PowerShell and everything you need to know about it

What is PowerShell? Microsoft PowerShell is a command line utility and scripting language that is a powerful tool for administrators that helps you automate a wide range of computer and network tasks.

What is PowerShell?

Microsoft PowerShell is a command line utility and scripting language that is a powerful tool for administrators that helps you automate a wide range of computer and network tasks. PowerShell contains Command Prompt components and is built on the .NET framework. If you are learning about network administration, then you need to know that PowerShell is the tool chosen by information technology administrators to manage large networks.

Learning how to use PowerShell will help simplify many tedious daily tasks. You can also make system-wide changes over the network without having to make individual adjustments to each server. PowerShell is becoming an essential part of running hybrid cloud environments.

PowerShell has many different uses to help you work more optimally and keep your network running efficiently. The most basic uses include: scheduling daily updates on systems, generating reports for current processes, cyclical services, and more. It's true that many of these tasks can be done through the GUI, but the point of PowerShell is to do them faster.

If you have a regular maintenance task that takes a few minutes to set up, you can script the same functions in a single command, named on PowerShell. So, next time, you just need to open that script and it will run in the background. Mastering PowerShell's scripting logic, understanding how objects and vars (variables) work, and deploying it intelligently across your network will make you wonder why you didn't use PowerShell sooner.

This tutorial will cover the basics of PowerShell, helping those new to IT, especially if you are familiar with Windows Command Prompt. The article will introduce how to use basic tools and commands, how to manipulate files and folders, understand objects, use variables, and manage remote servers.

A brief history of Windows command-line utilities

After the advent of Windows NT, CMD.EXE became the command line utility for Windows. Although CMD.EXE inherits some components of its DOS predecessor (COMMAN.COM), it is still based on a rather 'primitive' scripting language: using Windows Command files (.CMD and .BAT). The addition of Windows Scripting Host and the VBScript and JScript languages ??has significantly enhanced the scripting capabilities of the utility.

These technologies are a fairly balanced combination of advanced command line utilities and scripting environments. Actually, the issue of how many CMD.EXE, .CMD and Windows Scripting Host files can be manipulated is not a real concern. What makes people complain and worry the most is completing some seemingly simple tasks.

Using a 'framework' of scripting and command line tools, any moderately compiled script requires a combination of batch commands, Windows Scripting Host and standalone executions. Each script uses different conventions for executing and requesting, parsing, and returning data.

Weak variable support in CMD.EXE, inconsistent interfaces, and limited access to Windows settings, combined with another weakness, make command-line scripting more difficult to deploy and use . You will probably immediately wonder what 'another weakness' is here? Please tell me that it is plain text. In these technologies, everything is in text format. The output of a command or script is text and must be parsed and reformatted to act as input for the next command. This is the basic starting point that PowerShell takes from all traditional utilities.

Introducing PowerShell tools, commands, and modules

The three concepts introduced in this section are just the very basics to understanding the key concepts that form the foundation of PowerShell. You will need to spend more time to learn and master more advanced concepts when approaching PowerShell commands.

PowerShell Tools

PowerShell is installed by default in Windows 10, Windows 7, Windows Server 2008 R2, and later versions of Windows. New versions of PowerShell add many new features and "cmdlets" (Microsoft's term for PowerShell commands - pronounced "command-lets") and are installed using the corresponding version of Windows Management Framework (WMF).).

Currently WMF 5.1 is the latest version recommended for use. In some cases, some new features depend on the operating system in addition to the WMF version. For example, Windows 8 and Windows Server 2012 support the Test-NetConnection cmdlet, which allows you to test a connection to a specific TCP/IP port, but this cmdlet is not available in Windows 7 even when running the latest version of WMF .

On most Windows systems, users will have two PowerShell environments available, PowerShell console and PowerShell ISE (Integrated Scripting Environment). The PowerShell console appears just like the traditional command line, but with all the full functionality of PowerShell behind it. Variable names, loops, command autocompletion, and piping are available from the PowerShell console.

For deeper uses (like building scripts), PowerShell ISE provides command autocompletion, code highlighting, and Microsoft Intellisense code completion to assist you in creating and testing PowerShell code. PowerShell ISE also allows you to work with multiple PowerShell scripts simultaneously using tabbed navigation.

cmdlets in PowerShell

The foundation of PowerShell commands is cmdlets. Microsoft implemented several design strategies when creating cmdlets in PowerShell.

The first is the ability to easily infer cmdlet names, or at least make them more discoverable. PowerShell commands or cmdlets are also designed to be easier to use, with standardized syntax, making it easier to create

scripts from the command line interface.

The cmdlet uses the Verb-Noun (verb - noun) format as in Get-Service, Stop-Service, or Import-Csv. The verb part of the cmdlet's name will indicate the action performed on the noun. Typically, cmdlets used to get information will have the verb **Get** in the name, for example **Get-Process** or **Get-Content** . The command used to edit something usually starts with the verb **Set** , to add a new entity somewhere it usually starts with **Add** or **New** .

Second, commonly used parameters in PowerShell are also named in a standardized way. For example, the **-ComputerName** parameter allows the cmdlet to be executed on one or more remote computers. -Credential is used to provide a credential object, containing the user's credentials, to run the command as a specific user.

Modules in PowerShell

You can use alias for both cmdlets and parameters to save keystrokes, shortening the overall length of the command (very useful when you combine multiple commands together). Although these aliases do not always use standard naming conventions, they still reflect traditional command-line utilities.

In PowerShell, aliases like DIR, CD, DEL, and CLS correspond to the cmdlets Get-ChildItem, Set-Location, Remove-Item, and Clear-Host respectively. Parameter aliases can work in two ways: they can use an alias predefined by the cmdlet, or they can be aliased by entering enough characters to produce a unique match between the parameters specified. cmdlet support.

Manage files and folders

Most system administrators have to manipulate files and folders during their work, which can be moving folders to another location on the server, storing log files or searching for files. big. In cases where the same operations are repeated across multiple files, using PowerShell to automate them will be an effective time-saving solution.

To find files and folders, one of the first command line tools that administrators would learn in old computers was the dir command. Dir will list files and directories contained in the specified directory.

PowerShell has a similar command in the form of the **Get-ChildItem** Cmdlet . Get-ChildItem allows you to quickly build a list of files in a directory in such a way that you can operate on this file through a pipe command or assign the output to a variable.

Get-ChildItem can be used simply by providing a path, through the pipeline, using the *-Path* parameter or immediately after the cmdlet name. To adjust the response returned by Get-ChildItem, it is necessary to consider several parameters made available by the cmdlet.

The -Filter parameter is one way you can search for files. By default, Get-ChildItem only returns direct children of the target folder. This functionality can be extended by using *-Recurse* , which recursively searches for directories contained in the current directory.

In PowerShell 4.0 Get-ChildItem added the ability to limit results to files or directories using the *–File* or *–Directory* switch . Previous versions of PowerShell had to pass the result to Where-Object, filtering on the PSIsContainer property to make this determination. Examples of both techniques used to return directories contained in C:Users are shown here:

```
Get-ChildItem C:Users -Directory Get-ChildItem C:Users | Where-Object {$_.PSIsCo
```

To detect hidden or system files, use *-Force* . Get-ChildItem in PowerShell 4.0 and later can also be used to return only hidden, read-only, or system files using *-Hidden, -ReadOnly* , and *–System* respectively. Similar functionality could be achieved in previous versions by filtering the Mode property with Where-Object:

```
Get-ChildItem C:Users | Where-Object {$_.Mode -like '*R*'}
```

Check if the file exists

Usually when working with files, all we need to know is whether the file exists or whether the directory path is valid. PowerShell provides a cmdlet to do this in the form of Test-Path, which returns true or false.

Test-Path is used as a preventative step before attempting to copy or delete specific files.

Copy, move and delete files

Copy-Item: Copies one or more files or folders from a location, specified by the -Path parameter, to the location specified by the -Destination option.

Move-Item: Move files or folders.

When directory structures are being copied or moved, *-Recurse* should be used for the cmdlet to take action on the directory and its contents. In some cases, *-Force* is needed , such as when a read-only file is overwritten by a copy operation.

Remove-Item: Delete files and folders.

-Force switch should be used when encountering file or read-only and *-Recurse* should be used when deleting a directory and its contents.

Use PowerShell -WhatIf and -Confirm

Before performing a serious, mass deletion of something, use *-WhatIf* . -WhatIf allows you to see what will happen if you run a script or command, does it have any potential negative impact of deleting important business data. It's also important to note that -WhatIf is not limited to file operations, it is widely used in PowerShell.

For scripts that you intend to run manually, or worse, have dependent commands that run manually, consider using *-Confirm* . This allows you to request user interaction before the activity actually takes place.

PowerShell Script = Batch Files on Steroids

PowerShell itself is written in the .NET language and is based primarily on the .NET Framework. Therefore, PowerShell is designed as an object-oriented utility and scripting language. Everything in PowerShell is considered an object with full functionality of the .NET Framework. A command that exposes a collection of objects that can be used using the properties and methods of that object type. When you want to pipe the output of one command to another command, PowerShell actually passes the object, not just the text output of the first command line. This gives the next command full access to all properties and methods of the object in the pipeline.

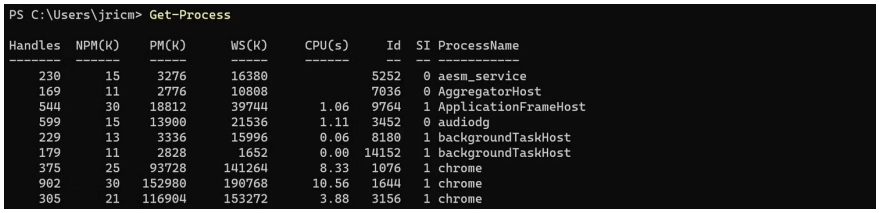
Treating everything as an object and the ability to pass objects between commands is a major change in theory for command line utilities. That said, PowerShell still works like a traditional shell daemon. Commands, scripts, and executables can be typed and run from the command line and the results are displayed in text format.

Windows .CMD and .BAT files, VBScripts, JScripts and executables that work inside CMD.EXE, all still run in PowerShell. However, because they are not object-oriented, they do not have full access to the objects created and used in PowerShell. These legacy scripts and executables will still treat everything as text, but you can combine PowerShell with a number of other technologies. This is very important if you want to start using PowerShell with a collection of existing scripts but cannot convert them all at once.

Explanation of PowerShell parameters

Cmdlets can accept parameters to change their behavior. When running a Cmdlet or function, you can provide parameter values ??to specify what, when, where, and how each PowerShell command runs.

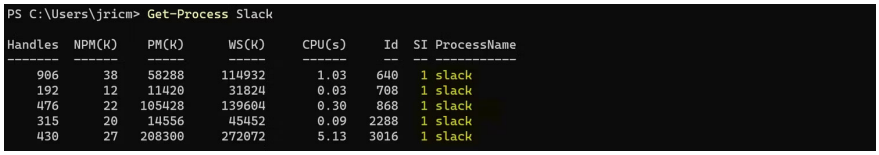
For example, Get-Process will get and list all active processes in your operating system:



But what if you just want to get a specific progress? You can do so using parameters. For example, to get all Slack processes, you can use the Name parameter with the Get-Process Cmdlet:

```
Get-Process -Name Slack
```

You will then only see processes named "slack":

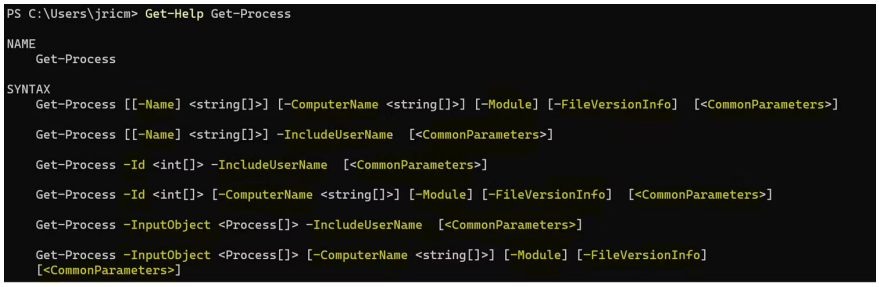


Tip : Some parameters are "positional" which means their names are optional. In this case, **Get-Process -Name Slack** and **Get-Process Slack** both perform the same task.

Each Cmdlet will accept different types of parameters. Use the Get-Help command to view the Cmdlet's accepted parameters in the SYNTAX section.

```
Get-Help Get-Process
```

You will see a list of all possible ways you can run the given Cmdlet:



In this case, the Get-Process Cmdlet accepts parameters such as **Name, Id, ComputerName, Module, FileVersionInfo** and other common parameters. The symbols here mean:

Symbol	Name	Meaning
	Drum	The parameter does not accept input
-	Hyphen	Specifies the parameter name
>	Angled brackets	Placeholder for text
[]	Parentheses	The parameter can accept one or more values
{}	Angled brackets	The parameter accepts a set of values

Parameters accept a set of values ??that will indicate the type of data they require, such as string, integer, boolean, or DateTime. For example, the following command:

```
Get-Process [[-Name]
       ]
```

. means that the **Name** parameter accepts one or more string values, while this command:

```
Get-Process -Id
```

. means that the **Id** parameter accepts one or more integer values.

The previous Get-Process example used the Name parameter to narrow the results. However, if you want to narrow it down to a more specific process, you can use the **ID** parameter , which requires an integer as stated in its syntax.

```
Get-Process -Id 3016
```

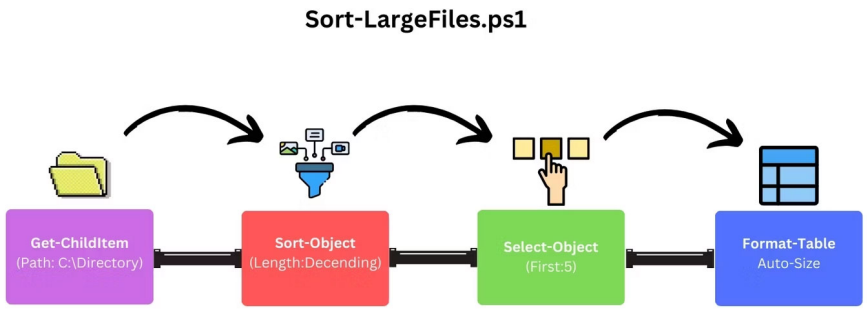
Then you will see only one process in the list:

```
PS C:\Users\jricm> Get-Process -Id 3016

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    424      26   178760     236764      17.53    3016   1 slack

PS C:\Users\jricm>
```

Create pipeline

PowerShell treats all data as objects. To build a script, these objects run through a series of Cmdlets or functions connected by the pipe symbol (|). Choosing the right Cmdlets and connecting them in the right sequence using a pipeline is important for an efficient script.

Suppose you are creating a script to sort and display the 5 files that take up the most storage space in a folder. There are more powerful ways to create file sorting scripts, but the following simple one is easy to understand:



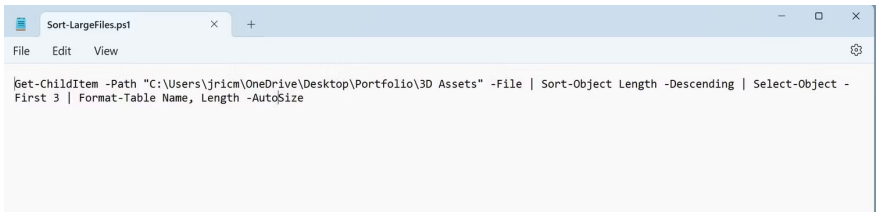
To do this in PowerShell, use a pipeline that looks something like this:

```
Get-ChildItem -Path "C:Directory" -File | Sort-Object Length -Descending ` | Sel
```

Save the pipeline as a PS1 script

Now that we have a working pipeline, you can save it as a PS1 script file so you don't have to import it every time you use it.

The simplest way to create a PS1 file is to paste your script into Notepad and save the file with the .ps1 extension.



After creating the PS1 file, you can use it in PowerShell by running **./ScriptName.ps1** :

```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\jricm> ./Sort-LargeFiles.ps1

Name                                                                  Length
----                                                                  ------
2D Large Plot Of Grass.skp                                           6158056
24-grass.rar                                                         5650120
10439_Corn_Field_v1_L3.123c80b91965-ab50-4dd0-8508-3847dcd0c84e.zip 1286814


PS C:\Users\jricm>
```

Tip : If you get permission errors, the quickest fix is ??to run PowerShell with admin rights when running your script.

Congratulations! You can now create PowerShell PS1 scripts.

Example of a PowerShell Script

Reading and understanding the wonders of new technology is one thing, but examining and using it is another! In the remainder of this article, we will develop a PowerShell script to demonstrate its capabilities and uses.

DIR is one of the most common commands in CMD.EXE. This command outputs all files and subfolders contained in a parent folder (as shown in Figure 1). Along with the name of each object, the information provided also includes the date and time of the last update and the size of each file. DIR also displays the aggregate size of all files in the folder, as well as the total number of files and total subfolders.

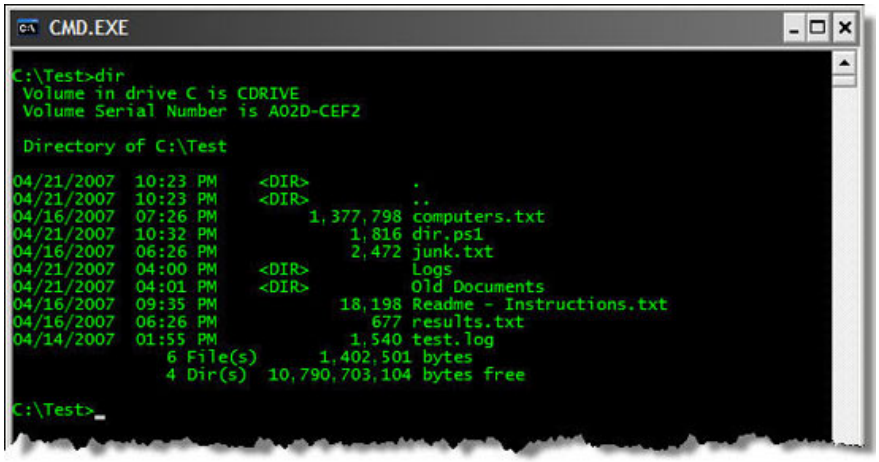


Figure 1

Running DIR in PowerShell also produces a directory listing like Figure 2, but it's a little different. PowerShell does not have a DIR command but instead has Get-ChildItem, which also performs the same function. In PowerShell, DIR is an alias for Get-ChildItem. I do not intend to go into detail about aliases in this article. You can think of DIR in PowerShell as an abbreviation for Get-ChildItem.

DIR in PowerShell provides much of the same information as mentioned above: a list of files and folders, the date and time of the last update and the size of each file. However, it lacks the summary information that DIR in CMD.EXE provides: total size of all files in the folder, total number of files and total number of subfolders.

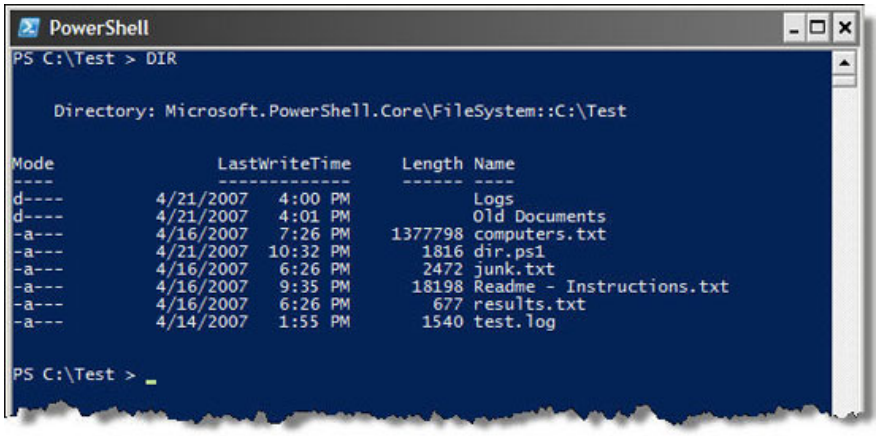


Figure 2

For the example scenario, you will need to create a PowerShell script that simulates the CMD.EXE DIR command. Below I will explain the most essential parts of a script.

DIR.PS1: Header

A PowerShell script includes PowerShell commands in a plain text file with the .PS1 extension. Instead of DIR, you will use a text file called DIR.PS1.

To run the script, type the following command at the PowerShell screen:

```
.DIR.PS1 X:Folder
```

Where X is the drive partition letter (like C, D, E) and Folder is the folder name.

If you want to know some information about drive partitions, you will have to use Windows Management Instrumentation (WMI). Details about WMI are beyond the scope of this article so we will not discuss them here. But the PowerShell code below is quite easy to understand without needing WMI's help. You can create a '$filter' variable to use with the Get-WmiObject command. This filter variable tells the Get-WmiObject command that you only want information about a specific drive. The results of the Get-WmiObject command are stored in a variable called $volInfo. Remember, in PowerShell everything is an object; $volInfo is now also a result object returned from Get-WmiObject.

```
$filter = "DeviceID = '" + $drive + ":'" $volInfo = Get-WmiObject -Class Win32_L
```

You can now access all the objects and methods associated with the object. The serial number of the drive partition can be accessed through the VolumeSerialNumber property. The returned information is an 8-character string of numbers. But usually you want to format it as four separate numbers, separated by a hyphen. can be done similarly as in the line below. The hyphen at the end of the first line is the line continuation character in PowerShell. It basically just tells PowerShell that the line doesn't break but includes the next line. Line separation is not necessary when writing code, but to reduce the width and make the code easier to read, you should do so.

```
$serial = $volInfo.VolumeSerialNumber.SubString(0, 4) + "-" + ` $volInfo.VolumeS
```

Now that you have a $volInfo object, you can write the DIR header information to the screen. If the drive has no name, the text written to the screen will be slightly different than if the drive has a name. A simple If-Else

statement is used to check whether the VolumeName property is an empty string or not. The Write-Host command is used to write each line of command to the screen.

```
If ($volInfo.VolumeName -eq "") { Write-Host (" Volume in drive " + $drive + " ha
```

The ' `n ' character at the beginning and end of the Write-Host command is used to insert new lines before and after text. The Write-Host command adds a newline to the end of each line. So the effect of ' `n ' is to create blank lines before and after the line of text.

Did you notice the '-eq' in the If statement? It's an equality comparison operator. The table below shows you all the comparison operators:

-eq, -ieq	Compare equals
-ne, -ine	Comparison is not equal
-gt, -igt	Greater comparison
-ge, -ige	Compare greater than or equal
-lt, -ilt	Compare smaller
-le, -ile	Compare less than or equal

The -i character in front of comparison operators indicates that the operator is case-insensitive.

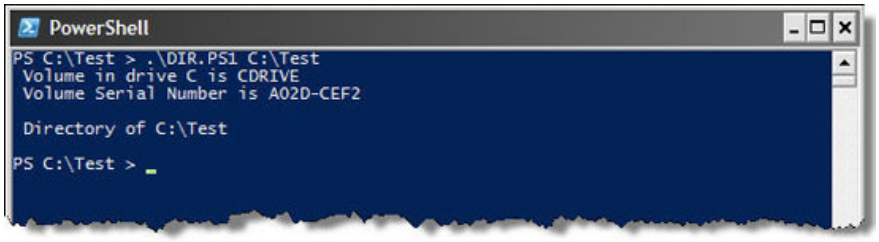


Figure 3: Output of the script you currently have

DIR.PS1: List of files/folders

Now you're ready to display the contents and properties of this folder. The first thing to do is call the PowerShell Get-ChildItem command to get the set of files and pass it to the script as a parameter. The Get-ChildItem command will retrieve a collection of file and folder objects, not just the names but also pipe these objects directly into the Sort-Object command to sort them. By default, the Sort-Object command will sort objects based on the Name attribute. So you don't need to describe any other parameters. The sorted collection of objects is then stored in a variable named $items.

```
$items = Get-ChildItem $args[0] | Sort-Object
```

Once you have a set of file and folder objects, you need to iterate over them and display the appropriate features. The command used for this is ForEach. For each file or folder, the displayed characteristics will be the date and time of the last update, name, length or file size. The strange thing that looks like strings of characters inside parentheses is the .NET string format code. They are used to left/right align fields and format dates, times, and numbers. Understanding these string format codes is not very important, because they are not essential to the nature of this script.

The If statement is where you determine whether any object is a directory or not. If the first character of the Mode attribute is 'd', the object is a directory. You need to double check because the code written for directories is often different from the code written for files.

Notice the $totalDirs++ line inside the If statement. This is the counter responsible for keeping track of directory numbers. Similarly, there is a $totalFiles variable that is used to track the total size of all files. These values ?? are always calculated during execution. But they are only displayed when the file listing process is finished.

```
ForEach ($i In $items) { $date = "{0, -20:MM/dd/yyyy hh:mm tt}" -f $i.LastWriteT
```

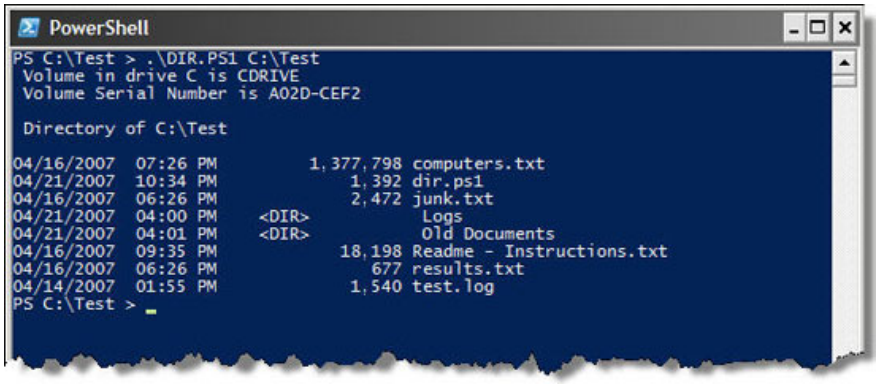


Figure 4: Displaying the output data of the updated script.

DIR.PS1: Footer

The only thing left is to write on the screen the total number of files, folders, total size of all files and free space on this drive partition. To do this you will need to use the counter variables ($totalFiles, $totalDirs, $totalSize) created in the previous section. You can know the amount of free space from the $volInfo variable created at the beginning of the script.

```
Write-Host ("{0, 16:N0}" -f $totalFiles + " File(s)" + ` "{0, 15:N0}" -f $totalS
```

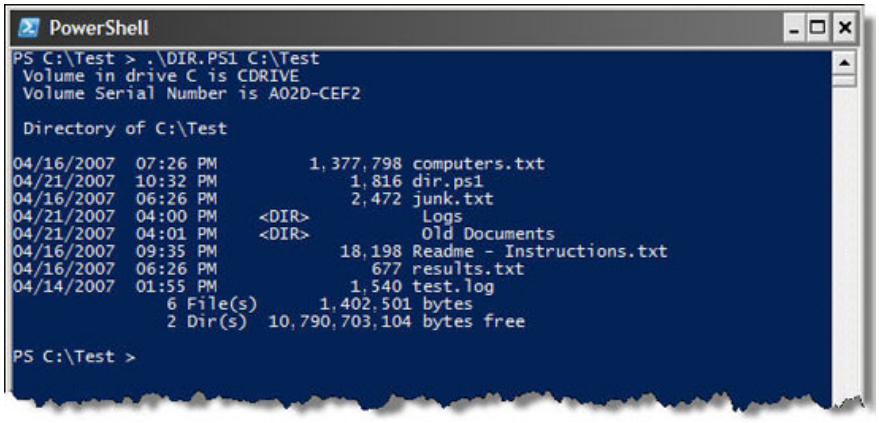


Figure 5: Complete display of the script's output data.

Enhanced forecasts and capabilities are possible

Although the script you create produces nearly identical output to the CMD.EXE DIR command, there are some predictions you need to be aware of and some enhancements that can be made.

1. This script does not perform any error checking.
2. If a valid path is not included in the script, the script will crash with a PowerShell error message.
3. The total number of folders given in the script is 2 less than the result from the CMD.EXE DIR command because the Get-ChildItem command does not count the two '.' folders. and '.' as in CMD.EXE.
4. Your script only sorts by file name and folder name and does not provide any other way to sort by attribute.
5. Your script is not capable of displaying folder contents and all subfolders.

Conclude

Although PowerShell is a powerful utility and scripting language, it only takes a little time to grasp and use it, especially if you are not familiar with the .NET Framework environment. I hope this article with its example script will be useful to anyone who wants to understand PowerShell. But the script created as an example in the article is quite simple. Believe that it can be built and developed more completely to serve more complex applications well.