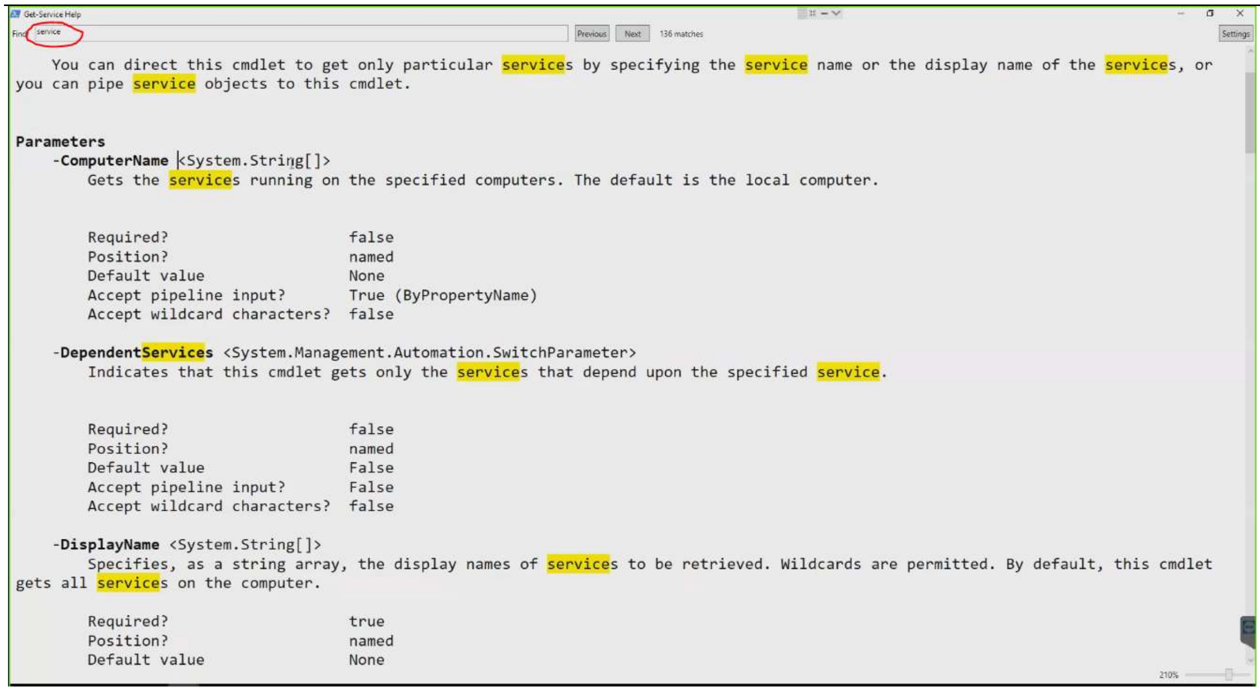
Don’t use “Unrestricted”



```

# Allsigned, RemoteSigned, ByPass, Restricted (Unrestricted NO NO)
# CurrentUser, LocalMachine, Process  (MachinePolicy, UserPolicy)
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine



Get-Command
Get-Command -Module dbatools
Get-Command *sql*

Get-Help get-service -ShowWindow

Update-Help
Save-Help -Path C:\Demo\PSHelp
```

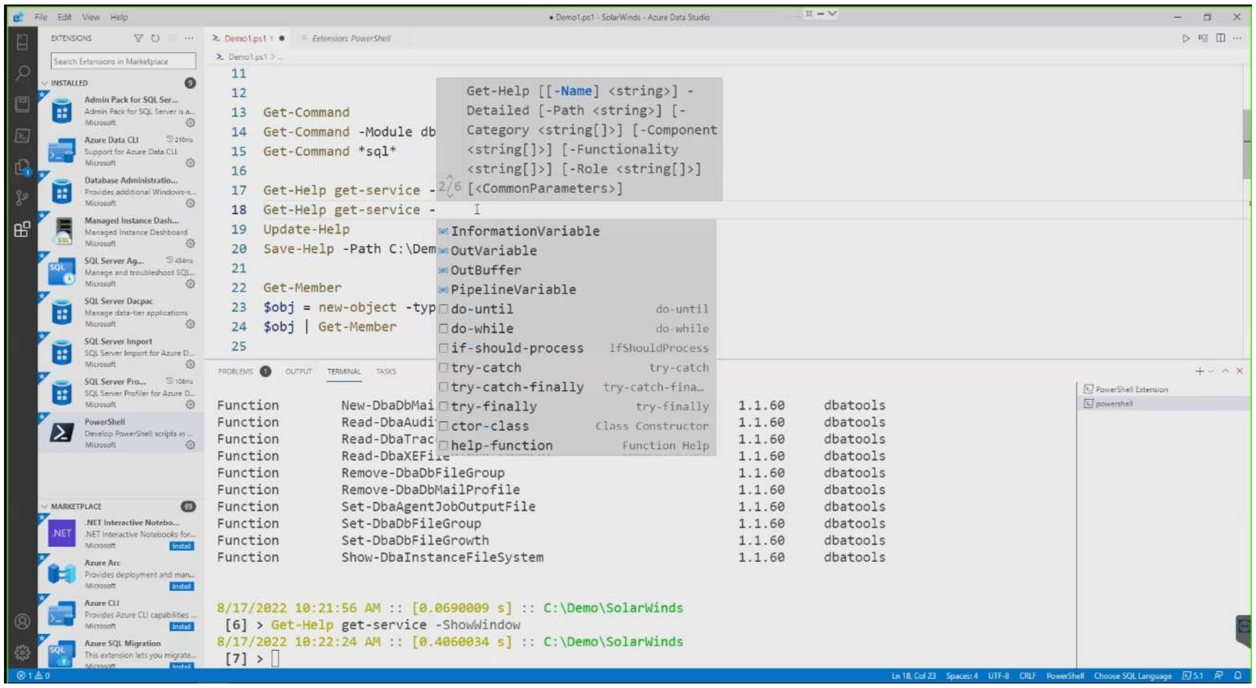
```
----                           -----
PSVersion                      5.1.17763.2931
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.17763.2931
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1


8/17/2022 10:17:34 AM :: [0.0099987 s] :: C:\Demo\SolarWinds
 [2] > Get-ExecutionPolicy
RemoteSigned
8/17/2022 10:18:40 AM :: [0.0409925 s] :: C:\Demo\SolarWinds
 [3] >
```

Get-Command -module ___________ actually loads that module.



```
    You can direct this cmdlet to get only particular services by specifying the service name or the display name of the services, or
you can pipe service objects to this cmdlet.


Parameters
    -ComputerName <System.String[]>
        Gets the services running on the specified computers. The default is the local computer.


        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       True (ByPropertyName)
        Accept wildcard characters?  false

    -DependentServices <System.Management.Automation.SwitchParameter>
        Indicates that this cmdlet gets only the services that depend upon the specified service.


        Required?                    false
        Position?                    named
        Default value                False
        Accept pipeline input?       False
        Accept wildcard characters?  false

    -DisplayName <System.String[]>
        Specifies, as a string array, the display names of services to be retrieved. Wildcards are permitted. By default, this cmdlet
gets all services on the computer.

        Required?                    true
        Position?                    named
        Default value                None
```

When using the ShowWindow feature. There are settings in the Control menu of the window also.



```
Get-Command
Get-Command -Module db
Get-Command *sql*

Get-Help get-service -
Get-Help get-service -
Update-Help
Save-Help -Path C:\Dem

Get-Member
$obj = new-object -typ
$obj | Get-Member
```

```
Get-Help [[-Name] <string>] -
Detailed [-Path <string>] [-
Category <string[]>] [-Component
<string[]>] [-Functionality
<string[]>] [-Role <string[]>]
[<CommonParameters>]
```

```
Function        New-DbaDbMai                                    1.1.60      dbatools
Function        Read-DbaAudi                                    1.1.60      dbatools
Function        Read-DbaTrac                                    1.1.60      dbatools
Function        Read-DbaXEFile                                  1.1.60      dbatools
Function        Remove-DbaDbFileGroup                           1.1.60      dbatools
Function        Remove-DbaDbMailProfile                         1.1.60      dbatools
Function        Set-DbaAgentJobOutputFile                       1.1.60      dbatools
Function        Set-DbaDbFileGroup                              1.1.60      dbatools
Function        Set-DbaDbFileGrowth                             1.1.60      dbatools
Function        Show-DbaInstanceFileSystem                      1.1.60      dbatools


8/17/2022 10:21:56 AM :: [0.0690009 s] :: C:\Demo\SolarWinds
 [6] > Get-Help get-service -ShowWindow
8/17/2022 10:22:24 AM :: [0.4060034 s] :: C:\Demo\SolarWinds
 [7] >
```

InformationVariable
OutVariable
OutBuffer
PipelineVariable
do-until   do-until
do-while   do-while
if-should-process   IfShouldProcess
try-catch   try-catch
try-catch-finally   try-catch-fina...
try-finally   try-finally
ctor-class   Class Constructor
help-function   Function Help

You get some code-writing help in Azure Data Studio.

Square brackets imply an **array** object.

You can install modules with “-force” (like a Jedi, hehe).

You can “side load” modules as well, the same module with different versions, I think.

When you import a module with maxversion or minversion, and maybe a specific version.

When you install the SQL Server product you get **sqlps** module.

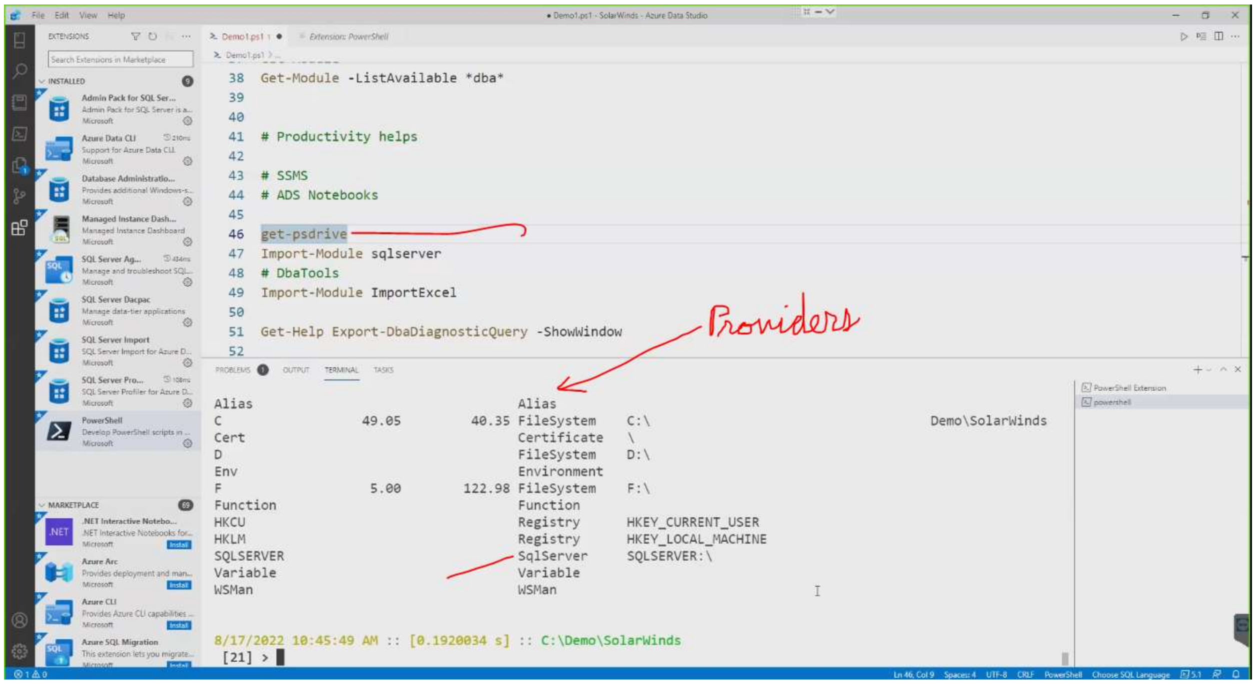
At 12:40 he talked about a circumstance where you can have a **sqlps** conflict.

Get-Module -ListAvailable tells you if the module can even be loaded, and it will show available module versions.

Linux can run PowerShell.

In SSMS you can right click on a number of objects and choose “Start PowerShell”.

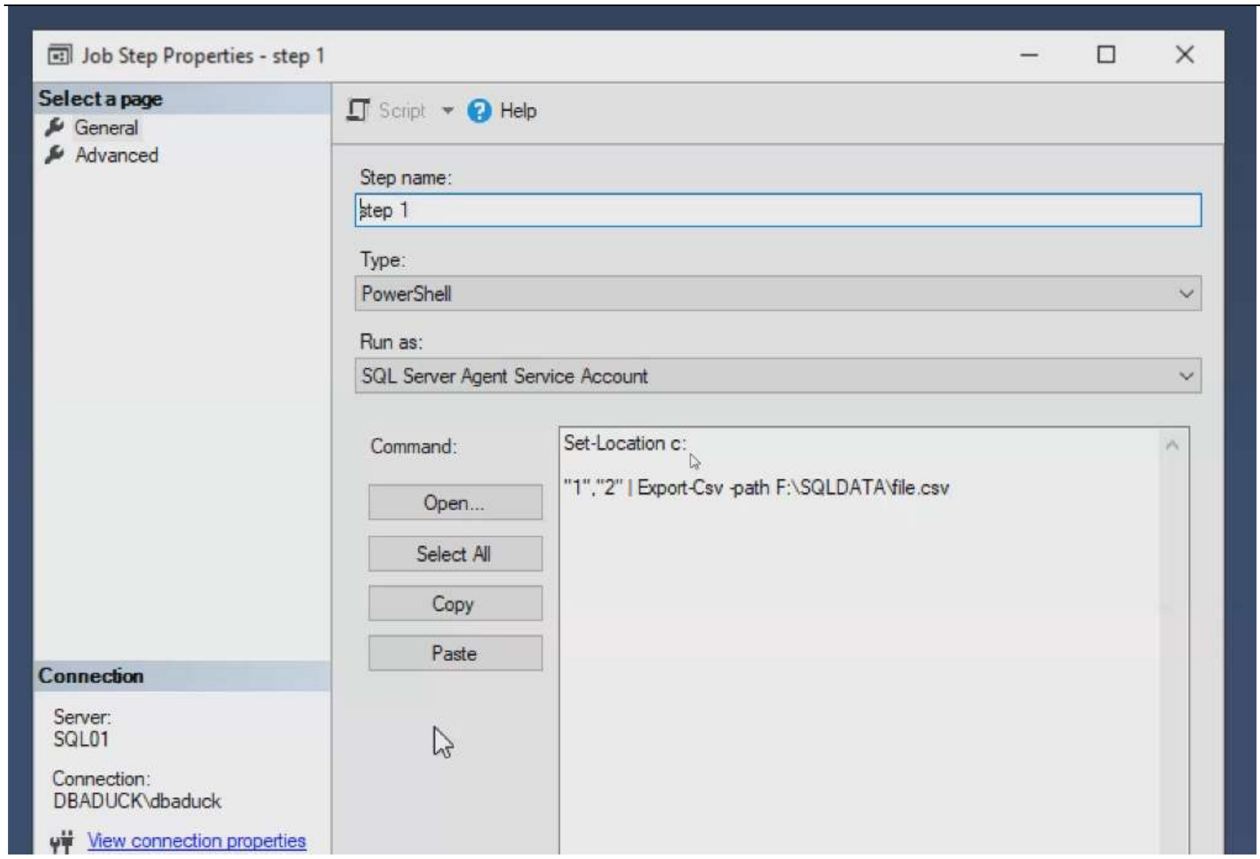
PowerShell runs on *Providers*.



```
Get-Module -ListAvailable *dba*


# Productivity helps

# SSMS
# ADS Notebooks

get-psdrive
Import-Module sqlserver
# DbaTools
Import-Module ImportExcel

Get-Help Export-DbaDiagnosticQuery -ShowWindow
```

```
Alias                                 Alias
C                 49.05         40.35 FileSystem     C:\                                  Demo\SolarWinds
Cert                                  Certificate    \
D                                     FileSystem     D:\
Env                                   Environment
F                  5.00        122.98 FileSystem     F:\
Function                              Function
HKCU                                  Registry       HKEY_CURRENT_USER
HKLM                                  Registry       HKEY_LOCAL_MACHINE
SQLSERVER                             SqlServer      SQLSERVER:\
Variable                              Variable
WSMan                                 WSMan

8/17/2022 10:45:49 AM :: [0.1920034 s] :: C:\Demo\SolarWinds
 [21] >
```

Type and execute **$host** and it will return what kind of console you have.



Step name: step 1

Type: PowerShell

Run as: SQL Server Agent Service Account

Server: SQL01

Connection: DBADUCK\dbaduck

Command:

```
Set-Location c:

"1","2" | Export-Csv -path F:\SQLDATA\file.csv
```

Regarding the SQL Agent you want to set the path (or you may get an unwanted default) at the top of your PowerShell code.