Conversational **PowerShell**

A ConversationalGeek™ Book

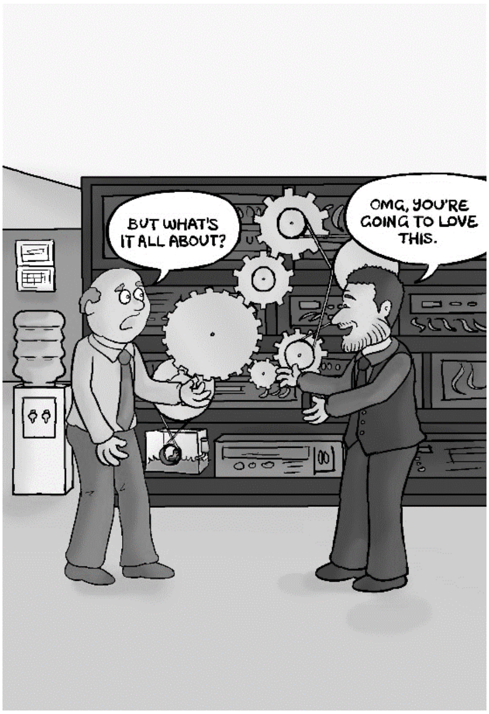
Learn about:

- **What is PowerShell and how do you learn it?**
- **What can PowerShell really do for me as an IT professional?**
- **From 0 to 60 learning PowerShell**

By Don Jones (Microsoft PowerShell MVP)

Conversational PowerShell

By Don Jones



ConversationalGeek™

Conversational PowerShell

Published by Conversational Geek Inc.

www.conversationalgeek.com

Trademarks

Warning and Disclaimer

Additional Information

Publisher Acknowledgments

All of the folks responsible for the creation of this guide:

Author:	Don Jones
Project Editor:	J. Peter Bruzzese
Copy Editor:	John Rugh
Content Reviewer(s):	Karla Reina

Note from the Author

I’ve been working with Windows PowerShell since… wow, since before it was released in 2006. Now I feel old. Thanks. Anyway, it’s been a really interesting ride, because the product was basically in stealth mode for the first couple of years.

As Microsoft continued to invest in it, and then build other things on top of it, it became really clear that *this was the future!* And of course, now it’s the present. Although it’s easy to not realize it, PowerShell probably represents one of the most impactful and fundamental technology shifts that Microsoft has made since moving from MS-DOS to Windows. Yeah, it’s that big of a deal. And if you’re not sure *why* it’s such a big deal – well, you’ve picked up the right book.

Don Jones

The “Conversational” Method

We have two objectives when we create a “Conversational” book: First, to make sure it’s written in a conversational tone so that it’s fun and easy to read. Second, to make sure you, the reader, can immediately take what you read and include it into your own conversations (personal or business-focused) with confidence.

These books are meant to increase your understanding of the subject. Terminology, conceptual ideas, trends in the market, and even fringe subject matter are brought together to ensure you can engage your customer, team, co-worker, friend and even the know-it-all Best Buy geek on a level playing field.

“Geek in the Mirror” Boxes

We infuse humor into our books through both cartoons and light banter from the author. When you see one of these boxes it’s the author stepping outside the dialog to speak directly to you. It might be an anecdote, it might be a personal experience or gut reaction and analysis, it might just be a sarcastic quip, but these “geek in the mirror” boxes are something you don’t want to skip.

In these boxes I can share just about anything on the subject at hand. Read ’em!

Who the Shell Cares?

It’s incredibly easy to look at PowerShell and just dismiss it as another tech-geek tool for getting stuff done. That’s largely because… well, that’s what it is. It’s just another tool for administering servers, client computers, and IT services. So… yeah. That’s it. Move on to the next book on your reading list.

Oh, wait, there’s actually a bit more to it. Think back to the days when people mainly got around in horse-drawn carriages. Those carriages were built by hand, and it could take weeks to put one together. When they were first invented, even automobiles were built by hand. Every one of them was a unique creation, with its own quirks and custom-fit pieces. Then Henry Ford came along with his mass-production factory lines and changed *everything.* It wasn't just a matter of producing cars faster: they could now be produced more *consistently.* Every one exactly alike. That reduced manufacturing expenses, since nobody had to deal with unique, "one-off" vehicles anymore. And lowering expenses meant lowering *prices,* too, meaning a ton more people could afford cars. Automation changed not only the automobile industry, but the entire world.

And that's why PowerShell matters. Yeah, it's just another tool for accomplishing the same things you've always accomplished. You're not achieving anything new, necessarily. You're still building a car, so to speak. But PowerShell shifts you over to a different *way* of achieving those things, and it's that shift that's important. PowerShell lets you accomplish things in less time – which lowers costs. It lets you be more consistent – which reduces errors. And PowerShell acts as a platform upon which you can build other, even more amazing things that further automation, consistency, and reduced costs.

So What *is* PowerShell?

PowerShell is a command-line interface for managing technology systems and services. Big words, and not an especially novel idea. In the 1980s, MS-DOS was also a command-line interface. Novell's NetWare operating system operated largely from a command-line interface. Unix and Linux have always been predominantly run from command-line interfaces.

So wait, Microsoft just stole this idea?

Totally. Why reinvent the wheel?

Command-line interface simply means that, instead of clicking graphical buttons and whatnot to get things done, you type commands into a big text box. Like in basically every hacker movie Hollywood ever made.

Command-line interfaces – or CLIs, if we can go truly geeky – have disadvantages and advantages compared to the graphical stuff Microsoft has traditionally used for administration. On the down side, a CLI doesn't just lay out all your options in front of you. There are no menus to poke around in to see what you can do, and no icons to help you figure out what's what. You have to know what the commands are, and you have to know what they do. Oh, and you have to be a pretty good typist. So there can be this huge learning curve with PowerShell, although it does a few things specifically to try and lower that curve a bit. We'll dig into those things in a moment.

On the up side, the things you type into a CLI are just text, right? I mean, if you were trying to complete some complex, multi-step task, you could theoretically plan it all out by just typing the commands into Notepad or something. Well, it turns out that's actually a thing. When you type a bunch of commands into a text file, that text file is called a *script.* And the CLI can run the script – essentially, it just opens the text file, reads it, and then starts running whatever commands are in there. This is a big deal, because it means you only have to hammer out the commands one time. The next time you need to accomplish that task, you just run the script. So although it might take you a while to get the script right the first time, from then on, you could run it with no effort. This is exactly what Ford did, right? I'm sure it took a long time to get the production line machinery all perfectly set up, but once he did, cars just flowed through the process nice and easy. Contrast that to clicking around in a graphical wizard or something – you can't really make that any faster. There's no practical way to tell the computer, "OK, right here, move the cursor to about an inch down and two inches across, and then click." So every time you do something in a graphical interface like that, you're burning the same amount of time, over and over.

So that's the promise and the price of PowerShell: being able to accomplish things faster and more consistently by using scripts, but having to learn all the commands to do so.

How's the Uptake Been?

Pretty good. Here's the thing you have to understand: a lot of guys and gals got into IT operations because Windows NT Server – as it was known at the time – was easy to use. You clicked a few icons, you checked a few checkboxes, and you were on your way. In fact, Microsoft probably owes much of its success to those guys and gals. The company made running a server so darn easy that anyone could do it. You didn't need some super highly trained network god who'd been through months of training on NetWare (which was dominant at the time). You just needed someone who was pretty good with their home Windows computer. So a *lot* of people got into the industry based on the promise of Windows being easy to learn and easy to use.

And then Microsoft rips it all away with this PowerShell thing. So there've been some hard feelings. But the thing is, Microsoft has moved beyond those one-off departmental file servers that provided the company's initial success. They are thinking cloud-scale now. Graphical administration is fine when you're running a server or two, but you can't run a thousand servers that way. So the company adapted (and that took a *long* time), and PowerShell is the answer now.

Don't underestimate the sour grapes, though. I've been literally yelled at about PowerShell, as if I'd personally planned the whole thing.

But Microsoft did a clever thing with PowerShell, because they *knew* the shift away from graphical administration would tick people off. Especially people working in smaller companies who might only *have* a handful of servers, and who might not actually *need* what PowerShell was offering. PowerShell can actually be run "behind the scenes," sort of, acting as the engine underneath a traditional graphical interface. So, you click your buttons, you check your checkboxes, and so on, but the computer invisibly runs PowerShell commands "under the hood" to make stuff happen. That way, administrators *can* still have a graphical administrative experience if that's what's right for them. Or, if they need the advantages of PowerShell, they can use it directly. Everyone wins.

The Big Takeaways

PowerShell is a command-line interface, where administrators type commands into a big text box to administer computers and services. It's got a bigger learning curve than a graphical user interface, but it enables a much better level of automation and consistency. Once you've invested in learning PowerShell, the payoff can be big in terms of time saved and errors avoided. And, more than ten years into its life, PowerShell's a pretty mature platform upon which Microsoft has been building lots of other amazing stuff.

Learning PowerShell

As I pointed out already, learning PowerShell definitely involves a bit of investment, because you have to know what commands to type. And simply memorizing all the commands isn’t an option. Windows Server 2012R2 ships with more than 3,000 commands. On top of that, nearly every Microsoft server product – Exchange, SharePoint, System Center, SQL Server, you name it – adds in a few hundred more. Then there are the non-Microsoft products that use PowerShell, like VMware vSphere, NetApp storage servers, and so on. Oh, and all the commands created by members of the community. So yeah, probably just under a million commands to learn.

PowerShell v1.0 shipped with about 150 commands, so it seemed feasible to learn ‘em all. 2.0 shot the count up past 500, and it’s been growing exponentially ever since.

So how do you learn this thing?!? Well, it turns out that PowerShell itself is willing to help you.

Knowing the Rules

PowerShell commands are all named using a pretty rigid syntax, which can make it easier to guess the name of the command that might do what you need. Each command's name consists of a verb (and the verbs come from a fixed list), a hyphen, and a noun. So, **Get-ADUser** would retrieve a list of users from Active Directory. **Set-Service** would modify the settings for a background service. **Stop-Process** would… well, you probably get the idea. Knowing these rules is the key to finding the commands you need.

Finding What You Need

The **Get-Command** command searches across all the commands *currently installed on your computer.* It accepts wildcards, so running something like **Get-Command *User*** would return a list of all commands that have "user" in their name. Or, run something like **Get-Command –Noun Service** to see what commands can work with services. Or **Get-Command –Verb Start** to see all the commands that use "Start" for the verb part of their name. With a little guesswork, it becomes pretty easy to find the command you need for a given task.

Asking for Help

Once you have the command you need, you're probably going to want to know how to make it do what you want. For that, just ask for help. Running **Get-Help Get-Service**, for example, will display extensive help options for the Get-Service command. Or, run **Get-Help Stop-Process –Full** to display help for Stop-Process, including examples on how to use the command.

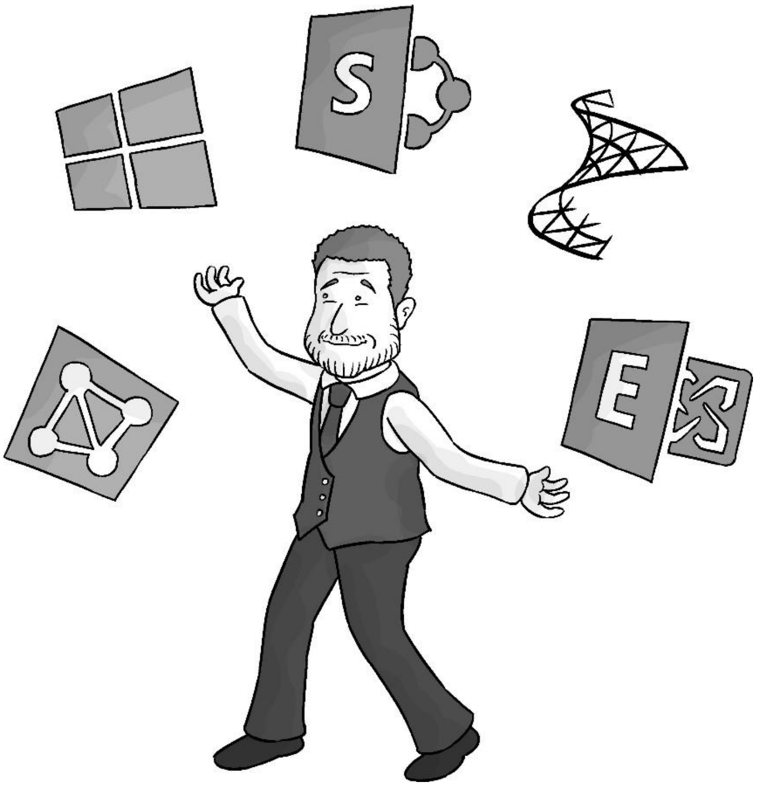
Since PowerShell v3.0, help doesn't actually come pre-installed. You need to open a shell as Administrator and run **Update-Help** to download the help content.

The Big Takeaways

So although PowerShell doesn't come with menus and icons to help you find your way, it *does* come with some easy tools and straightforward rules that help you find what you need. You're not consigned to a life of trying to find examples on Google, either, as the provided help files usually contain a lot of examples (I've seen as many as two dozen for one command) that help you actually see how a given command is meant to be used.

And yeah, it actually *is* that straightforward. I didn't say "easy," though. I mean, you do need to know a lot about the technology you're working with. For example, you wouldn't know to search for commands containing "user" if you didn't know that Active Directory has "user" objects that represent humans. In fact, I'd say the toughest part about learning PowerShell is that it tends to expose how little we sometimes know about the technologies that have been lurking beneath those graphical interfaces!

So, What Can PowerShell Do?



Anything.

Well, not really. Although, sort of. Okay, look, this is actually a really fun part of the PowerShell story.

First, PowerShell is built on Microsoft's .NET Framework, which essentially means that PowerShell can do anything the Framework can do. That's a *lot.* And if you're not comfortable using the Framework directly, that's fine, because the whole point of a PowerShell command is to be a friendly, well-documented wrapper around .NET stuff. When you run Get-Service, you're running .NET under the hood. But anything in

.NET is fair game, whether there's a "wrapper" command or not.

Second, PowerShell has strong connections to ~~the Force~~ Windows Management Instrumentation, or WMI. WMI contains a ton of management and administrative information and functionality, and PowerShell makes it easy to get to. Many PowerShell commands are actually "wrappers" around WMI stuff, making it all easier and more consistent.

Third, PowerShell can use older software modules written in Microsoft's Component Object Model (COM) and Distributed COM (DCOM). That lets PowerShell hook up to a lot of legacy technologies that might not have any .NET functionality.

Fourth and finally, PowerShell can run external command-line utilities and applications, like Netstat.exe, Ping.exe, Tracert.exe, and so on. So anything you've done under Windows' old Command Line world still works in almost exactly the same way.

And best of all, you can mix and match all of these technologies right from a single script. So you can really just take the best of every world, use whatever's available to get the job done, and munge (that's a technical term) it all together into a PowerShell script. It's pretty awesome.

PowerShell also supports embedded C# code, further expanding the crazy things you can do with it.

Focus on Commands

Although PowerShell’s ability to connect to all those acronyms (.NET, WMI, COM, and their friends) is a huge benefit, the shell really shines when you’re using its commands. .NET, as one example, is a really a friendly thing for developers, but it’s kinda hostile to the way administrators work. Admins aren’t usually programmers, after all, and .NET is all about programming. PowerShell commands can kind of translate that into something more admin-friendly. So Microsoft, other vendors, and even the general community tend to produce PowerShell commands to provide admins with a consistent, easier-to-use way of accessing all those underlying technologies.

By the by, it’s worth pointing out that in PowerShell-speak, *command* is a generic word that encompasses several distinct things:

- A *cmdlet* (pronounced “command-let”), which is written in .NET and designed to run natively inside PowerShell.
- A *function,* which is a PowerShell script that can look, act, smell, and taste like a cmdlet (tastes like chicken).
- An *application,* which is usually an external thing like Tracert.exe.
- A *script,* which is just a bunch of PowerShell commands strung together in a particular order.

Shopping for Commands

PowerShell bundles commands into *modules.* Most often, modules are related to a specific set of related tasks or technologies. There’s a module for managing Active Directory, a module for messing with shared folders, a module for SQL Server, and so on. Modules simply make it easier to load up an entire set of commands all at once, so you can use them.

The Windows operating system, depending on the version, comes with bunches of modules (especially from Windows 8 on, and from Windows Server 2012 on). Most Microsoft server products come with modules, too, and you usually obtain them by installing that product's administrative tools onto your computer. Other modules might be downloaded from the Internet.

There are a few commands you'll want to know with regard to modules:

- **Get-Module –ListAvailable** will show the modules installed on your computer. Well, if they're installed in the correct location. Hopefully they are.

- **Find-Module** (available in PowerShell v5 and later, or if you've installed PowerShell Package Manager on older versions) can search for modules in the online PowerShell Gallery. **Install-Module** can download and install modules from the Gallery.

Be a little careful. Microsoft might run PowerShell Gallery, but they don't vet or approve what people put into it. So, you know, proceed with due caution.

The Big Takeaways

So what can PowerShell do? A lot – and the list grows longer every day. The ability for developers to extend PowerShell by writing in C# or even C++, and for PowerShell to use existing WMI, COM, and external application functionality, means PowerShell has extensive reach.

You Mentioned Stuff Built Atop PowerShell?

Oh, yeah. Back in 2002, a guy at Microsoft named Jeffrey Snover wrote something called the "Monad Manifesto" (an annotated version is at https://www.penflip.com/powershellorg/monad-manifesto-annotated). It laid out what became a four-version vision for PowerShell, and pretty much the entire document has, at this point, come true. PowerShell started as the foundation for a lot of cool stuff. Not a lot of people really read the Manifesto back in the day (although "Monad" was the code-name for what became PowerShell), but it turns out that Snover had a really almost prophetic vision for what PowerShell needed to become. And, as the guy steering the ship for the Windows Management Framework product team (that's the team that produces PowerShell, amongst other goodies), he was able to make it happen.

Snover, who is now a Technical Fellow at Microsoft, is a hilarious dude. Check out snoverisms.com for some of his pithier pieces of wisdom.

Remoting

Version 2 of PowerShell introduced *Remoting,* a technology built on the open Web Services for Management (WS-MAN; we loves our acronyms!). It basically lets one computer send commands to bunches of other computers – in parallel! – and wait for them all to do whatever they've been told. It suddenly made it easier for administrators to manage dozens, hundreds, or thousands of computers as easily as managing one. And Remoting itself served as the foundation for…

Workflow

You can't build Rome in a day, and you can't always administer a computer in a single command. Sometimes, you need to do something that will require dozens or hundreds of commands, and some of them might take a long time to run. To make things more complicated, the computer might have to reboot in the middle of everything, or there might be the possibility of a network or power failure at some point. Ideally, you want the computer to pick up where it left off, right? Well, that's what Workflow is all about. Basically, you write a script, and the underlying Workflow engine makes sure it runs, even if that means being interrupted and picking back up later. And thanks to Remoting, you can push workflows out to multiple computers to run them in parallel! Workflow was born in PowerShell v3.

Workflow can actually get crazy-complicated. This shameless oversimplification has been brought to you by Don Jones and Conversational Geek.

Desired State Configuration

Introduced in PowerShell v4, Desired State Configuration (or… wait for it… DSC! More acronyms!) is kind of the penultimate pinnacle of PowerShell prosperity. DSC leverages all the investment Microsoft has been making in PowerShell since 2006, including the thousands of commands that have been written along the way.

Essentially, DSC lets you write a more-or-less-plain-English document that describes what a computer should look like once it's fully configured and operational. It doesn't need to contain code, or the actual instructions for configuring the computer. You just write down what the computer should be when it's all over. DSC then takes over and makes it happen. And, if you want it to, DSC will sit there like a mother hen, putting the computer's configuration back the way you want it whenever things change for some reason. It's awesome stuff.

The Big Takeaways

PowerShell sits as the bedrock of a lot of amazing, incredibly useful tools and technologies. Since its introduction in 2006, the product and its supporting technologies have continued to grow, and now form a major part of Microsoft’s strategy for business systems administration and automation. Yeah, all those folks back in 2006 who said, “oh, it’s just another VBScript, it’ll never catch on”, probably don’t have jobs anymore.

But WHAT'S IT LOOK LIKE???

```
Function PowerShell-Code
{
  [OutputType([string])]

  if($ClearHost -eq $false)
  {
      while($count -lt $S){
        if($count -eq ($S - 1)){
          Write-Host $S
        }
        else
        {
          Write-Host $T
        }
        sleep-host
        $count++
      }
```

Oh, right. Sorry, I just get so excited about what PowerShell can do that I… well, nevermind, let's just get into it. Suppose I want to find out what version of Windows my computer is running:

```
PS C:\> Get-CimInstance -ClassName Win32_OperatingSystem |
>>> Select-Object -Property Version,ServicePackMajorVersion

Version    ServicePackMajorVersion
-------    -----------------------
10.0.10240                       0
```

Not too difficult. And it's just as easy to find out from a remote computer:

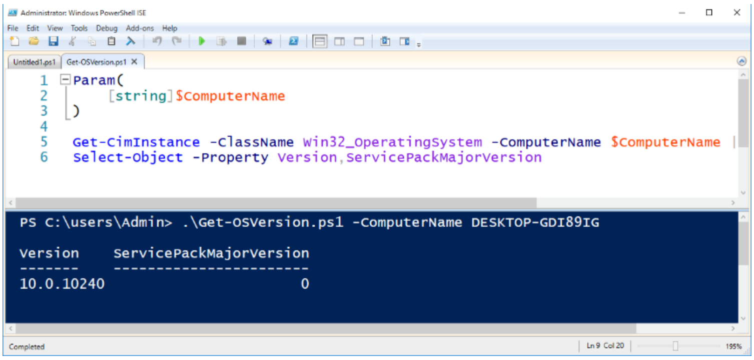
```
PS C:\> Get-CimInstance -ClassName Win32_OperatingSystem –
ComputerName CLIENT-B |
>>> Select-Object -Property Version,ServicePackMajorVersion

Version     ServicePackMajorVersion
-------     -----------------------
10.0.10240                        0
```

And I can even, without a lot of hassle, turn that into a standalone tool – a command, if you will – that other people can use. For example, I can create the following in a text file named Get-OSVersion.ps1:

```
1  Param(
2          [string]$ComputerName
3  )
4
5  Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName $ComputerName |
6  Select-Object -Property Version,ServicePackMajorVersion
```

In this script, I've added a **Param()** block, which lets me indicate I want someone to provide a computer name as input. You can see at the end of line 5 where that parameter "placeholder" (it's called a variable) is used. To run the script:


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1
Get-OSVersion.ps1
Completed
Ln 9 Col 20
195%

```
1  Param(
2          [string]$ComputerName
3  )
4
5  Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName $ComputerName |
6  Select-Object -Property Version,ServicePackMajorVersion
```

```
PS C:\users\Admin> .\Get-OSVersion.ps1 -ComputerName DESKTOP-GDI89IG

Version     ServicePackMajorVersion
-------     -----------------------
10.0.10240                        0
```

It's really that easy to get started. I mean, yeah, you've got to know what the commands are, but finding those can be part of the fun (as can a basic education on PowerShell). But how'd I come up with all that?

Finding the Commands

I obviously had a bit of an advantage here, since I've been doing this for a minute, but I almost always start with a **Get-Command** search. In this case, I'd read about the Win32_OperatingSystem thing and how it had something to do with something called CIM (acronyms!). I ran **Get-Command *Win32*** and it came up empty. See, even I get it wrong sometimes! So I ran **Get-Command *CIM*** and got several hits back. I read the help on a few of those until one of them, **Get-CimInstance**, seemed to make sense. I read the examples, and decided to play around with it.

Start in the Console

I always experiment in the PowerShell console window first. It's kind of the de facto way to use the shell. And I relied on a trick I haven't yet shared with you (here it comes): Tab. Like, the actual Tab key on the keyboard. Left-hand side. Go on, look for it. Turns out, PowerShell uses the Tab key really nicely. I typed **Get-CimI** and hit Tab… and the command name was completed. Then I typed a space, a hyphen, and hit Tab again to cycle through the parameter names until I found **–ClassName**. Space again, and then I typed **Win32_Op**, got bored of typing, and hit Tab to complete the class name. Then I hit Enter to see the results.

I basically kept fussing around until I got the output I wanted. The initial output had more than I wanted, so I used **Select-Object** (which I learned in my basic PowerShell education) to grab just the pieces of information I wanted. Then I went back (the Up Arrow key recalls what you typed previously) and added the **–ComputerName** parameter to query a remote computer. That worked, so I kept fussing around to get what I wanted.

This particular example requires that the remote computer have Remoting enabled, and that you have Administrator privileges to it. I made sure of those in advance.

Making a Tool

Once I got it working at the command line, I copied and pasted my working command into the PowerShell Integrated Scripting Environment (or ISE; ACRONYMS!). It's basically a script editor, and although it's fairly bare-bones, it gets the job done. Plus, it's built into Windows.

I immediately ran the command in the ISE, just to make sure it had copied and pasted correctly. I've messed that up in the past, believe it or not; it's worth taking the "baby steps" approach and testing.

Then, I replaced the hardcoded computer name with a parameter, added the **Param()** block, and saved the file so I could test it. You saw the results.

The $ComputerName variable could have been named anything; it didn't have to have that name just because it went with the –ComputerName parameter.

I start literally every tool and script I write in the same exact way. Even after 10 years of the shell stuff, I always start right in the console, running a single command until I get what I want. I figure out all my errors in the console, get everything all tweaked and perfect – and then move it into the script. When I’m ready to add to the script, it’s back to the console to test the next command. That way, I’m always sure of what the command is doing, and what I’m supposed to be typing.

The Big Takeaways

Although PowerShell has massive abilities, the shell itself is a pretty simple little guy. Using it isn’t necessarily complex, although it can initially be a challenge to find the commands you need. But once you start to learn a few commands, it tends to snowball until all of a sudden one day, you know hundreds of ‘em.

And PowerShell does make it really easy to take working commands and turn them into self-contained little tools – and *that,* my friend, is where the PowerShell investment pays off. After you figure out a command or two and put them into a script, you’ll never have to figure them out again.

Getting a Basic PowerShell Education

Which I suppose brings me to your next question: now that you're getting conversational in PowerShell, what's next?

Do me a favor: don't make Google your primary learning mechanism. I live near a quick-care clinic, and I see more people being treated for bloody foreheads after banging their heads on their desk after trying to learn PowerShell by Googling stuff. Just say no.

Microsoft's Virtual Academy (MVA! Acronyms!) has some great getting-started video tutorials starring my good friend Jason Helmick, and the inventor of PowerShell, Jeffrey Snover. Those are free, and they're a great place to start. My other good friend (I only have two), Jeffery Hicks, helped me write a book called “Learn Windows PowerShell in a Month of Lunches,” and I obviously think it’s a great place to start. Both the book and the MVA will help you wrap your head around some of PowerShell’s eccentricities, so you know what’s happening under the hood. Once you’ve digested that material, you’re cleared to start Googling again, because the stuff you find will start to make sense.

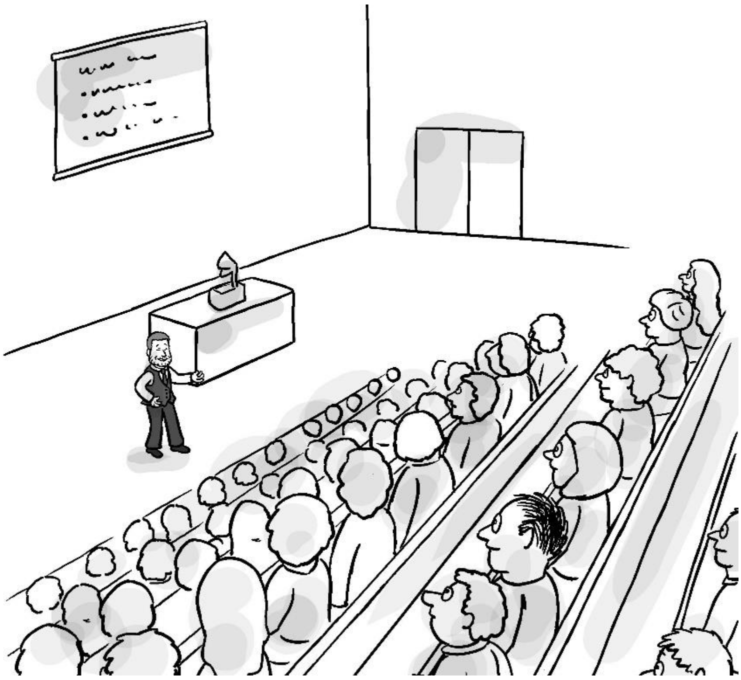
Some people sometimes post dumb, and wrong, stuff on the Internet, so be a skeptic.

So let’s say you get some basic education under your belt. What next?

Well, I’d say *get to work.* Pick some project at work, something that you find personally dull and repetitive, and automate it. But don’t pick the largest, most insane process you can find to start with! Bite off something smaller, first. Maybe write a little tool to unlock user accounts, or to reset user passwords (hint: **Get-Random** is fun for generating randomized new passwords).

As you tackle this first task, and probably every task after it, *you will run into problems.* It’s cool. Just take a deep breath and ask for help. There are Q&A forums on PowerShell.org, on places like ServerFault.com, and even on Microsoft’s own TechNet bulletin boards. And if someone gives you the answer you need, (A) thank them, and (B) make sure you understand *why the answer solves the problem.* If you don’t, ask again, because the “why” is the most important piece of the puzzle.

Get Yourself Involved!



Alright, we’re nearing the end of this thing, so I wanted to leave you on a really upbeat note. Here’s the most awesome news about PowerShell:

People. Love. It.

No kidding. By and large, Windows administrators tend to keep to themselves. They don’t typically form great big user groups, hang out in chat rooms discussing their technologies, or wear polo shirts with jokes that are only funny if you’re a geek. But with PowerShell, they *absolutely* do *all* of those things. And if you’re going to use PowerShell (and you totally should), then you need to know where to find these fellow Shellers, so that you can hang out with them and learn their jokes. And the secret hand sign (not kidding).

Start at PowerShell.org. It’s run by a nonprofit organization that I happen to head up, and it’s chock full of free ebooks, Q&A forums, webinars, you name it. We run an annual conference called PowerShell + DevOps Global Summit (powershellsummit.org), and we work with a lot of great vendors who make tools to help make PowerShell even more fun and powerful. We publish all kinds of articles, and the best part is that *everyone contributes.* Even total newcomers can kick in an article about how they conquered some problem in the shell, and those articles help everyone else who comes along later. Also, secret hand sign.

From there, you can start branching out. Did you know that a lot of Microsoft’s PowerShell code (especially for DSC) is in a public repository? Yup! Head to github.com/powershell and you’ll see bunches of open source code you can not only get, but you can also contribute to. Fix a bug, improve a comment, it’s all welcome.

Look for a local user group. There’s bunches of them, all over the world (we keep a list at PowerShell.org), and user groups are a great way to geek out, and often enjoy pizza, with like-minded PowerShell fans. But whatever you do, *get involved.* Nothing makes technology easier than having friends along to help you through the hard times and congratulate you on your achievements, and it feels *fantastic* to give something back when you can.

I hope to see you there!

NOTES

NOTES

Easily “converse” about PowerShell in any setting.

Look, PowerShell has been around since 2006, so you’re going to have to get around to learning it eventually. It’s at the heart of Microsoft’s direction for systems management, too, so learning it will just help make you a better IT person all-around. And there’s no better way to quickly become “conversational” in PowerShell than this book by PowerShell MVP Don Jones.

About Don Jones

Don Jones is a Windows PowerShell MVP, author of a half-dozen PowerShell books, and a co-founder and current President of PowerShell.org. This guy loves PowerShell. He’s also a Curriculum Director at online training firm Pluralsight, and a co-organizer of PowerShell + DevOps Global Summit.

ConversationalGeek™