Windows PowerShell: Batch Files on Steroids

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefieldquery.com
Corporate Web sites: www.stonefieldquery.com
and www.stonefieldsoftware.com
Personal Web site : www.DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: [DougHennig](http://twitter.com/DougHennig)

Windows PowerShell has been included with the operating system since Windows 7. What is PowerShell? It's Microsoft's task automation scripting framework. PowerShell isn't just a replacement for batch files; it can do a lot more than batch files ever could. This session looks at PowerShell, including why you should start using it and how to create PowerShell scripts.

Introduction

While having a graphical user interface like Microsoft Windows is great for both ease-of-use and productivity, nothing beats commands for performing repetitive tasks. After all, a single command can replace clicking buttons, choosing menu items, and selecting choices in dialogs. Microsoft has provided a command-line interface ever since the first release of DOS. It also provided a way to put multiple commands into a text file with a "bat" extension (a "batch" file) and have one command execute all the commands in the file.

If you've ever used batch files to automate system tasks, you know they have numerous shortcomings. Error handling is rudimentary and the list of commands is short, so there's really only a small set of things you can do with batch files, like copying or moving files.

Although the command shell (cmd.exe) and batch files are still available in Windows, they've been supplanted in functionality by Windows PowerShell. PowerShell has been around for more than a decade. Starting with Windows 7, it ships with the operating system.

There are two versions of PowerShell: the classic PowerShell, the latest version of which is 5.1 and is no longer developed, and PowerShell Core, the latest version of which at this writing (April 2022) is 7.2.2. The difference is that the former is built on the .NET Framework while the latter is based on .NET Core. PowerShell Core isn't currently installed by default; to install it, open a PowerShell console window (see the "Starting with PowerShell" section) and type:

```
winget install --id=Microsoft.PowerShell -e
```

PowerShell is Windows-only while PowerShell Core is cross-platform (there are versions for Windows, MacOS, and Linux). I'm only going to marginally discuss the differences between them and will just refer to them collectively as "PowerShell." See https://techgenix.com/powershell-core/ for some of the differences between the two versions.

The key things to know about PowerShell are:

- It's built on .NET, so it has full access to .NET's extensive class library.
- It also has full access to COM and WMI, so it can do just about any system administrative task.
- PowerShell statements are interpreted rather than compiled, so they can be executed interactively in PowerShell's console window.
- PowerShell is dynamically rather than statically typed, so it works more like VFP than like .NET for data typing.
- Like .NET, everything in PowerShell is an object.

What is PowerShell good for?

In a word, everything:

- Server administration: restart or shutdown servers, change passwords, restart services, terminate processes, or just about any other task you can think of.
- Web server administration: manage web sites, application pools, and virtual directories.
- Workstation administration: perform backups, create user accounts, set or determine configuration settings, kill processes, and so on.
- Application administration: many tasks you would perform using a GUI to manage tools like SQL Server and Exchange Server are available as PowerShell commands.
- Anything else you want to automate. Later in this document, we'll look at automating deploying VFP applications.

Here's a simple example: suppose you want to document the values of system environment variables. Here's how you do it using the Windows GUI:

- Click the Windows Start button, type "environment," and choose Edit the System Environment Variables to display the System Properties dialog.
- Click the *Environment Variables* button.
- For each one, click the Edit button, copy the name and the values, and paste them into the documentation.

Here's how to do it using PowerShell:

- Launch PowerShell; the next section discusses several ways to do that.
- Type the following:

  ```
  dir env: | select key, value | export-csv result.csv
  ```

Not only does this one statement save time, it can also be automated, which the GUI steps cannot.

Starting with PowerShell

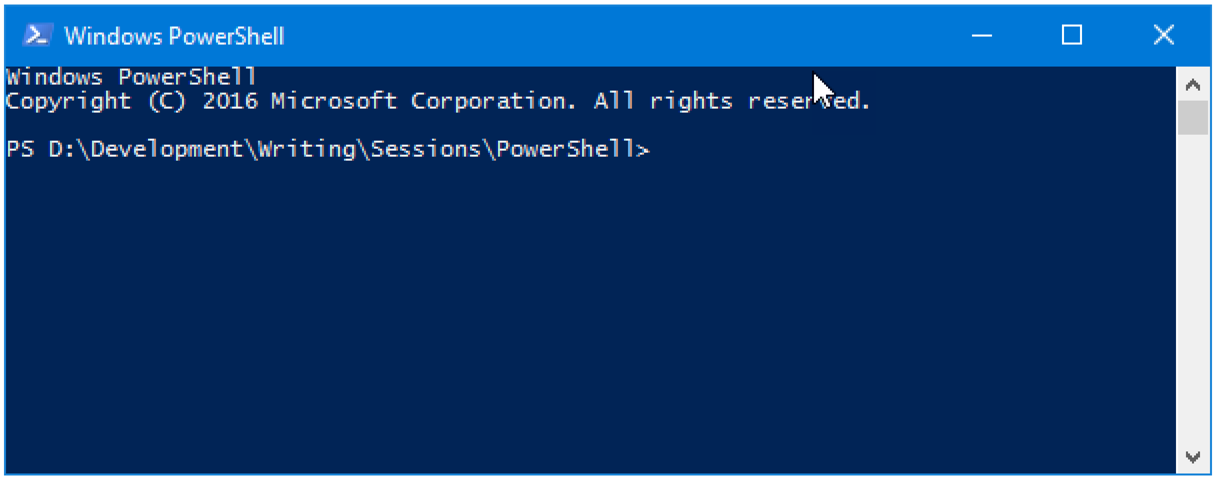
The easiest way to start playing with PowerShell is to open a console window. There are several ways you can do that:

- Open File Explorer, choose the File tab, and select either *Open Windows PowerShell* or *Open Windows PowerShell as administrator* to open a window in the current folder.
- Type "PowerShell" ("pwsh" for PowerShell Core) in the address bar of File Explorer to open a window in the current folder.

- Press Windows-S to open Search, type "PowerShell," and select *Windows PowerShell* from the list of matches (*PowerShell 7* for PowerShell Core). Right-click it and choose *Run as administrator* if necessary.

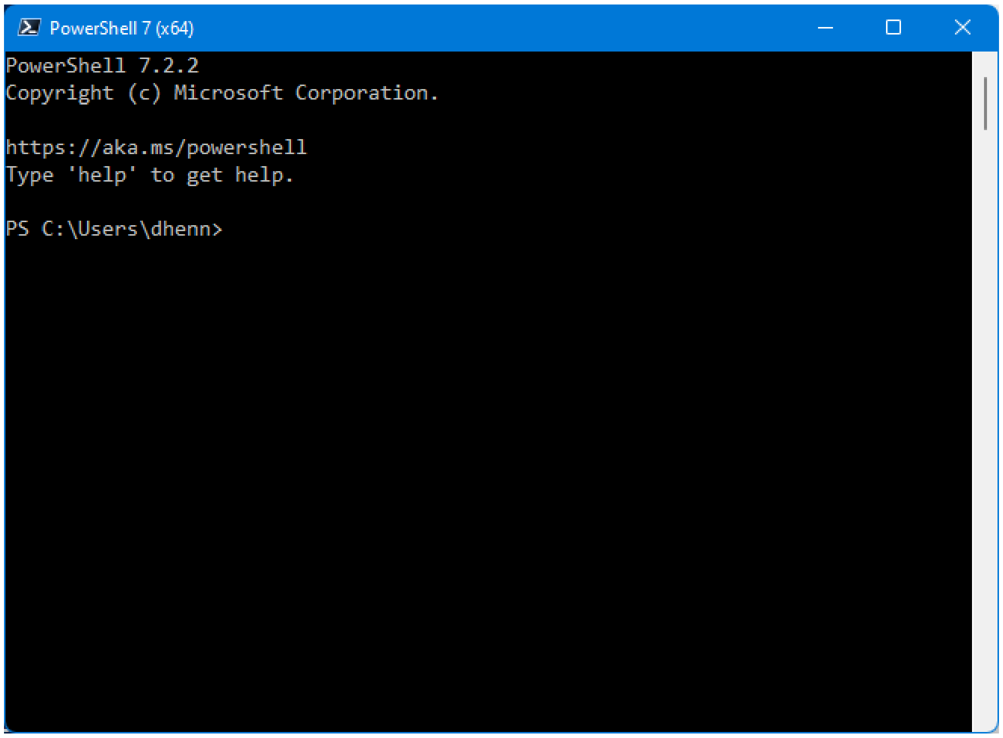
PowerShell is in C:\Windows\System32\WindowsPowerShell\v1.0, so you can also run PowerShell.exe in that folder. PowerShell Core is in C:\Program Files\PowerShell\7, so you can run Pwsh.exe in that folder.

The PowerShell console window is shown in **Figure 1** and the PowerShell Core console window in shown in **Figure 2**.



```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS D:\Development\Writing\Sessions\PowerShell>
```

Figure 1. The Windows PowerShell console looks very much like the Cmd.exe console.



```
PowerShell 7.2.2
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

PS C:\Users\dhenn>
```

Figure 2. The PowerShell Core console looks a little different than the PowerShell version.

The first thing to know about the console window is that any Cmd.exe command works. So, you can type cls (to clear the display), dir, cd, or the name of an executable or BAT file.

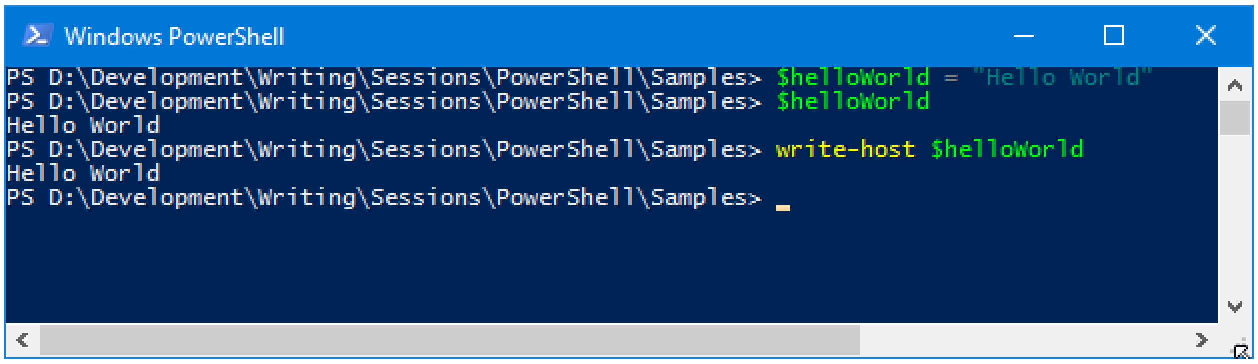
The second thing to know about the console window is that you can customize it a bit by clicking the icon in the title bar and choosing Properties from the shortcut menu. You can change the font, the cursor size, the window size, and text and background colors.

Let's explore some of PowerShell using the console window.

Variables

PowerShell variable names start with "$," are not case-sensitive, and can contain any character except "!," "@," "#," "%," "&," comma, period, and space. You should avoid using reserved words (use `Get-Help about_reserved_words` for a list of them; you may have to use `Update-Help` to download the latest help files first) and built-in variable names (`Get-Help about_automatic_variables` shows a list). Also, variable names starting with "Data" seem to cause problems. The normal naming convention for variables is camel case, such as "myVariableName."

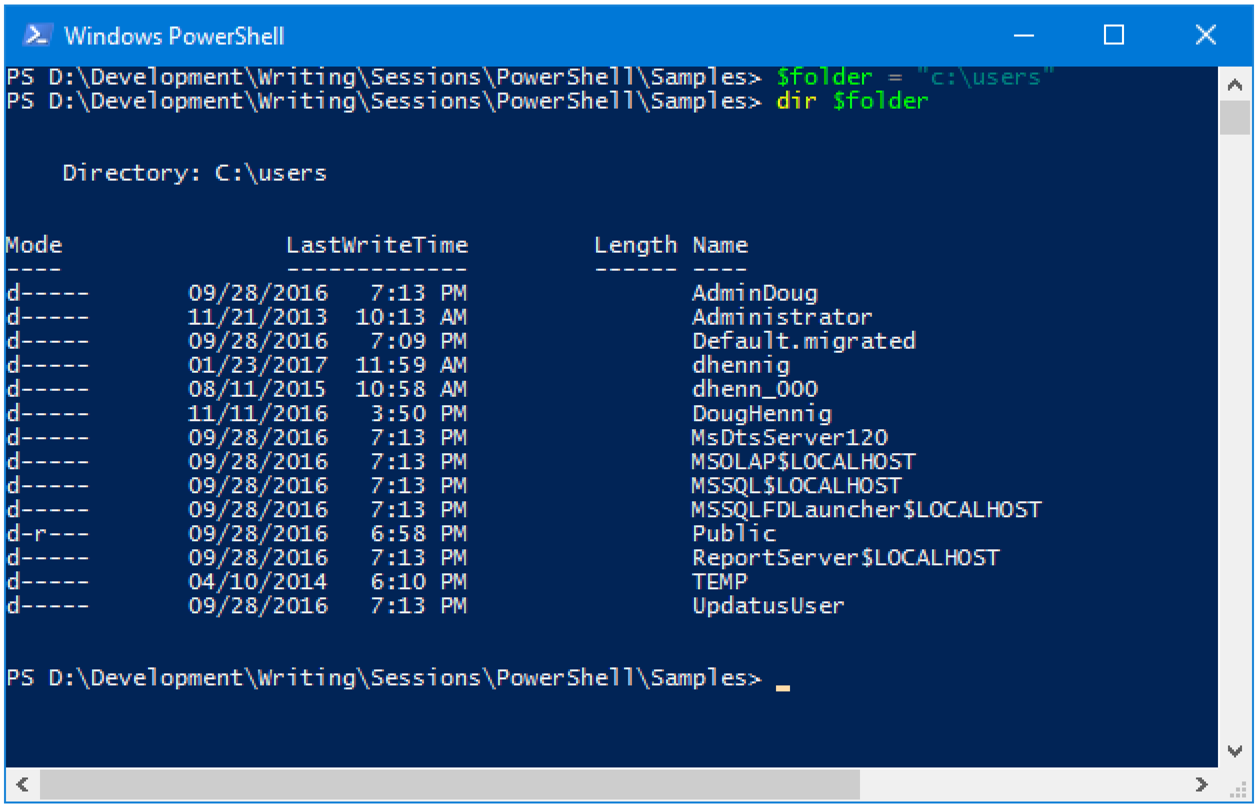
Variables are assigned a value with "=." You can output the value of a variable using just its name or with the `Write-Host` command, as shown in **Figure 3**.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> $helloWorld = "Hello World"
PS D:\Development\Writing\Sessions\PowerShell\Samples> $helloWorld
Hello World
PS D:\Development\Writing\Sessions\PowerShell\Samples> write-host $helloWorld
Hello World
PS D:\Development\Writing\Sessions\PowerShell\Samples> _
```

Figure 3. Assign a value to a variable using "=" and display it by typing its name or using Write-Host.

Variables can be passed to commands. In **Figure 4**, the variable $folder is passed to the dir command.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> $folder = "c:\users"
PS D:\Development\Writing\Sessions\PowerShell\Samples> dir $folder


    Directory: C:\users


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----       09/28/2016   7:13 PM                AdminDoug
d-----       11/21/2013  10:13 AM                Administrator
d-----       09/28/2016   7:09 PM                Default.migrated
d-----       01/23/2017  11:59 AM                dhennig
d-----       08/11/2015  10:58 AM                dhenn_000
d-----       11/11/2016   3:50 PM                DougHennig
d-----       09/28/2016   7:13 PM                MsDtsServer120
d-----       09/28/2016   7:13 PM                MSOLAP$LOCALHOST
d-----       09/28/2016   7:13 PM                MSSQL$LOCALHOST
d-----       09/28/2016   7:13 PM                MSSQLFDLauncher$LOCALHOST
d-r---       09/28/2016   6:58 PM                Public
d-----       09/28/2016   7:13 PM                ReportServer$LOCALHOST
d-----       04/10/2014   6:10 PM                TEMP
d-----       09/28/2016   7:13 PM                UpdatusUser


PS D:\Development\Writing\Sessions\PowerShell\Samples> _
```

Figure 4. Variables can be passed to commands.

PowerShell supports the same data types as .NET; the more commonly used ones are shown in **Table 1**.

Table 1. Commonly used data types in PowerShell.

Type	Description
[array]	Array of values
[bool]	Boolean True/False value
[char]	Unicode 16-bit character
[datetime]	Date and time
[decimal]	128-bit decimal value
[double]	Double-precision 64-bit floating point number
[hashtable]	Hashtable object
[int]	32-bit signed integer
[long]	64-bit signed integer
[single]	Single-precision 32-bit floating point number
[string]	Fixed-length string of Unicode characters
[xml]	Xml object

PowerShell automatically determines the data type when you assign a value:

```
$v1 = "1"
$v2 = 1
$v3 = 1.1
```

In this case, $v1 is a string, $v2 an int, and $v3 a decimal. PowerShell does automatic type conversion when necessary:

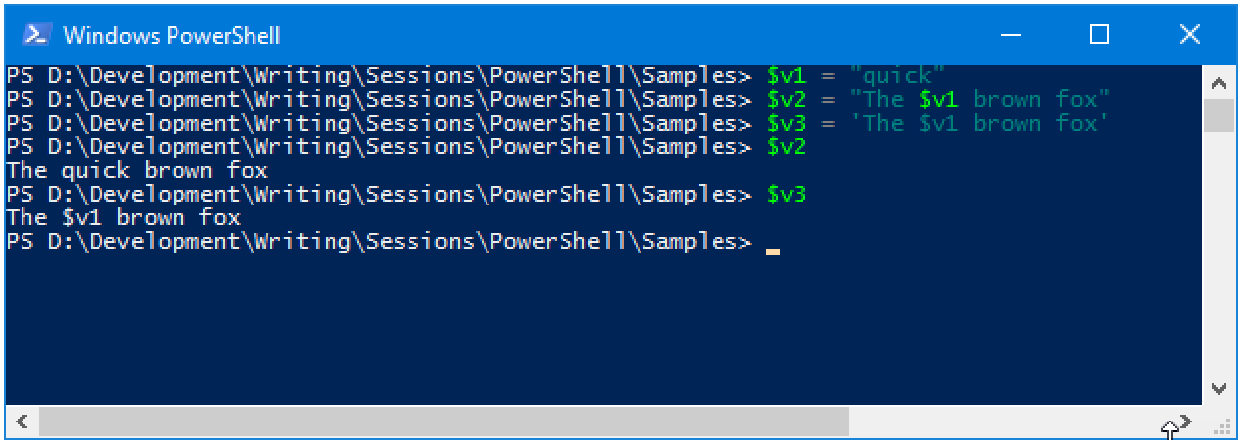
```
"1" + 1
1 + "1"
```

The first statement displays “11” while the second displays “2.”

You can also specify the data type in square brackets:

```
[int]$value = 10
```

String literals can either be surrounded with single or double quotes. The difference between the two is that double quotes cause “interpolation,” or expansion of variables, while single quotes don’t. See **Figure 5** for an example.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> $v1 = "quick"
PS D:\Development\Writing\Sessions\PowerShell\Samples> $v2 = "The $v1 brown fox"
PS D:\Development\Writing\Sessions\PowerShell\Samples> $v3 = 'The $v1 brown fox'
PS D:\Development\Writing\Sessions\PowerShell\Samples> $v2
The quick brown fox
PS D:\Development\Writing\Sessions\PowerShell\Samples> $v3
The $v1 brown fox
PS D:\Development\Writing\Sessions\PowerShell\Samples> _
```

Figure 5. Double quotes cause interpolation; single quotes don’t

Use “+” to concatenate strings. In this code, “Windows PowerShell” is displayed:

```
$v1 = "Windows"
$v2 = "PowerShell"
$v1 + ' ' + $v2
```

Like VFP macros, a period ends a variable in evaluation:

```
$here = get-item .
"Hello $here.Name"
```

This displays “Hello *path*.Name” rather than the expected “Hello *pathname*,” where *path* is the current folder. That happens because the period terminates the variable name rather than being used as the dot between the object and member names. In VFP, we’d handle this by using two periods, one to terminate the variable name and one for the dot notation. In PowerShell, use:

```
"Hello $($here.Name)"
```

Arrays are zero-based indexed lists of variables. You can specify an array using a comma-delimited list of values using one of the following ways:

```
$a = 1, 2, 3
$b = @(4, 5, 6)
```

Use "@()" for an empty array:

```
$c = @()
$c.Count
```

"+" adds an element to an array or can even combine arrays:

```
$a + 4
$a + $b
```

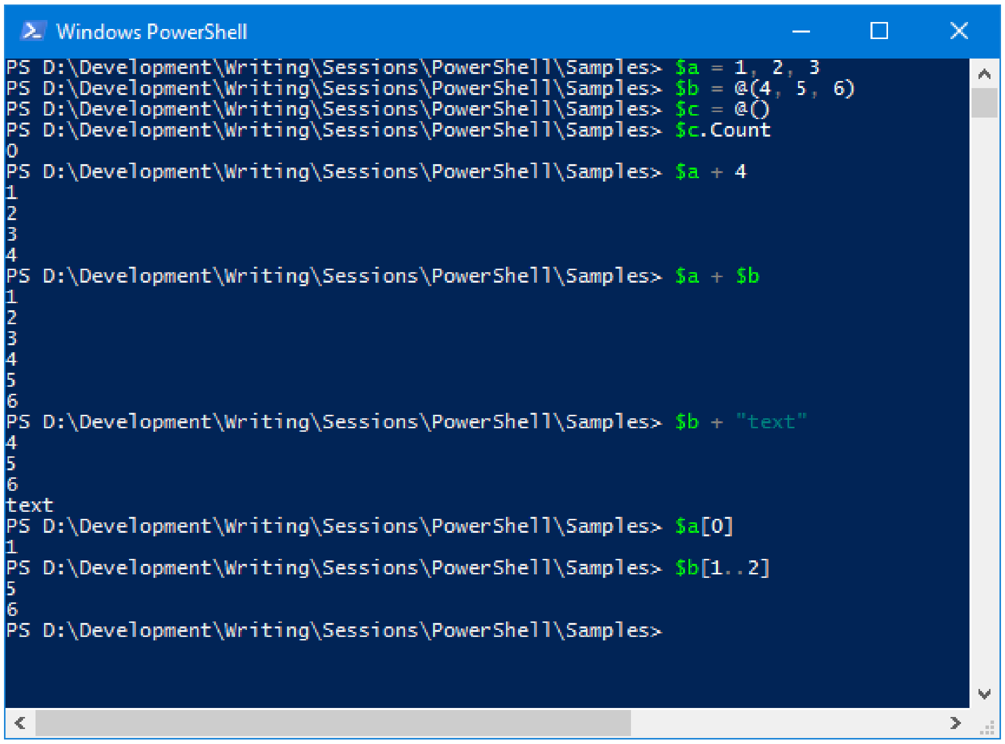
Arrays can be mixed data types:

```
$b + "text"
```

Use "array[index]" to access individual elements or "startIndex..EndIndex" for a range of values:

```
$a[0]
$b[1..2]
```

See **Figure 6** for the results of these operations.



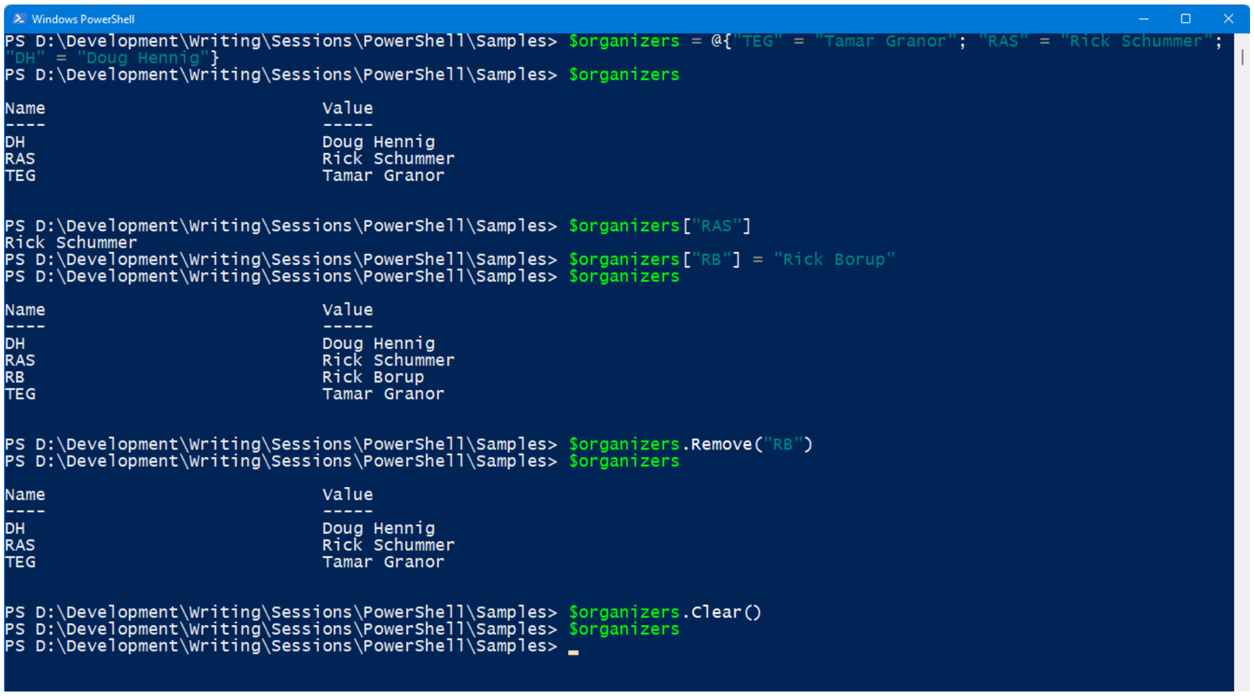
```
PS D:\Development\Writing\Sessions\PowerShell\Samples> $a = 1, 2, 3
PS D:\Development\Writing\Sessions\PowerShell\Samples> $b = @(4, 5, 6)
PS D:\Development\Writing\Sessions\PowerShell\Samples> $c = @()
PS D:\Development\Writing\Sessions\PowerShell\Samples> $c.Count
0
PS D:\Development\Writing\Sessions\PowerShell\Samples> $a + 4
1
2
3
4
PS D:\Development\Writing\Sessions\PowerShell\Samples> $a + $b
1
2
3
4
5
6
PS D:\Development\Writing\Sessions\PowerShell\Samples> $b + "text"
4
5
6
text
PS D:\Development\Writing\Sessions\PowerShell\Samples> $a[0]
1
PS D:\Development\Writing\Sessions\PowerShell\Samples> $b[1..2]
5
6
PS D:\Development\Writing\Sessions\PowerShell\Samples>
```

Figure 6. Arrays are easy to work with in PowerShell.

A HashTable stores name-value pairs of any data type and length. To declare a HashTable, use similar syntax to arrays but with curly braces instead of parenthesis and semi-colons instead of commas.

```
$organizers = @{"TEG" = "Tamar Granor"; "RAS" = "Rick Schummer"; "DH" = "Doug
Hennig"}
$organizers
$organizers["RAS"]
$organizers["RB"] = "Rick Borup"
$organizers
$organizers.Remove("RB")
$organizers
$organizers.Clear()
$organizers
```

See **Figure 7** for the results.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers = @{"TEG" = "Tamar Granor"; "RAS" = "Rick Schummer";
"DH" = "Doug Hennig"}
PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers

Name                           Value
----                           -----
DH                             Doug Hennig
RAS                            Rick Schummer
TEG                            Tamar Granor


PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers["RAS"]
Rick Schummer
PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers["RB"] = "Rick Borup"
PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers

Name                           Value
----                           -----
DH                             Doug Hennig
RAS                            Rick Schummer
RB                             Rick Borup
TEG                            Tamar Granor


PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers.Remove("RB")
PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers

Name                           Value
----                           -----
DH                             Doug Hennig
RAS                            Rick Schummer
TEG                            Tamar Granor


PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers.Clear()
PS D:\Development\Writing\Sessions\PowerShell\Samples> $organizers
PS D:\Development\Writing\Sessions\PowerShell\Samples> _
```

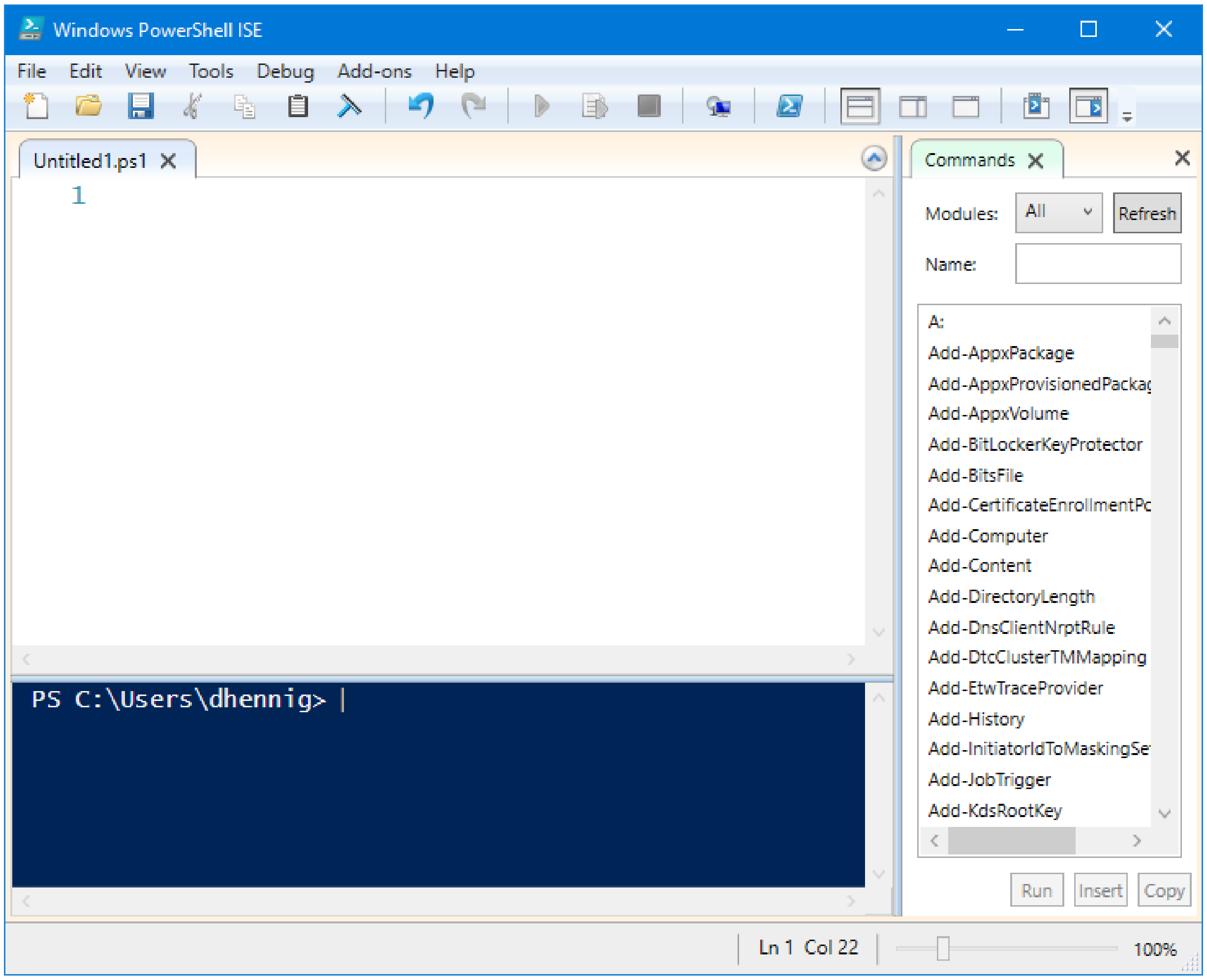
Figure 7. A HashTable is a list of name-value pairs that can be accessed by name or index.

"Splatting" allows you to execute a cmdlet with parameters from an array or HashTable rather than listing the parameters individually. To do that, pass the variable containing the array or HashTable to the cmdlet using an "@" prefix (sort of like passing a value by reference in VFP). Here's an example of using splatting to zip some files:

```
$compress = @{
  Path = "C:\SomeFolder\*.*"
  CompressionLevel = "Fastest"
  DestinationPath = "C:\Backup\Backup.zip"
}
Compress-Archive -Force @compress
```

PowerShell ISE

While the PowerShell console window is fine for playing around, eventually you'll want a better environment to work in. That environment is PowerShell ISE (interactive scripting environment). ISE has a built-in console window, a script editor, an integrated debugger, and IntelliSense for commands, so like the VFP or Visual Studio interactive development environments (IDEs), it's a better environment to work in. See **Figure 8** to see what ISE looks like.



```
PS C:\Users\dhennig>
```

```
A:
Add-AppxPackage
Add-AppxProvisionedPackag
Add-AppxVolume
Add-BitLockerKeyProtector
Add-BitsFile
Add-CertificateEnrollmentPc
Add-Computer
Add-Content
Add-DirectoryLength
Add-DnsClientNrptRule
Add-DtcClusterTMMapping
Add-EtwTraceProvider
Add-History
Add-InitiatorIdToMaskingSe
Add-JobTrigger
Add-KdsRootKey
```

Figure 8. PowerShell ISE is an interactive development environment for PowerShell.

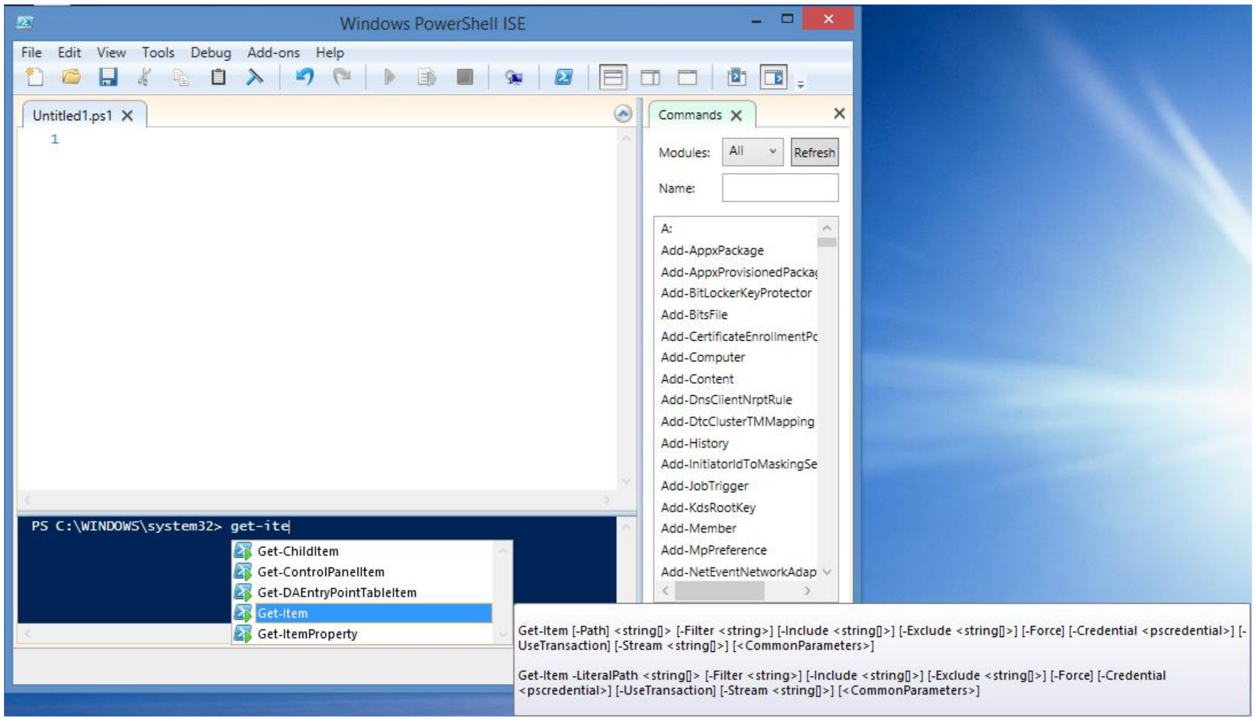
(Note: PowerShell Core doesn't have its own IDE. Instead, use Visual Studio Code with the PowerShell Extension.)

To run PowerShell ISE, search for "powershell ise" or run PowerShell_ise.exe in C:\Windows\System32\WindowsPowerShell\v1.0. You can also right-click a ps1 file (a PowerShell script file we'll discuss later) and choose Edit, or from a Windows Explorer window, type "powershell_ise" in the address bar.

In addition to the usual menu, toolbar, and status bar of most IDEs, ISE has three main areas:

- The script pane is an editor where you work on a script file. It's like the program editor in VFP: it has IntelliSense and syntax coloring to assist with editing commands.
- The Commands pane shows a list of commands. You can search the list by name, filter by module, and see the syntax and parameters for the selected command.
- The console pane is like the console window but adds IntelliSense so it's a lot easier to work in.

In both the script and console panes, command completion makes it easy to enter a command. Type the first part of the command and, as you can see in **Figure 9**, IntelliSense lists matching commands. Hit Tab for command completion; that is, inserting the selected item into the text. This even works for non-command things like "CD .\;" a list of directories pops up and you can press Tab to enter the selected one.



```
PS C:\WINDOWS\system32> get-ite
```

```
Get-Item [-Path] <string[]> [-Filter <string>] [-Include <string[]>] [-Exclude <string[]>] [-Force] [-Credential <pscredential>] [-
UseTransaction] [-Stream <string[]>] [<CommonParameters>]

Get-Item -LiteralPath <string[]> [-Filter <string>] [-Include <string[]>] [-Exclude <string[]>] [-Force] [-Credential
<pscredential>] [-UseTransaction] [-Stream <string[]>] [<CommonParameters>]
```

Figure 9. The console pane in ISE includes IntelliSense.

Commands

There are four types of commands in PowerShell:

- Native commands
- Cmdlets

- Scripts
- Functions

I discussed native commands earlier so let's look at the others.

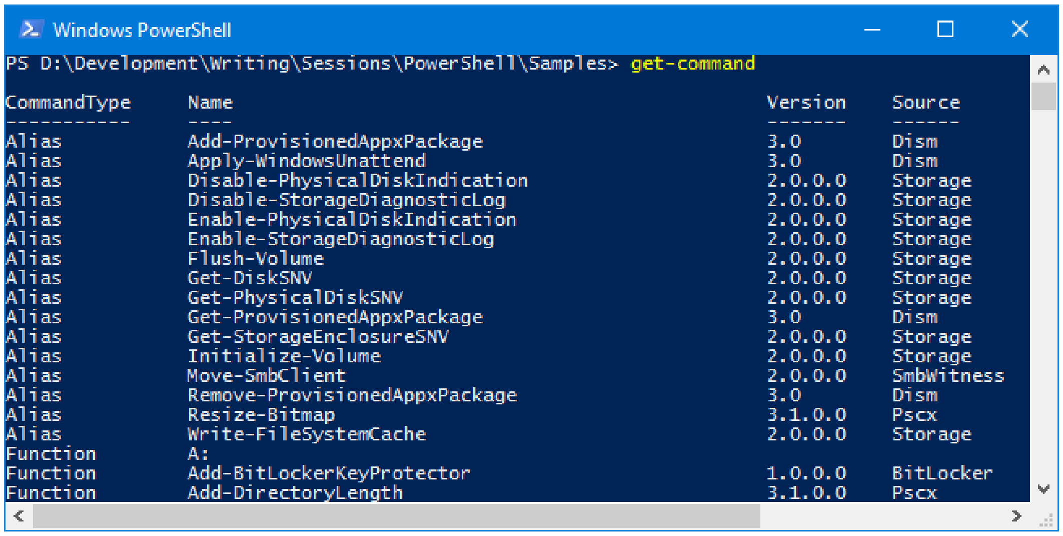
Cmdlets

Cmdlets are commands built into PowerShell. Their names are standardized to use verb-noun syntax. For example, `Get-Item` displays information about the specified folder:

```
Get-item .
$here = get-item .
$here.Name
$here.GetType()
```

Notice that PowerShell cmdlets are also not case-sensitive; Get-Item, Get-item, and get-item all do the same thing.

To see a list of cmdlets, use `Get-Command`; see **Figure 10**.

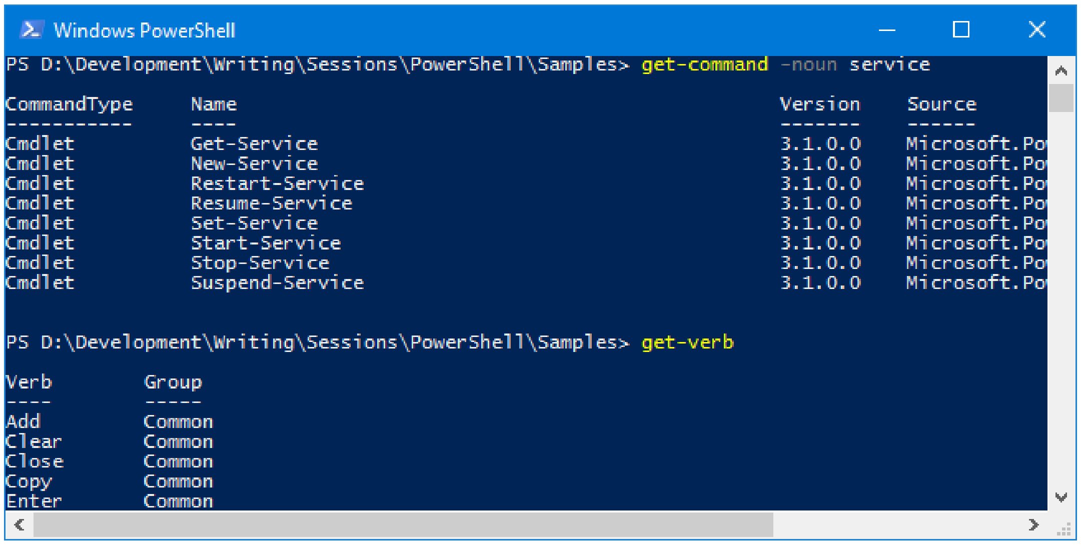


Windows PowerShell

```
PS D:\Development\Writing\Sessions\PowerShell\Samples> get-command

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           Add-ProvisionedAppxPackage                         3.0        Dism
Alias           Apply-WindowsUnattend                              3.0        Dism
Alias           Disable-PhysicalDiskIndication                     2.0.0.0    Storage
Alias           Disable-StorageDiagnosticLog                       2.0.0.0    Storage
Alias           Enable-PhysicalDiskIndication                      2.0.0.0    Storage
Alias           Enable-StorageDiagnosticLog                        2.0.0.0    Storage
Alias           Flush-Volume                                       2.0.0.0    Storage
Alias           Get-DiskSNV                                        2.0.0.0    Storage
Alias           Get-PhysicalDiskSNV                                2.0.0.0    Storage
Alias           Get-ProvisionedAppxPackage                         3.0        Dism
Alias           Get-StorageEnclosureSNV                            2.0.0.0    Storage
Alias           Initialize-Volume                                  2.0.0.0    Storage
Alias           Move-SmbClient                                     2.0.0.0    SmbWitness
Alias           Remove-ProvisionedAppxPackage                      3.0        Dism
Alias           Resize-Bitmap                                      3.1.0.0    Pscx
Alias           Write-FileSystemCache                              2.0.0.0    Storage
Function        A:
Function        Add-BitLockerKeyProtector                          1.0.0.0    BitLocker
Function        Add-DirectoryLength                                3.1.0.0    Pscx
```

Figure 10. Get-command lists all commands.

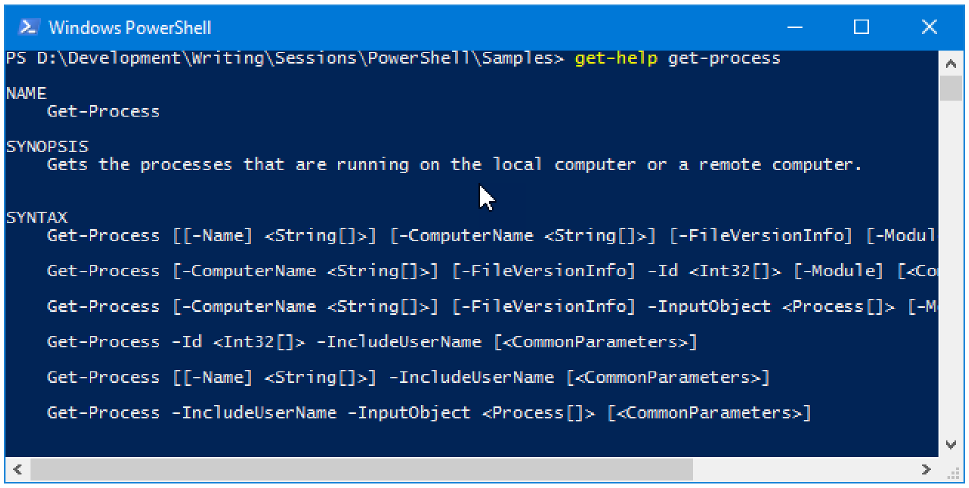
To see commands using a particular verb, use `Get-Command -verb` *`verb`*, such as `Get-Command -verb get`. Common verbs are add, clear, get, new, and set. For commands using a particular noun, use `Get-Command -noun` *`noun`*, such as `Get-Command -noun service`. For a list of verbs, use `Get-Verb.` See **Figure 11**.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> get-command -noun service

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Get-Service                                        3.1.0.0    Microsoft.Po
Cmdlet          New-Service                                        3.1.0.0    Microsoft.Po
Cmdlet          Restart-Service                                    3.1.0.0    Microsoft.Po
Cmdlet          Resume-Service                                     3.1.0.0    Microsoft.Po
Cmdlet          Set-Service                                        3.1.0.0    Microsoft.Po
Cmdlet          Start-Service                                      3.1.0.0    Microsoft.Po
Cmdlet          Stop-Service                                       3.1.0.0    Microsoft.Po
Cmdlet          Suspend-Service                                    3.1.0.0    Microsoft.Po


PS D:\Development\Writing\Sessions\PowerShell\Samples> get-verb

Verb        Group
----        -----
Add         Common
Clear       Common
Close       Common
Copy        Common
Enter       Common
```

Figure 11. Get-Command –noun lists nouns and Get-Verb lists verbs.

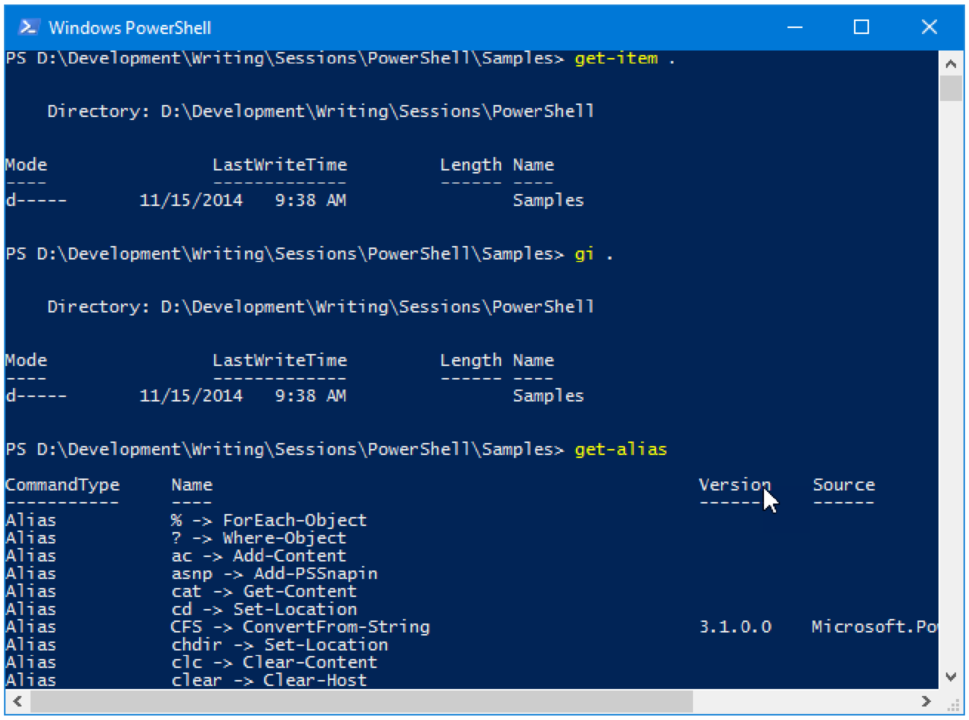
`Get-Help` is a great command because it gives you help about PowerShell. `Get-Help` *`cmdlet`* displays help about the specified cmdlet (see **Figure 12**). Interestingly, the text displayed by `Get-Help` isn't built into PowerShell but is instead in the scripts that create the cmdlet; we'll see that later in the "Documenting your scripts" section. `Get-Help` *`cmdlet`* `-examples` shows examples of the command.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> get-help get-process

NAME
    Get-Process

SYNOPSIS
    Gets the processes that are running on the local computer or a remote computer.


SYNTAX
    Get-Process [[-Name] <String[]>] [-ComputerName <String[]>] [-FileVersionInfo] [-Modul

    Get-Process [-ComputerName <String[]>] [-FileVersionInfo] -Id <Int32[]> [-Module] [<Co

    Get-Process [-ComputerName <String[]>] [-FileVersionInfo] -InputObject <Process[]> [-M

    Get-Process -Id <Int32[]> -IncludeUserName [<CommonParameters>]

    Get-Process [[-Name] <String[]>] -IncludeUserName [<CommonParameters>]

    Get-Process -IncludeUserName -InputObject <Process[]> [<CommonParameters>]
```

Figure 12. Get-help is a great way to see information about a specific item.

Many cmdlets have abbreviated names, known as aliases. For example, "gi" is the alias for `Get-Item`. Helpfully, some well-known DOS commands, like cd and dir, are just abbreviations for PowerShell commands (`Set-Location` and `Get-ChildItem`, in this case). Use `Get-Alias` or its alias `gal` to see a list of aliases. See **Figure 13**.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> get-item .


    Directory: D:\Development\Writing\Sessions\PowerShell


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----       11/15/2014   9:38 AM                Samples


PS D:\Development\Writing\Sessions\PowerShell\Samples> gi .


    Directory: D:\Development\Writing\Sessions\PowerShell


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----       11/15/2014   9:38 AM                Samples


PS D:\Development\Writing\Sessions\PowerShell\Samples> get-alias

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           % -> ForEach-Object
Alias           ? -> Where-Object
Alias           ac -> Add-Content
Alias           asnp -> Add-PSSnapin
Alias           cat -> Get-Content
Alias           cd -> Set-Location
Alias           CFS -> ConvertFrom-String                          3.1.0.0    Microsoft.Po
Alias           chdir -> Set-Location
Alias           clc -> Clear-Content
Alias           clear -> Clear-Host
```

Figure 13. Aliases, such as “gi” for “get-item,” make typing in the command window easier.

You can create your own aliases using `Set-Alias` *`alias command`*. However, user-defined aliases are only in effect for that PowerShell session and are forgotten once you’ve closed it. If you want to use a certain set of aliases regularly, use the `Export-Alias` and `Import-Aliases` commands to write to and read from text files of aliases.

Like FoxPro commands and functions, which in general can be abbreviated to the first four letters of the name, abbreviations are fine for use at the command line but I don’t recommend using them in scripts for reasons of clarity.

Table 2 lists some commonly used cmdlets.

Table 2. Commonly used cmdlets.

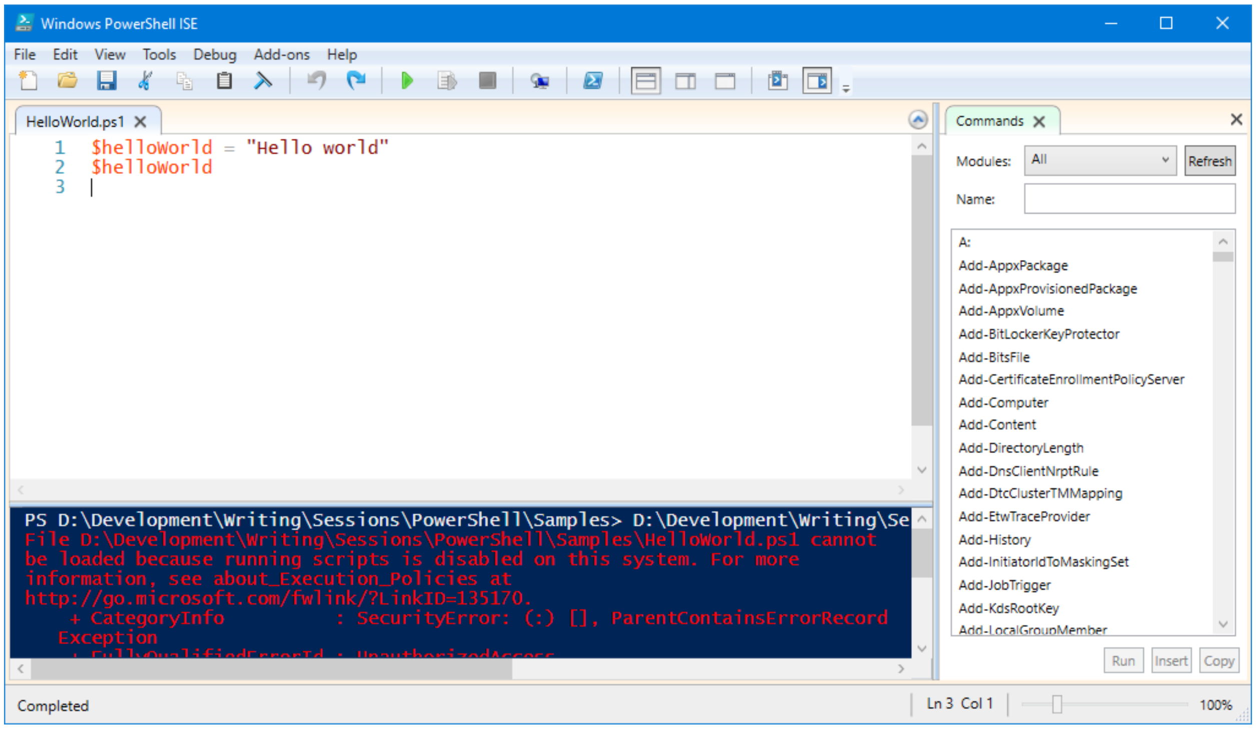
Name	Description
Add-Content	Writes to a file. Similar to the VFP STRTOFILE() function.
Get-ChildItem	Gets the items in the specified path. Similar to the VFP ADIR() function but can work with other things such as environment variables and the Windows Registry, and has a lot of flexibility including recursively reading subdirectories.
Get-Content	Reads the content of a file. Similar to the VFP FILETOSTR() function.
Get-Date	Gets the current date. Similar to the VFP DATE() function.
Get-ItemProperty	Gets the properties of the specified item. Often used to get values from the Registry.
Get-Location	Gets the current location. Similar to the VFP CURDIR() function.
Get-Member	Displays a list of the members of the specified object (remember that everything in PowerShell is an object). Similar to the VFP AMEMBERS() function.
Get-Variable	Lists variables. Similar to the VFP DISPLAY MEMORY command.
New-Object	Instantiates the specified class. Similar to the VFP NEWOBJECT() function.
Remove-Item	Removes the specified item. Similar to the VFP ERASE and RD commands but can be recursive and can handle non-empty folders.

Test-Path	Determines if the specified path exists. Similar to the VFP FILE() and DIRECTORY() functions, although it works on any path, including Registry entries.
Write-Host	Outputs to the console. Similar to the VFP ? command.

Scripts

A PowerShell script is a text file with a "ps1" extension that contains PowerShell statements. Although you can use any text editor to create a script, as we saw earlier, using PowerShell ISE is the preferred editing environment. You can type the statements in a new script or open and edit an existing script file.

To run a script, click the green arrow in the toolbar or press F5. One thing you'll quickly find is that by default, PowerShell execution policy prevents running scripts (**Figure 14**).



```
$helloWorld = "Hello world"
$helloWorld
```

```
PS D:\Development\Writing\Sessions\PowerShell\Samples> D:\Development\Writing\Se
File D:\Development\Writing\Sessions\PowerShell\Samples\HelloWorld.ps1 cannot
be loaded because running scripts is disabled on this system. For more
information, see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
    + CategoryInfo          : SecurityError: (:) [], ParentContainsErrorRecord
   Exception
```

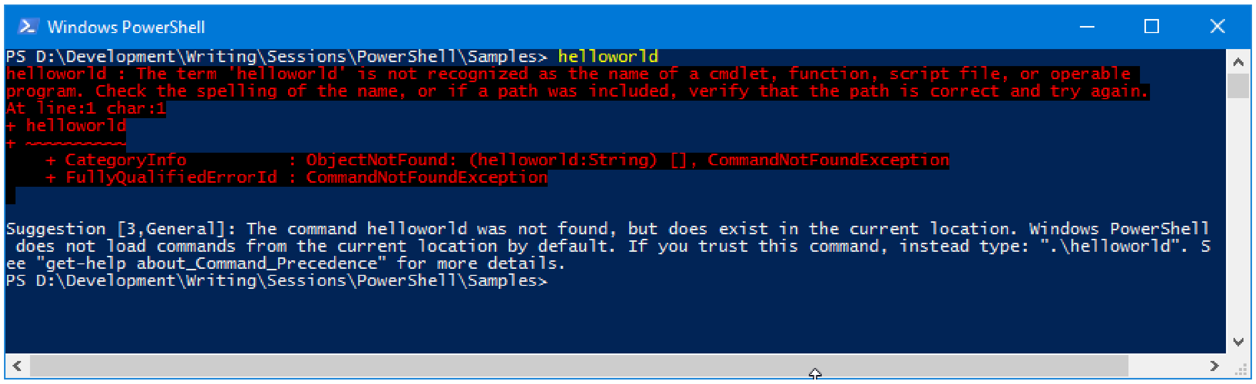
Figure 14. By default, running PowerShell scripts is disabled.

To allow script execution, use `Set-ExecutionPolicy` *`level`*, where *level* is one of the levels shown in **Table 3**. Note that you either need to run PowerShell as administrator or add `-scope currentuser` to the command.

Table 3. Use one of these settings with Set-ExecutionPolicy to control script execution.

Type	Description
Restricted	Scripts cannot be executed.
AllSigned	All scripts must be signed by a trusted publisher.
RemoteSigned	Scripts downloaded from the Internet must be signed by a trusted publisher.
Unrestricted	Scripts can be executed. You are prompted for permissions before running scripts downloaded from the Internet.

You can’t run a script by simply typing its name; see **Figure 15** for an example.



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> helloworld
helloworld : The term 'helloworld' is not recognized as the name of a cmdlet, function, script file, or operable
program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ helloworld
+ ~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (helloworld:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException


Suggestion [3,General]: The command helloworld was not found, but does exist in the current location. Windows PowerShell
 does not load commands from the current location by default. If you trust this command, instead type: ".\helloworld". S
ee "get-help about_Command_Precedence" for more details.
PS D:\Development\Writing\Sessions\PowerShell\Samples>
```

Figure 15. Running a script in the console window gives an error unless you prefix it with “.\.”

As the hint suggests, PowerShell doesn’t run scripts from the current directory by default. You have a couple of choices:

- Add the current directory to the PowerShell path by issuing `$env:Path += ";."`.
- Prefix the script name with `.\` to run it as an “encapsulated” script; that is, like a separate application. This is sometimes called “dot source running.” You can also run a script using `. .\`*`script`* (that is, a period, a space, a period, a backslash, then the script name), which runs the script as part of the command session, which mean variables can overwrite each other. This probably isn’t practical in a production environment but is handy for testing and debugging.

Speaking of debugging, debugging PowerShell scripts is very much like debugging in other IDEs. Set or clear a breakpoint on a statement by choosing Toggle Breakpoint from the Debug menu or pressing F9. Start running a script or continue to the next breakpoint using Run/Continue in the Debug menu or F5. Step through code using the Step Over, Step Into, and Step Out items in the Debug menu (F10, F11, and Shift-F11, respectively). Stop debugging using Stop Debugger in the Debug menu or Shift-F5. The console pane acts like the VFP Command window in that you can change the values of variables, execute other commands while execution is paused, and so on.

Some other things to know about scripts:

- “#” is the comment begin character, similar to “*” in VFP:

  ```
  # Get the current folder.
  $currFolder = Get-Location
  ```

- “&” runs an executable program, similar to “!” in VFP. For example, this code runs VFP with a parameter:

  ```
  $VFPexe = "C:\Program Files (x86)\Microsoft Visual FoxPro 9\vfp9.exe"
  $buildConfig = "-c$currFolder\Build.fpw"
  & $VFPexe $buildConfig
  ```

- To force a script to be executed by PowerShell Core, add this at the start:

  ```
  #requires -PSEdition Core
  ```

  Or use this to force it to PowerShell:

  ```
  #requires -PSEdition Desktop
  ```

PowerShell profile

A profile is a script that runs when PowerShell starts, similar to the Startup Program setting in VFP. The path of the profile is contained in the built-in variable $profile; by default, it contains "C:\Users\username\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1," but that script file isn't created automatically.

The following set of commands checks whether the file specified in $profile exists, creates the file, and opens it in Notepad:

```
Test-Path $Profile
New-Item -Path $Profile -ItemType file –Force
notepad $Profile
```

Some of the things you can do in the profile are load aliases, load frequently-used modules (I'll discuss modules later), set the default path, and so on.

Functions

Functions in PowerShell are just like user-defined functions in VFP. The syntax for a function is:

```
function Name(parameters)
{
   code
}
```

Here's an example:

```
function Get-Greeting()
{
   "Hello, Doug"
}
```

Here are some function concepts:

- The function name doesn't have to follow verb-noun syntax. Here's the same function with a different name:

  ```
  function Hello-World()
  {
     "Hello, Doug"
  }
  ```

- As you can see in these examples, parameters are not required in the function definition.
- Parameters can be specified as shown in the syntax above, using a `param` statement, which must be the first thing in the code block, or not at all but retrieved using the $args variable.

```
function Hello-World()
{
   param ($name)
   "Hello, $name"
}

function Test-Args()
{
   $passed = $args[0] + " " + $args[1]
   "The parameters are $passed"
}
```

- Parameters are referenced like variables—that is, they are prefixed with “$”—and can either be data-typed or not. Multiple parameters are separated with commas.

```
function Hello-World([string]$name)
{
   "Hello, $name"
}
```

- Even if they are included in the definition, parameters don’t have to be passed unless they’re mandatory (see the next point); the variables specified as parameter names are null or false in that case. Use something like the following to determine if a parameter was passed a value:

```
if (!$name)
```

- You can specify that a parameter must be passed a value using the `mandatory` keyword. You’re prompted for missing values when the script is run in ISE but get an error otherwise.

```
function Hello-World([parameter(mandatory=$true)] $name)
{
   "Hello $name"
}
```

- You can specify a default value for parameters that aren’t passed:

```
function Hello-World()
{
   param ($name = "Unknown")
   "Hello, $name"
}

function Hello-World($name = "Unknown")
{
   "Hello, $name"
```

```
    }
```

- Use the `return` statement to return a value from the function.

Similar to a VFP PRG file, if a script file contains only a function, executing the script file won't do anything. You have to "load" the script file for PowerShell to recognize its functions. To do that, type period, space, and the path to the script file (which you need to prefix with ".\." For example, to make the Get-Greeting function available, do the following (assuming it's in a script file named GetGreeting.ps1):

```
. .\GetGreeting.ps1
```

Now you can type "Get-"and see "Get-Greeting" in the IntelliSense list.

Note that if you name the script file and the function the same, PowerShell gets a little confused and sometimes lists both. For that reason, I suggest using a different script file name.

To avoid having to do this every time you start ISE, you can load your functions at startup by adding the appropriate commands to your profile.

Documenting your scripts

As I mentioned earlier, `Get-Help` doesn't display hard-coded text but instead reads the help information from the script file. This is called "comment-based help." To add documentation to your own script files, use syntax like the following:

```
function MyFunction($someParameter)
{
   <#
   .Synopsis
   This describes what the function does.

   .Description
   This is a longer description of the function.

   .Example
   Show the syntax here

   Describe what happens here. You can have multiple .Example sections.

   .Parameter someParameter
   Discuss the parameter.

   .Notes
   Anything you want, such as your name.

   .Link
   The URL for documentation, your web site, etc.
   #>

   # Code for the function
}
```

Other keywords are available; see http://tinyurl.com/orlvxte for more information on comment-based help.

To display help for the function, use `Get-Help` *`function`* or `Get-Help` *`function`* `-full`, where *function* is the function name.

Modules

A module is a PowerShell script with a "psm1" extension. Modules usually consist of functions so they don't execute when loaded but simply make those functions available to other scripts. The environment variable psModulePath, which you can see in PowerShell using `$env:psModulePath`, contains the paths for modules. To load a module, use `Import-Module` *`moduleName`*. The `Get-Module` command lists loaded modules.

You can create your own module by creating a script with a "psm1" extension rather than "ps1." Putting your psm1 file into one of the folders in the modules path, usually C:\Users\UserName\Documents\WindowsPowerShell\Modules, makes it available to be imported into any script.

Lots of modules are available for PowerShell. Some popular ones are:

- PSCX (PowerShell Community Extensions; https://github.com/Pscx/Pscx) provides features like SQL queries against databases, reading from and writing to the clipboard, and so on.
- PowerShellGet (https://tinyurl.com/53z3fvzk) makes it easy to find and install other modules in the PowerShell Gallery, a central repository for PowerShell content.
- Posh-Git (available using PowerShellGet) provide PowerShell integration with the Git version control system.
- The popular application installer Chocolatey (https://chocolatey.org) is actually a PowerShell module that supports downloading and installing thousands of applications using a single command. We'll see this in the "Automating application installation" section of this document.

Operators

PowerShell uses the comparison operators shown in **PowerShell has** three logical operators: "-not" (or its abbreviation "!"), "-and," and "-or."

PowerShell has several assignment operators. "++" is the increment operator; "$i++" is the equivalent of "$i = $i + 1." Similarly, "--" is the decrement operator. "+=" adds the specified value; "$x += $y" is the equivalent of "$x = $x + $y."

Table 4. Note that comparisons are case-insensitive; use a prefix of "c" (such as "-ceq") for a case-sensitive comparison. If the item being compared is a scalar value, the result of the comparison is true or false. If the item is a collection or array, the result is any matching values.

PowerShell has three logical operators: "-not" (or its abbreviation "!"), "-and," and "-or."

PowerShell has several assignment operators. "++" is the increment operator; "$i++" is the equivalent of "$i = $i + 1." Similarly, "--" is the decrement operator. "+=" adds the specified value; "$x += $y" is the equivalent of "$x = $x + $y."

Table 4. PowerShell comparison operators.

Operator	Description
-eq	Equals
-ne	Not equals
-gt	Greater than
-lt	Less than
-ge	Greater than or equals
-le	Less than or equal
-like	Like; you can use "*" as a wildcard character.
-notlike	Not like
-contains	Contains
-notcontains	Does not contain
-in	In a list
-notin	Not in a list

Branching code

Like VFP, PowerShell has an `if` statement that allows conditional branches. Here's the syntax:

```
if (some condition)
{
   statement(s)
}
```

The condition must be in parentheses and be an expression that returns a Boolean value, and the statements must be surrounded by curly braces. Typically, you'll compare one value to another using one of the comparison operators listed in PowerShell has three logical operators: "-not" (or its abbreviation "!"), "-and," and "-or."

PowerShell has several assignment operators. "++" is the increment operator; "$i++" is the equivalent of "$i = $i + 1." Similarly, "--" is the decrement operator. "+=" adds the specified value; "$x += $y" is the equivalent of "$x = $x + $y."

Table 4. You can combine expressions using logical operators discussed earlier.

The `else` and `elseif` statements provide alternate execution paths:

```
if (some condition)
{
   statement(s)
}
elseif
{
   statement(s)
}
elseif
{
   statement(s)
}
else
{
   statement(s)
}
```

The PowerShell `switch` statement is similar to the VFP DO CASE structure, but not as flexible: rather than each CASE statement being a complete Boolean expression, including function and method calls, `switch` only allows you to test the various possible values of a single expression (although it doesn't have to be, and is rarely, Boolean). Here's the syntax:

```
switch (expression)
{
   value1
   {
      statement(s)
   }
   value2
   {
      statement(s)
   }
   default
   {
      statement(s)
   }
}
```

Here are some comments about this:

- Like `if`, the expression must be in parentheses and the statements must be surrounded with curly braces.
- `default` is the PowerShell equivalent of VFP's `otherwise`. It's not required but is a good idea.

Branching.ps1, one of the files accompanying this document, shows examples of `if` and `switch` statements; see **Listing 1**.

Listing 1. Branching.ps1 shows examples of branching commands.

```
$a = 3
$b = ""
```

```
# IF
if ($a -eq 1)
{
   $b = "Red"
}
elseif ($a -eq 2)
{
   $b = "Blue"
}
if ($a -eq 3)
{
   $b = "Green"
}
Write-Host $b

# SWITCH
$a = 1
switch ($a)
{
   1 {$b = "Red"}
   2 {$b = "Blue"}
   3 {$b = "Green"}
   default {$b = "Unknown"}
}
Write-Host $b
```

Loops

PowerShell has the traditional set of looping constructs: do/while/until, for, and foreach. In addition, it has two commands that control loop execution: `break` and `continue`.

do while, while, and do until

`do while` and `while` execute a block of code as long as a condition is true. The difference is that `do while` tests the condition at the end so it executes at least once; `while` tests the condition at the start so it may not execute at all. This code outputs the numbers 1 through 5:

```
$i = 1
do
{
   Write-Host $i
   $i++
}
while ($i -le 5)
```

`do until` is the opposite of `do while`: it executes until the condition becomes true.

```
$i = 1
do
{
   Write-Host $i
   $i++
}
until ($i -gt 5)
```

for

The `for` command use the same syntax as C, C#, JavaScript, and similar languages:

```
for (initialization; termination condition; repeat expression)
```

Initialization typically initializes a loop counter to a starting value, the termination condition specifies when the loop is done, and the repeat expression typically increments the loop counter. For example:

```
for ($i = 1; $i -lt 6; $i++)
{
   Write-Host $i
}
```

foreach

`foreach` executes a block of code once for each item in an array or collection. It uses similar syntax to VFP and other languages. For example, this code lists all the files in the current folder and its subdirectories:

```
$currFolder = Get-Location
foreach($item in (Get-ChildItem -Path $currFolder -Recurse -File))
{
   $source = $item.DirectoryName + "\" + $item
   Write-Host $source
}
```

break and continue

In VFP, if you need to break out of a loop early, you use the `exit` command. In PowerShell use `break` instead. The PowerShell equivalent of VFP's `loop` command, which skips to the next iteration, is `continue`.

Error handling

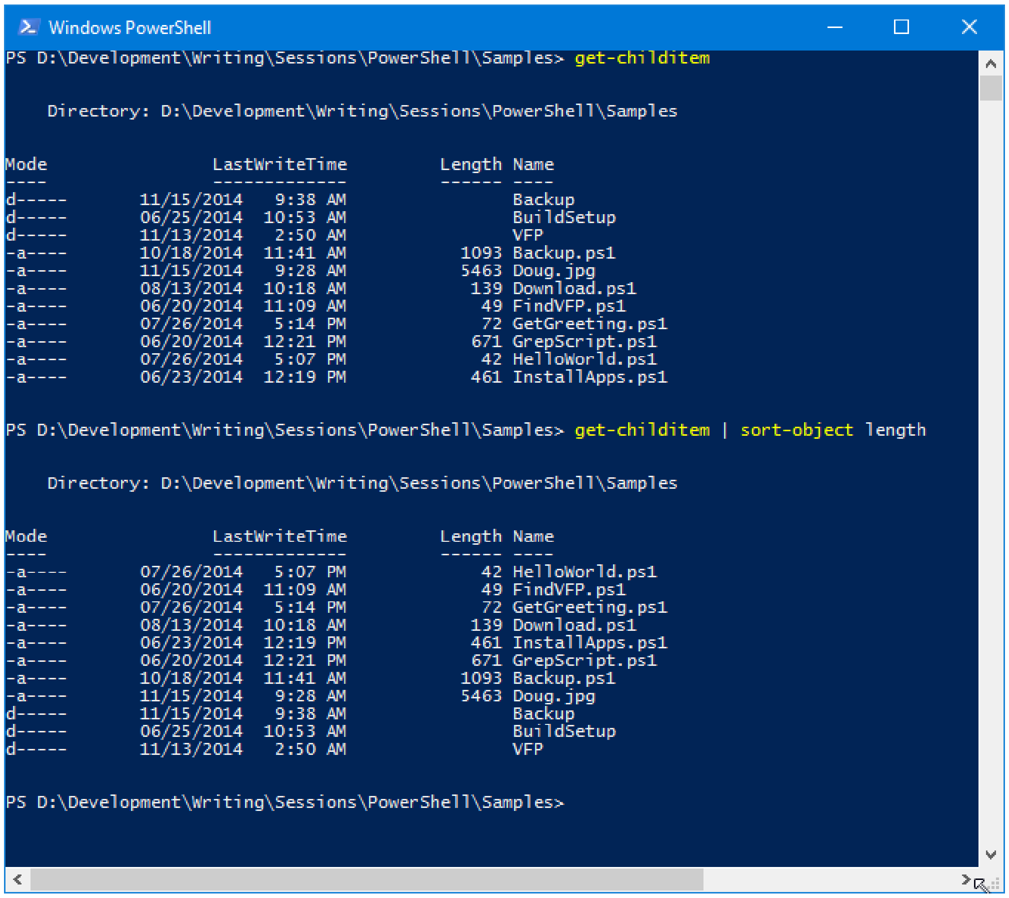
PowerShell has the familiar try/catch/finally construct for handling errors. You can even throw an exception. See the code in the "Backing up files" section of this document for an example.

"$?" is a built-in variable containing the status of the last statement. If it's true, the statement succeeded. It can be used, for example, in branching code to do different things based on whether the code was successful or not.

If an error occurred, the built-in variable $Error can be used to retrieve the error information. The "Backing up files" section of this document also has an example.

Pipelining

Pipelining means sending the output of one command to the input of another. For example, `Get-ChildItem` lists the files in the current folder while `Sort-Object` sorts a collection by some attribute. What happens if we send the output of `Get-ChildItem` to `Sort-Object`? See **Figure 16** for the results. The pipe ("|") character causes the pipelining to occur.

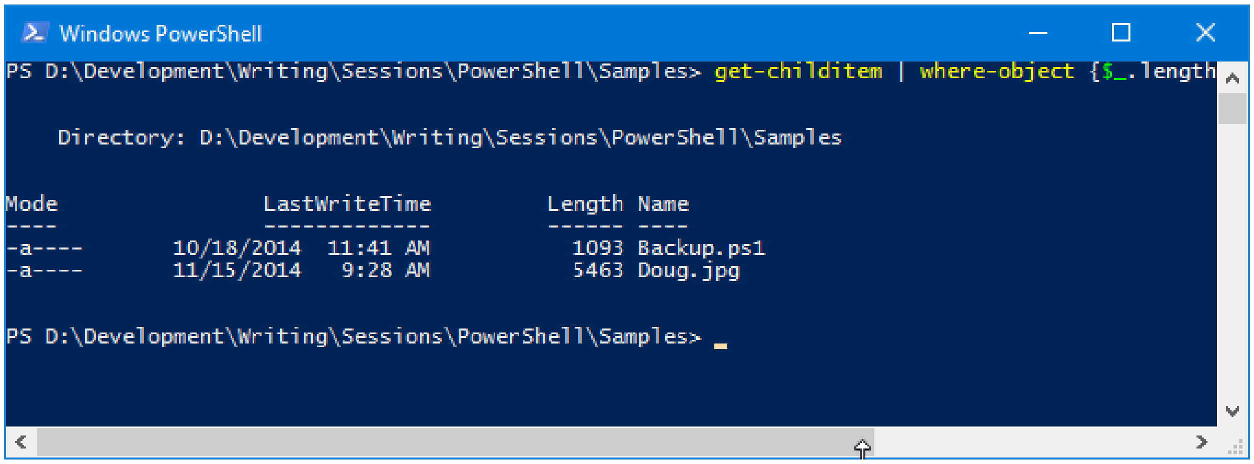


```
PS D:\Development\Writing\Sessions\PowerShell\Samples> get-childitem


    Directory: D:\Development\Writing\Sessions\PowerShell\Samples


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----       11/15/2014   9:38 AM                Backup
d-----       06/25/2014  10:53 AM                BuildSetup
d-----       11/13/2014   2:50 AM                VFP
-a----       10/18/2014  11:41 AM           1093 Backup.ps1
-a----       11/15/2014   9:28 AM           5463 Doug.jpg
-a----       08/13/2014  10:18 AM            139 Download.ps1
-a----       06/20/2014  11:09 AM             49 FindVFP.ps1
-a----       07/26/2014   5:14 PM             72 GetGreeting.ps1
-a----       06/20/2014  12:21 PM            671 GrepScript.ps1
-a----       07/26/2014   5:07 PM             42 HelloWorld.ps1
-a----       06/23/2014  12:19 PM            461 InstallApps.ps1


PS D:\Development\Writing\Sessions\PowerShell\Samples> get-childitem | sort-object length


    Directory: D:\Development\Writing\Sessions\PowerShell\Samples


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----       07/26/2014   5:07 PM             42 HelloWorld.ps1
-a----       06/20/2014  11:09 AM             49 FindVFP.ps1
-a----       07/26/2014   5:14 PM             72 GetGreeting.ps1
-a----       08/13/2014  10:18 AM            139 Download.ps1
-a----       06/23/2014  12:19 PM            461 InstallApps.ps1
-a----       06/20/2014  12:21 PM            671 GrepScript.ps1
-a----       10/18/2014  11:41 AM           1093 Backup.ps1
-a----       11/15/2014   9:28 AM           5463 Doug.jpg
d-----       11/15/2014   9:38 AM                Backup
d-----       06/25/2014  10:53 AM                BuildSetup
d-----       11/13/2014   2:50 AM                VFP


PS D:\Development\Writing\Sessions\PowerShell\Samples>
```

Figure 16. The result of piping the output of Get-ChildItem to Sort-Object.

A special kind of piping cmdlet is a filter. A filter takes something as an input, removes unwanted items, and sends the rest as output. One built-in filter is `Where-Object`, which acts like the WHERE clause in a SQL statement. **Figure 17** shows the result of this command:

```
Get-ChildItem | Where-Object {$_.Length -gt 1000} | Sort-Object Length
```



```
PS D:\Development\Writing\Sessions\PowerShell\Samples> get-childitem | where-object {$_.length

    Directory: D:\Development\Writing\Sessions\PowerShell\Samples

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----       10/18/2014  11:41 AM           1093 Backup.ps1
-a----       11/15/2014   9:28 AM           5463 Doug.jpg

PS D:\Development\Writing\Sessions\PowerShell\Samples> _
```

Figure 17. Using Where-Object to filter a list.

The built-in variable $_ means the current item being examined by the filter cmdlet. In this case, it only returns items whose Length property is greater than 1,000 bytes.

Your own scripts can support pipelining using the built-in variable $input, which contains the pipelined data. For example, this one-line script in a file named FindVFP.ps1 looks through the input piped to it for a file named "vfp9.exe:"

```
$input | Where-Object {$_.Name -eq "vfp9.exe"}
```

This command shows the location of vfp9.exe somewhere under C:\Program Files (x86) (you could use C:\ instead but that would take longer):

```
Get-ChildItem 'C:\Program Files (x86)' – Recurse | FindVFP
```

You can also create your own filter. Although the FindVFP script acts like a filter, it doesn't start executing until $input has been completely filled with data. A filter, on the other hand, starts processing immediately, using the $_ variable for the current item. The only difference between a filter and a function is the "filter" keyword.

For example, here's a filter which filters out anything not containing the specified keyword:

```
filter Grep($keyword)
{
   if (($_ | Out-String) -like "*$keyword*")
   {
      $_
   }
}
```

The following command lists all PRG files in the current folder and all subdirectories:

```
Get-ChildItem –Recurse | Grep .prg
```

Working with .NET

Since PowerShell is based on .NET, it's easy to use .NET classes in PowerShell statements. The `new-object` cmdlet instantiates the specified class; "System" is inferred so you don't have to specify that namespace. For example:

```
$web = New-Object Net.WebClient
$web.DownloadFile("http://www.swfox.net/speakers/doughennig.jpg",
   "$((Get-Location).Path)\Doug.jpg")
```

instantiates the System.Net.WebClient class and downloads a file. You can use `Get-Member` to see a list of the members of the instantiated object.

To call a static member of a class, use the syntax `[Class]::Member`. For example, this gets a new Guid:

```
$guid = [Guid]::NewGuid()
```

Script examples

Let's look at some examples of scripts to learn more about how they work.

Automating application installation

Chocolatey is a PowerShell utility that automates the installation of Windows applications. It saves you having to manually navigate to the publisher's web site, finding the correct download link, and downloading and installing the application. As of this writing, over 9,000 applications can be installed using Chocolatey, including Google Chrome, Firefox, Microsoft SQL Server Express, Microsoft Visual Studio, Notepad++, Filezilla, Git, Dropbox, and so on. Obviously, you'll need valid license keys for some applications. You can read more about Chocolatey and see a list of the available applications at http://chocolatey.org. Scott Hanselman also had an informative blog post about it at http://tinyurl.com/pmdhfv5.

You can install Chocolatey with a single, albeit complex, command:

```
@powershell -NoProfile -ExecutionPolicy unrestricted –Command
   "iex ((new-object net.webclient).DownloadString('https://
   community.chocolatey.org/install.ps1'))" &&
   SET PATH=%PATH%;%systemdrive%\chocolatey\bin
```

This is actually three statements in one:

- @powershell invokes PowerShell from any command window, and the statement prior to "-Command" tells PowerShell to allow unrestricted script execution.
- The text between "-Command" and "&&" instantiates the .NET WebClient class, tells it to download Install.ps1 from chocolatey.org, and runs it.
- The SET PATH statement adds Chocolatey to your path so it can be executed from anywhere.

Using Chocolatey to install an application is as easy as typing `choco install` *`appName`* in a command window. https://community.chocolatey.org/packages lists the applications and their installation command lines.

Of course, you can even automate Chocolatey itself. Here's an example of a simple script that installs Chocolatey and then uses it to download and install Filezilla, Google Chrome, and Notepad++. Think about how quickly such a script could set up a new machine for you, whether a physical or virtual machine, whether local, remote, or in the cloud.

```
# Allow scripts to run; this needs to be executed in the console pane.
#Set-ExecutionPolicy unrestricted -Scope CurrentUser

# Download and install Chocolatey
Invoke-Expression ((new-object net.webclient)
   .DownloadString('https://community.chocolatey.org/install.ps1'))

# Add Chocolatey to the path so we can run it from anywhere
set PATH=%PATH%;%systemdrive%\chocolatey\bin
# Start installing apps
choco install Filezilla
choco install GoogleChrome
choco install NotepadPlusPlus
```

For even more automation, look at Boxstarter (http://boxstarter.org). It can perform completely unattended installs of Windows boxes, including installing the latest Windows updates.

Backing up files

Backup.ps1 performs a backup by zipping files into a file named Backup_*date*.zip in the Backup subdirectory of the specified folder, where *date* is the date the backup was made.

There are a variety of mechanism for zipping files in PowerShell. Two common techniques are using the Compress-Archive cmdlet and the .NET System.IO.Compression.ZipFile class. Compress-Archive is much simpler but has limitations if you use recursion to zip files in subdirectories; see https://tinyurl.com/44xu39n2 for details. Here's an example of Compress-Archive:

```
# Zip all the files in the folder.
$compress = @{
  Path = $sourceFolder + "\*.*"
  CompressionLevel = "Fastest"
  DestinationPath = $zipFileName
}
Compress-Archive -Force @compress
```

We'll use Compress-Archive in this example and System.IO.Compression.ZipFile when we look at automating Thor updates.

The script starts by ensuring a string parameter named $sourceFolder was passed and that it's not null or empty. After assigning the name of the zip file to the $zipFileName variable and ensuring the Backup folder exists, the script zips all the files in the folder. “$?” is a built-in variable containing the status of the last statement, so it’s used to write the appropriate message. If an error occurred, the built-in variable $Error is used to retrieve the error information.

```
Param(
   [parameter(Mandatory = $true)]
   [ValidateNotNullOrEmpty()]
   [string]$sourceFolder
)

# Get the name of the zip file.
$zipFileName = $sourceFolder + "\Backup\Backup_$(Get-Date -f yyyy-MM-dd).zip"

try
{
   # Create the Backup folder if it doesn't exist.
   $exists = Test-Path ($sourceFolder + "\Backup")
   if (-not $exists)
   {
      md ($sourceFolder + "\Backup")
   }

   # Zip all the files in the folder.
   $compress = @{
     Path = $sourceFolder + "\*.*"
     CompressionLevel = "Fastest"
     DestinationPath = $zipFileName
   }
   Compress-Archive -Force @compress

   # Check for errors
   if ($?)
   {
      Write-Host "Backup successful"
   }
   else
   {
      Write-Host "Backup failed"
   }
}

catch
{
   Write-Host "Error occurred at $(Get-Date): $($Error[0].Exception.Message)"
}

finally
{
   Write-Host "Backup finished at $(Get-Date)"
}
```

To see this script in action, pass it the name of a folder to backup:

```
Backup (Get-Location)
```

Backing up SQL Server databases

I have a scheduled task that backs up the SQL Server databases on my server to a ZIP file every Saturday night. The task executes this command:

```
powershell -File C:\Backup\BackupSQLServer.ps1
```

BackupSQLServer.ps1 has this content:

```
$SkipDBs = @('master','model','msdb','tempdb')
Get-SqlDatabase -ServerInstance STONEFIELDSOFTW\SQLEXPRESS |
   Where { $SkipDBs -NotContains $_.Name } |
   foreach { Backup-SqlDatabase -DatabaseObject $_
   -BackupFile "C:\SQLBackup\$($_.Name).bak" }
$compress = @{
  Path = "C:\SQLBackup\*.*"
  CompressionLevel = "Fastest"
  DestinationPath = "C:\ActBackup\BackupSQL $(get-date -f yyyy-MM-dd).Zip"
}
Compress-Archive -Force @compress
```

This code backs up all the databases in the specified SQL Server instance except those listed in the $SkipDBs variable to BAK files in C:\SQLBackup, then zips the files in that folder to a file named YYYY-MM-DD.zip (for example, 2022-03-20.zip).

Automating Thor updates

VFPX (http://vfpx.org) is a GitHub-based VFP community extensions Web site. It consists of over 100 open-source projects. Many of the projects support the Thor Check for Updates (CFU) feature for both installation and updates (Thor is a popular VFP tool manager and itself a VFPX project). There are a variety of ways for a project to support the Thor CFU but the easiest is described at https://vfpx.github.io/thorupdate. It uses a PowerShell script, CreateThorUpdate.ps1 in the ThorUpdater subdirectory of the project folder, to automate the generation of the ZIP file the Thor CFU process downloads. We'll look at that script in sections.

First, it reads from settings from Project.txt: the name (appName), ID (appID), and major version (majorVersion) settings for the project, and a list of files and folders to exclude when creating the ZIP file.

```
# Get the project settings from Project.txt
$appInfo = Get-Content Project.txt
$appName = $appInfo[0].Substring($appInfo[0].IndexOf('=') + 1).Trim()
$appID   = $appInfo[1].Substring($appInfo[1].IndexOf('=') + 1).Trim()
$majorVersion = $appInfo[2].Substring($appInfo[2].IndexOf('=') + 1).Trim()
$exclude = $appInfo[3].Substring($appInfo[3].IndexOf('=') + 1).Trim()
# Ensure no spaces before or after comma
$excludeFiles = $exclude.Split(',')
# Ensure no spaces before or after comma
$exclude = $appInfo[4].Substring($appInfo[4].IndexOf('=') + 1).Trim()
```

```
$excludeFolders = $exclude.Split(',')
```

It then creates a variable containing the path and name for the ZIP file using the ID, changes directory to the parent folder (the folder containing the project files), deletes any existing ZIP file, and creates a list of all files in the project folder and all subdirectories except those specifically excluded.

```
# Get the name of the zip file.
$zipFileName = "ThorUpdater\$appID.zip"

# Move up to the project folder

cd ..
try
{
    # Delete the zip file if it exists.
    $exists = Test-Path ($zipFileName)
   if ($exists)
    {
        del ($zipFileName)
    }

    # Loop through all the files in the project folder except those we don't want
    # and add them to a zip file.
    # See https://stackoverflow.com/questions/15294836/how-can-i-exclude-multiple-
folders-using-get-childitem-exclude
    # for how to exclude folders when -Recurse is used
    $files = @(Get-ChildItem . -recurse -file -exclude $excludeFiles |
        %{
            $allowed = $true
            foreach ($exclude in $excludeFolders)
            {
                if ((Split-Path $_.FullName -Parent) -ilike $exclude)
                {
                    $allowed = $false
                    break
                }
            }
            if ($allowed)
            {
                $_
            }
        }
    );

    # See https://stackoverflow.com/questions/51392050/compress-archive-and-preserve-
relative-paths to compress
    # exclude directory entries and generate fullpath list
    $filesFullPath = $files |
      Where-Object -Property Attributes -CContains Archive |
      ForEach-Object -Process {Write-Output -InputObject $_.FullName}
```

Next it creates the zip file. Because of Compress-Archive's limitation in handling subdirectories (it works but it creates duplicate entries in the ZIP file), we'll use the .NET

System.IO.Compression.ZipFile class instead, which as you can see from the code is considerably more complicated but much more flexible.

```
    #create zip file
    Add-Type -AssemblyName System.IO.Compression, System.IO.Compression.FileSystem
    $zip = [System.IO.Compression.ZipFile]::Open((Join-Path -Path
      $(Resolve-Path -Path ".") -ChildPath $zipFileName),
      [System.IO.Compression.ZipArchiveMode]::Create)

    #write entries with relative paths as names
    foreach ($fname in $filesFullPath)
    {
        $rname = $(Resolve-Path -Path $fname -Relative) -replace '\.\\',''
        $zentry = $zip.CreateEntry($rname)
        $zentryWriter = New-Object -TypeName System.IO.BinaryWriter $zentry.Open()
        $zentryWriter.Write([System.IO.File]::ReadAllBytes($fname))
        $zentryWriter.Flush()
        $zentryWriter.Close()
    }

    # clean up
    Get-Variable -exclude Runspace |
      Where-Object {$_.Value -is [System.IDisposable]} |
      Foreach-Object {$_.Value.Dispose(); Remove-Variable $_.Name};
```

It then updates the version file for the project, named *appID*Version.txt, which Thor CFU downloads and examines to decide if a newer version of the project is available, by reading the contents of Version.txt, replacing the current date and major version number in the file (the complete version number is the major version number plus the number of days since 01/01/2000, such as “1.0.8120” on March 26, 2022), and writing out *appID*Version.txt. Finally, it changes directory back to the ThorUpdater folder.

```
    # If the zip file was created, read in the Version source file, replace
    # placeholders, and write out the project version file.
    if ($?)
    {
        # Read in the update file
        $date = Get-Date
        $file = $appID + 'Version.txt'
        (Get-Content ThorUpdater\Version.txt).
            Replace('date()', 'date(' + $date.Year + ',' + $date.Month + ',' +
                $date.Day + ')').
            Replace('APPNAME', $appName).
            Replace('MAJORVERSION', $majorVersion) |
            Set-Content ThorUpdater\$file
        Write-Host "Update creation successful"
    }
    else
    {
        Write-Host "Zip creation failed"
      pause
    }
}
```

```
catch
{
    Write-Host "Error occurred at $(Get-Date): $($Error[0].Exception.Message)"
   pause
}

finally
{
# Move back to the ThorUpdater folder
    cd ThorUpdater
}
```

A project manager simply has to ensure Project.txt and Version.txt contain the correct settings for their project and run CreateThorUpdate.ps1 when they're ready to release a new version of their project.

Automating application deployment

Deployment of an application is a very important step. After all, if you can't install the application on someone's system, there's no point in writing it. Deployment is often a complex job involving many steps. If any of those steps are forgotten or done incorrectly, the user may end up with a faulty installer. That leads to extra support work and even mistrust of your abilities. Getting this part of the application lifecycle right is crucial.

I use a carefully crafted checklist to make sure I do all the steps necessary to deploy my applications. As they got more complicated, the list got longer and longer and therefore more likely to be error-prone. Automating as many tasks as possible helps to ensure the deployment process is both accurate and efficient.

Many of the tasks involved in deployment involve files: copying, moving, deleting, creating, changing, digitally signing, and so on. Sounds like a perfect job for PowerShell.

BuildSetup.ps1, included with the sample files accompanying this document, is a stripped-down version of an actual script we use to deploy Stonefield Query Enterprise. Let's look at it one section at a time.

The script starts with some helper functions. RemoveFile removes the specified file if it exists. The script uses this instead of the built-in Remove-Item because that cmdlet throws an exception if the file doesn't exist.

```
function RemoveFile([string]$path)
{
   If (Test-Path $path)
   {
      Remove-Item $path
   }
}
```

Pause waits for a keypress.

```
function Pause
```

```
{
   Write-Host "Press any key to continue ..."
   $x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")
}
```

As its name suggests, FTPUpload uploads the files in the specified folder, including those in subdirectories, to the specified FTP path. Note that the FTP server name, user name, and password are read from a file (ftp.txt) so they aren't exposed in the script.

```
function FTPUpload($sourceFolder, $targetFolder)
{
   # Get the FTP server settings.
   $settings = Get-Content DontDeploy\ftp.txt
   $ftp = $settings[0]
   $user = $settings[1]
   $pass = $settings[2]

   # Create a WebClient object.
   $webclient = New-Object System.Net.WebClient
   $webclient.Credentials = New-Object System.Net.NetworkCredential($user, $pass)
   # Upload all files in all subdirectories of the folder.
   foreach($item in (Get-ChildItem -Path $sourceFolder -Recurse -File))
   {
      try
      {
         # Get the source file.
         $source = $item.DirectoryName + "\" + $item;

         # Get the directory on the FTP site.
         $directory = "";
         if ($item.DirectoryName.Length -gt 0)
         {
            $directory = $item.DirectoryName.Replace($sourceFolder, "")
            $directory = $directory.Replace("\", "/")
         }
         $directory += "/"

         # Create the complete target path and upload it.
         $target = $ftp + $targetFolder + $directory + $item

         Write-Host Uploading $item.FullName to $target

         $uri = New-Object System.Uri($target)
         $webclient.UploadFile($uri, $item.FullName)
      }
      catch [Exception]
      {
         # Log if something failed.
         Write-Host $_.Exception.Message;
         $content = "`r`nFile " + $item.FullName + "`r`n" +
            $_.Exception.Message + "`r`n"
         Add-Content BuildLog.txt $content
      }
   }
}
```

The body of the script starts by deleting BuildLog.txt if it exists and getting the current path:

```
# Erase the log file.
RemoveFile BuildLog.txt

# Get the current folder.
$currFolder = Get-Location
```

The next step requires some explanation. We want to automate the building of our application's executable, so BuildProject.prg takes care of that. This code opens MyApp.pjx, sets its version number to "1.0." plus the number of days after 01/01/2000 (for example, building the EXE on June 19, 2014 results in a version number of "1.0.5283"), adjusts the copyright to use the current year, and builds MyApp.exe. Any errors are logged to BuildLog.txt.

```
local lcVersion, ;
   loProject, ;
   lcCopyRight, ;
   llReturn, ;
   laError[1]
lcVersion = GetVersionNumber()
modify project MyApp\MyApp noshow nowait
loProject               = _vfp.ActiveProject
loProject.VersionNumber = lcVersion
lcCopyRight             = loProject.VersionCopyright
if not empty(lcCopyRight)
   loProject.VersionCopyright = stuff(lcCopyright, at('-', lcCopyright) + 1, ;
      4, transform(year(date())))
endif not empty(lcCopyRight)
llReturn = loProject.Build('MyApp\MyApp.exe', 3, .F., .T.)
loProject.Close()
if not llReturn
   aerror(laError)
   strtofile(transform(datetime()) + ': building MyApp failed: ' + ;
      laError[2] + chr(13) + chr(10), 'BuildLog.txt', .T.)
endif llReturn
quit

* Get the version number: 1.0. + the number of days after 01/01/2000.

function GetVersionNumber
local lnJulian, ;
   lcJulian, ;
   lcVersion
lnJulian  = val(sys(11, date())) - val(sys(11, {^2000-01-01}))
lcJulian  = padl(transform(lnJulian), 4, '0')
lcVersion = '1.0.' + lcJulian
return lcVersion
```

To execute this code from PowerShell, we must run VFP and tell it to use Build.fpw as the configuration file because that file tells VFP to execute BuildProject.prg using this command:

```
command = do BuildProject.prg
```

Here's the part of the PowerShell script that finds where VFP is located by looking in the Windows Registry (I could have hard-coded the path but this is more flexible) and running it. Note that output is piped to Out-Null, which forces PowerShell to wait until BuildProject.prg is done and VFP terminates.

```
$regEntry = Get-ItemProperty –Path
Registry::HKEY_CLASSES_ROOT\Visual.FoxPro.Application.9\shell\open\command
$value = $regEntry.'(default)'
$file = $value.Substring(1, $value.IndexOf('"', 2) - 1)
$VFPexe = Get-ChildItem $file
$buildConfig = "-c$currFolder\Build.fpw"
& $VFPexe –t $buildConfig | Out-Null
```

Now that the executable is built, the next job is to digitally sign it. First, we find the location of SignTool.exe using similar code to how we found VFP, then we read the password for the certificate from a file (CertPW.txt), and finally sign MyApp.exe.

```
$regEntry = Get-ItemProperty -Path
"Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.1A"
$value = $regEntry.'InstallationFolder'
$file = $value + "Bin\signtool.exe"
$signTool = Get-ChildItem $file

# Get the password for the certificate.
$password = Get-Content DontDeploy\certpw.txt
$parms = 'sign', '/f', "$currFolder\DontDeploy\mycert.pfx", '/p', $password

# Sign the exe.
Write-Host "Signing executable"
$signParms = $parms + '/d', 'My Application', 'MyApp\myapp.exe'
& $signTool $signParms
```

The next step is building Setup.exe using Inno Setup. As before, we look in the Windows Registry to find out the location of the Inno compiler and have it run MyApp.iss.

```
$regEntry = Get-ItemProperty -Path
Registry::HKEY_CLASSES_ROOT\InnoSetupScriptFile\Shell\Compile\Command
$value = $regEntry.'(default)'
$file = $value.Substring(1, $value.IndexOf('"', 2) - 1)
$exe = Get-ChildItem $file
$innoCompiler = $exe.DirectoryName + "\iscc"

# Build Setup.exe.
$signParms = "/sStandard=$signTool $parms $" + "p"
& $innoCompiler $signParms myapp.iss
```

The last major task is uploading Setup.exe to the FTP site using the FTPUpload function we saw earlier.

```
Write-host "============ Uploading Setup.exe"
$sourcePath = Join-Path $currFolder "Output"
FTPUpload $sourcePath "httpdocs/Downloads/Sample"
```

Finally, if anything went wrong, display a message.

```
If (Test-Path BuildLog.txt)
{
   Write-Host "See BuildLog.txt for a list of errors in the process."
   Pause
}
```

Now, when I'm finished coding and testing my application, I simply run BuildSetup.ps1 and when the smoke clears, I have a digitally signed setup executable sitting on my FTP site ready for my users to download and install.

Calling PowerShell from VFP

Executing PowerShell commands and scripts from a VFP application is a little tricky but doable. The secret is to use Rick Strahl's wwDotNetBridge (http://tinyurl.com/l8axhx2) to talk to classes in the .NET System.Management.Automation assembly. RunningPSFromVFP.PRG, included with the downloads for this document, shows how to do it. This example displays the running processes.

```
* Instantiate wwDotNetBridge.

set procedure to wwDotNetBridge
loBridge = createobject('wwDotNetBridge', 'V4')

* Load the PowerShell assembly. Get the location using
* [psobject].assembly.location in PowerShell.

llReturn = loBridge.LoadAssembly('C:\Windows\Microsoft.Net\assembly\' + ;
   'GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35' + ;
   '\System.Management.Automation.dll')
if not llReturn
   messagebox(loBridge.cErrorMsg)
   return
endif not llReturn

* Create a PowerShell object.

loPS = loBridge.InvokeStaticMethod('System.Management.Automation.PowerShell', ;
   'Create')

* Add a command to execute. You can also call AddScript to add a script.

loBridge.InvokeMethod(loPS, 'AddCommand', 'Get-Process')

* We want to call the Invoke method of the PowerShell object but the following
* statement gives an "ambigous match found" error because there are two
* overloads of Invoke that accept no parameters but with different return
* values and wwDotNetBridge can't tell which to call.
*
* loResult = oBridge.InvokeMethod(loPS, 'Invoke')
*
* Instead, we'll go through the methods of the PowerShell object, look for the
* first Invoke method, and call it. It returns Collection<PSObject>. Note that
* we need an empty array for the second parameter of Invoke.

loType    = loBridge.InvokeMethod(loPS, 'GetType')
loMethods = loBridge.InvokeMethod(loType, 'GetMethods')
loArray   = loBridge.CreateArray()
for lnI = 0 to loMethods.Count - 1
   loMethod = loMethods.Item(lnI)
   if loMethod.Name = 'Invoke'
      loResult = loBridge.InvokeMethod(loMethod, 'Invoke', loPS, loArray)
      exit
   endif loMethod.Name = 'Invoke'
next lnI
* Create a cursor to hold the process information.

create cursor Processes (Name C(50), ID I)

* Copy the result collection to an array and process each entry.

loArray.FromEnumerable(loResult)
for lnI = 0 to loArray.Count - 1
   loItem = loArray.Item(lnI)

* We want to get the ProcessName member of the object but this doesn't work:
* Members is a PSMemberInfoCollection<PSMemberInfo> but accessing it like this
* gives the collection, not the PSPropertyInfo object.
*
* loName    = loBridge.GetPropertyEx(loProcess, 'Members["ProcessName"]')
*
* Instead, get the Members collection and go through it, looking for the
* members we're interested in.

   loMemb    = loBridge.GetPropertyEx(loItem, 'Members')
   loMembers = loBridge.CreateArray()
   loMembers .FromEnumerable(loMemb)
   append blank
   for lnJ = 0 to loMembers .Count
      loMember = loMembers.Item(lnJ)
      try
         lcName = loBridge.GetPropertyEx(loMember, 'Name')
         do case
            case lcName = 'ProcessName'
               replace Name with loBridge.GetPropertyEx(loMember, 'Value')
            case lcName = 'Id'
               replace ID with loBridge.GetPropertyEx(loMember, 'Value')
         endcase
      catch
      endtry
   next lnJ
next lnI

* Display the processes.

go top
browse
```

Resources

There are a ton of resources available for all things PowerShell; simply Google "powershell" and be prepared to be overwhelmed! A good place to start, though, is https://docs.microsoft.com/en-us/powershell, Microsoft's PowerShell center. It has links for webcasts, downloads, blogs, scripts, tutorials, and more.

Summary

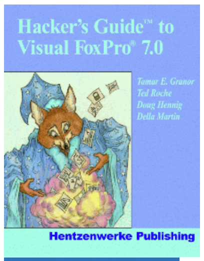
Windows PowerShell allows you to automate just about any task you can think of. It's relatively easy to learn, is very powerful, and once you've created a script, can save you a lot of time doing tasks the manual way.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Query; the award-winning Stonefield Database Toolkit (SDT; now open source); the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

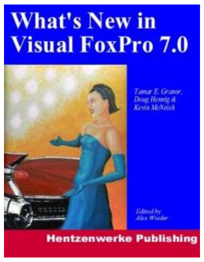
Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the Southwest Fox (http://www.swfox.net) and Virtual Fox Fest (https://virtualfoxfest.com) conferences. He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.org). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).


Hacker's Guide™ to Visual FoxPro® 7.0

Tamar E. Granor, Ted Roche, Doug Hennig, Della Martin

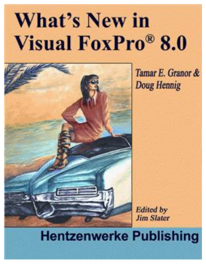
Hentzenwerke Publishing


What's New in Visual FoxPro 7.0

Tamar E. Granor, Doug Hennig & Kevin McNeish

Edited by Alex Wieder

Hentzenwerke Publishing


What's New in Visual FoxPro® 8.0

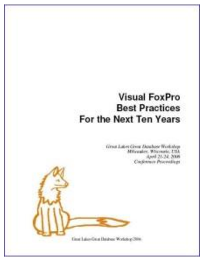
Tamar E. Granor & Doug Hennig

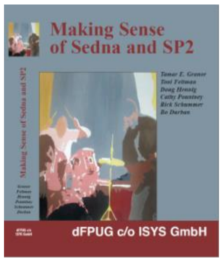
Edited by Jim Slater

Hentzenwerke Publishing

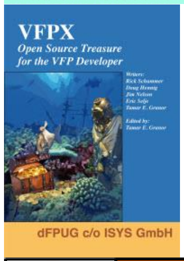
What's New in Nine: Visual FoxPro's Latest Hits

Hentzenwerke Publishing


Visual FoxPro Best Practices For the Next Ten Years


Making Sense of Sedna and SP2

dFPUG c/o ISYS GmbH


VFPX Open Source Treasure for the VFP Developer

dFPUG c/o ISYS GmbH