DON JONES

POWERSHELL BY MISTAKE

PowerShell by Mistake

Don Jones

This book is for sale at http://leanpub.com/powershell-by-mistake

This version was published on 2018-12-18

Tweet This Book!

Please help Don Jones by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I’m learning #PowerShell by reviewing mistakes - you can too!

The suggested hashtag for this book is #PowerShellByMistake.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PowerShellByMistake

Also By Don Jones

Contents

About This Book

This is a **continually-published** book, “sold” only via Leanpub[1]. You’ll have access to new chapters each time they’re published, whenever that happens in the future. It’s recommended that you permit Leanpub to notify you, via e-mail, when new chapters are posted. That way, you’ll know it’s time to come get the latest. If you don’t explicitly opt into those notifications, be sure to check back at least monthly for new content.

There’s no change-log for this book; each new “build” published on Leanpub will contain a new chapter.

This is an open-source book[2]. That means you have the ability to pay $0 for it, and anything you do choose to pay will be donated to The DevOps Collective’s scholarship and other nonprofit programs.

[1]http://leanpub.com/powershell-by-mistake
[2]http://github.com/concentrateddon/powershell-by-mistake/

Introduction

Human brains are funny things. At the end of the day, our brains are still wired the same way they were when we were cavemen trying to survive in the wild. Anything relevant to our immediate survival tended to "stick" in our brains, and anything not relevant tends to slip past. Tell a child, "don't touch that hot pot on the stove!" all you want to, they're still likely to touch it - once.

This book is the hot pot on the stove.

In this book, I've collected edited versions of posts from the PowerShell.org Q&A forums and other online forums. I've anonymized these, although nothing in this book should *ever* be construed as criticism of the original posters. We *all* make mistakes, and thanks to the way our brains work, making mistakes and solving them is the best way for us to learn. So in each chapter, I'll present a new problem from the forums, and I'd like you to take the time to try and figure it out for yourself. I'll also present a solution, along with explanations.

This book is *not* a how-to, and I presume that you already have a decent amount of basic PowerShell education under your belt. This book is not about how to make Active Directory work, or Exchange Server, or SharePoint Server, or anything else; this is about *PowerShell*. So you're going to see problems that relate to the core of how PowerShell works. Many of these problems spring from "gotchas" in PowerShell, while others have their origin in very common misunderstandings about how the shell works. If you're looking for a structured learning approach to PowerShell, I offer recommendations on my website[3].

If *you* feel that you learn best by getting your hands dirty, making

[3]http://donjones.com/powershell

mistakes, and figuring them out, then this book is for you.

How to Use This Book

Read each chapter. Pore over the code (if you prefer to look at the code in an editor, which I recommend, you'll find the code downloadable from GitHub[4], with each chapter's code samples in their own folder). Read it all *thoroughly*. Try to figure out what's wrong. That's really the point of this book: taking the time to understand the code, and to see if you can figure out the "fix."

Each chapter ends in a "Spoiler!!" section, which contains a possible solution. Use this to check yourself, but know that most problems can be solved in a number of ways, so if you came up with something different, you're not necessarily wrong.

It is **hugely important** that you make the effort to understand the problem code and solve it *before* looking at the solution. This is a fundamental part of how human brains work! If you just skip ahead and read the solution, *you won't actually learn anything*. You'd be cheating yourself, and you shouldn't do that. Treat each chapter as a puzzle, and make the effort to try and solve each one.

[4]http://github.com/concentrateddon/powershell-by-mistake-code

A Note on Code Samples

Printing code in a book is a huge pain in the neck. E-readers format it however they darn well please, giving authors no control. PDF is okay, but consider this:

```
Import-CSV users.csv | Select @{n='samAccountName';e={$_.\
name}},Name,Department,City,Title | New-ADUser -PassThru \
| Export-CSV output.csv
```

That backslash you see at the end of the first line isn't a typo, it's the code-formatter's way of saying, "this didn't all fit on one line, so I'm wrapping it to the next line, but you shouldn't do that if you're typing this in yourself." I can't fix that because there's no other reasonable thing to do.

Backslashes cause another problem, as in C:Program FilesWindowsPowerShell:

```
C:\Program Files\WindowsPowerShell
```

See, in a lot of instances the backslashes get removed, because Markdown considers them to be escape characters. If you notice this happening, consider visiting this book's GitHub repo and submitting a change ;).

As you can see, reading code in a book can be a pain. Instead, I recommend grabbing the code downloads from GitHub[5] so you can see the code in its "unadulterated" form. Or, just learn to deal with the foibles of printed code listings!

[5] http://github.com/concentrateddon/powershell-by-mistake-code

Contacting Me

I cannot help with technical Q&A via email, so please don't use the "Email the Author" form on this book's Leanpub page for that, and please don't tweet questions to me @concentratedDon. I do love hearing from people via those channels, but they're not suitable for technical Q&A. Instead, please use the Q&A forums on PowerShell.org[6] for technical Q&A.

That said, if you found a typo or something, please consider logging an issue on this book's GitHub repo[7]. I do appreciate it, and I'll make corrections for the next build of the book. And know what I appreciate even more? Just fork the repo, make the fix yourself, and submit a Pull Request! As a "continually published, Agile book," there's not really an opportunity for an editor to step in and fix those things ahead of time (and financially, this book wouldn't be worth doing if an editor needed to be paid), so it's really awesome of you to let me know if you find things. I also appreciate your patience with my typos!

[6]http://powershell.org/forums

[7]http://github.com/concentrateddon/powershell-by-mistake/

Problem 1: On Apples and Apples

Consider the following code:

```
$jsonFiles = Get-ChildItem -path "c:\tmp\json" -filter *.\
json |
  get-content -raw
$allobjects = $jsonFiles | convertfrom-json

$alreadyCreatedGroup = Get-UnifiedGroup | select alias

function checkForExistingGroup {
    [CmdletBinding()]
    Param(
        [Parameter(Position=0, Mandatory=$true, ValueFrom\
Pipeline=$true)]
        $InputObject
    )

    Process {
            if("gr-$($InputObject.alias)" -in $alreadyCre\
atedGroup){
                echo "$($inputobject.alias) exists"
                }
                    else{
                        echo "$($InputObject.alias) does \
not exist"
                    }
            }
        }
```

```

$allobjects | checkForExistingGroup
```

Basically, this code is grabbing some group aliases in `$alreadyCreatedGroup`. It's then grabbing a bunch of objects which have an Alias property, piping them to a function, and then checking to see which of those already exist in `$alreadyCreatedGroup`. A trick is that the piped-in objects have a "gr-" prefix, whereas the items in `$alreadyCreatedGroup` do not. As written, this will not work. Can you see why?

Spoiler!!

There are a few problems with the script, and I'll start with the non-substantive ones.

First, the function is using a name that doesn't meet PowerShell naming conventions.

Second, the function is using `echo`, which is an alias to `Write-Host`; it really should be outputting this "test output" using `Write-Verbose`, which can be disabled when it's no longer needed, and re-enabled when it is, simply by running (or not running) the function using `-Verbose`.

Substantively, the function itself is using `$alreadyCreatedGroup`, which *exists outside the scope of the function*. This is a very bad coding practice, and can create a variety of difficult-to-debug situations. Except in extremely rare situations, functions should accept input *only* via parameters.

Finally, all those issues aside, the main problem is here:

```
$alreadyCreatedGroup = Get-UnifiedGroup | select alias
```

This creates a collection of objects which have an Alias property. Compare that to this:

```
$inputobject.alias
```

Presuming the piped-in objects have an Alias property also, this code will return a simple string. So when the comparison is run:

```
("gr-$($InputObject.alias)" -in $alreadyCreatedGroup)
```

We're checking to see if a *simple string*, prefixed with "gr-", exists within a collection of objects. PowerShell doesn't "know" that it's supposed to be comparing the prefixed string *to the Alias property* of the objects in `$alreadyCreatedGroup`. Instead, PowerShell is going to "think" of this like, "okay, is the string 'gr-something' the same as this complex object that potentially has multiple properties?" It's not like comparing apples to oranges, it's like comparing apples to an entire produce section at the grocery store.

This is a *very common "gotcha"* in PowerShell. Whenever you're comparing things, you have to make absolutely certain you're comparing the same *kinds* of things. Here, we've got a simple string coming from `$InputObject.alias`. The fact that `$alreadyCreatedGroup` only possess a property named Alias may be meaningful to *humans*, but PowerShell can't make the "mental leap" and think, "OOOOOH, I'm supposed to compare the string to the contents of the only property that happens to exist, here." So the comparison fails.

Revised:

```
$alreadyCreatedGroup = Get-UnifiedGroup | select -Expand \
alias

function Test-ForExistingGroup {
    [CmdletBinding()]
    Param(
        [Parameter(Position=0, Mandatory=$true, ValueFrom\
Pipeline=$true)]
        $InputObject,

        [Parameter(Position=0, Mandatory=$true)]
        $ExistingGroups
    )

    Process {
            if("gr-$($InputObject.alias)" -in $ExistingGr\
oups) {
                Write-Verbose "$($inputobject.alias) exis\
ts"
            } else{
                Write-Verbose "$($InputObject.alias) does\
 not exist"
            }
    }
}

$allobjects | Test-ForExistingGroup -Verbose -Existing $a\
lreadyCreatedGroup
```

1. I've renamed the function to conform to PowerShell patterns.
2. I've fixed some of the bracket indentation and alignment. This is a pet peeve.
3. I've added a parameter so that the `$alreadyCreatedGroup` can be passed into the function properly. The function no longer needs to rely on external data.

4. I’ve modified the `echo` commands to use `Write-Verbose` instead. Note that I add `-Verbose` when running the function, at the end.
5. Importantly, I’m extracting the *contents* of the Alias property for `$alreadyCreatedGroup`, so that I’m passing the function a list of strings to compare to other stringsâ€”apples to apples. That was done here:

```
$alreadyCreatedGroup = Get-UnifiedGroup | select -Expand \
alias
```

The `-ExpandProperty` parameter of `Select-Object` *extracts* the string from the Alias property, leaving an array of strings rather than a collection of objects.

Problem 2: On Alternate Credentials

Here’s some code:

```
$Password = ConvertTo-SecureString "abcd" -AsPlainText -F\
orce
$Credentials = New-Object System.Management.Automation.PS\
Credential ("domainname\aduser", $Password )
$sql1 = New-PSSession -ComputerName sql1 -Credential $Cre\
dentials
Import-PSSession -Session sql1 -Module SQLPS -Prefix sql1\
 -AllowClobber
```

As you can see, the intent is to create a Remoting session to a computer (which appears to be running SQL Server), and use implicit remoting to import the SQLPS module from the server. The trick is, we’re assuming that the user running this doesn’t have permission to connect to the remote machine.

Specifically, the business requirement is:

> Domain users will be running the script but they don’t have access to the server itself so the New-PSSession cmdlet blows up with ‘access denied’.

To confirm that Remoting was indeed working, the above code was run. Obviously *the code works*, but it’s a bad idea. So what do you think of this, and how would you fix it?

Spoiler!!

Hardcoding credentials is always a bad idea. Always. And to be fair, the individual who originally posted this acknowledged very clearly that they knew it was a bad idea. They just didn't know a better way.

Did you?

The answer is Just Enough Administration, or JEA. It's really just a set of commands (available in PowerShell Gallery, by the by) that help define and create *constrained endpoints* in PowerShell Remoting. JEA is a big deal: it's a huge part of Azure Stack, for example, and it's a robust-production ready tool that solves a specific problem, which is how to let users safely run commands that they don't normally have permission to run.

When you set up Remoting on a computer, you're creating up to four *endpoints*. Nobody notices this, because normally when we use Remoting we're accessing the main default endpoint, which is named "Microsoft.PowerShell." But look at the help for commands like `Invoke-Command` and `Enter-PSSession` and you'll see that you can indeed specify the name of any endpoint you want. The default endpoint only allows Admins in (and members of a special Remote Admins group), and once you're in, you can run whatever commands you want. The default endpoint doesn't have a "run as" account assigned, so it just runs commands by using your credentials.

But that's just the default behavior.

When you create a custom endpoint, which is what JEA does, you can specify:

- Who is allowed to access it
- What commands they are allowed to run
- A "run as" account, which is stored *on the server* and used to execute everything run inside the endpoint

The “what commands they are allowed to run” is a huge deal. You can pre-import whatever modules you want, and provide a “whitelist” of permitted commands. Only those commands will be visible, and they’ll run as the “run as” account. You can even create *proxy functions*, which basically take an existing command and do stuff like remove parameters, add parameters, and so on. You could, for example, hardcode certain parameter values of a command rather than exposing them inside the endpoint. This restricts what someone can do with the command. It’s exactly how Office 365’s Remoting works, so that Microsoft can prevent you from accessing bits of Exchange (for example) that belong to another customer.

Constrained endpointsâ€”and JEA, which makes them easier to manageâ€”are a massively useful and important PowerShell feature that’s existed since v2. It’s a shame so few people know about it, and what it can do.

Problem 3: On Strings ByValue

As a note, the code for this one is pretty concise and illustrative, so there's no downloadable code version in the GitHub repo.

Consider this:

```
PS C:\> $MAC="08-00-27-35-AE-2C"
PS C:\> $MAC.replace('-','%')
08%00%27%35%AE%2C
```

Clear enough, right? The "-" characters are replaced with "%." But continuing in the same shell session, why does the following happen?

```
PS C:\> $MAC.replace('%',':')
08-00-27-35-AE-2C
```

That's your first problem. Second, consider this:

```
PS C:\> $MAC = Get-WMIObject win32_networkadapterconfigur\
ation | foreach { $_.MacAddress }
PS C:\> $MAC
08:00:27:35:AE:2C
PS C:\> $MAC.replace(':','-')
You cannot call a method on a null-valued expression.
At line:1 char:1
+ $MY_MAC.replace(':','-')
```

We've clearly gotten a MAC address in `$MAC`, so why the error message?

Spoiler!!

Let's cover the first one first.

```
PS C:\> $MAC="08-00-27-35-AE-2C"
PS C:\> $MAC.replace('-','%')
08%00%27%35%AE%2C
PS C:\> $MAC.replace('%',':')
08-00-27-35-AE-2C
```

In .NET Framework, and therefore PowerShell (and in most languages, for that matter), variables refer to things *by value* as opposed to *by reference*. That means when you run a method like `Replace()`, you aren't modifying the contents of the original variable. Instead, you produce a *new value*, and the variable continues to contain its *old value*. In the above example, we didn't capture the new value into a variable, so it displayed at the console and then basically vanished into the ether.

By reference works a bit differently, and it's a bit unusual to see it used in a situation like the above (PowerShell doesn't do "ByRef" very easily, either). In a "ByRef" system, using a variable actually just points to the location in memory where the variable "lives," so any actions taken "against" the variable actually do modify its contents. I'm mentioning this because it *is* a legit thing in computers, but it's unusual in PowerShell.

Next problem:

```
PS C:\> $MAC = Get-WMIObject win32_networkadapterconfigur\
ation | foreach { $_.MacAddress }
PS C:\> $MAC
08:00:27:35:AE:2C
PS C:\> $MAC.replace(':','-')
You cannot call a method on a null-valued expression.
At line:1 char:1
+ $MY_MAC.replace(':','-')
```

This one's actually really subtle, and it illustrates how PowerShell can mess with your head without you realizing it. In this case, the `Get-WmiObject` call function as you might think, and the return objects do in fact have a `MacAddress` property. And that `MacAddress` property does return a string value, which does have a `Replace()` method. So what's happening?

It turns out that the `MacAddress` property is *actually an array*, not a single value. When we just ran `$MAC`, PowerShell said, "well, I suppose I could just display the fact that this is an array, but I bet what this person wants is to see all the items in the array. So I'll just display them all. Oh, there's only one."

You see, an array of one object *is still an array*, and *the array itself* does not have a `Replace()` method. If we'd have done this:

```
PS C:\> $MAC[0].replace(':','-')
```

It would have worked, because now we're referring to a specific object within the array, which is a string, which does have a `Replace()` method.

PowerShell can actually be a little irritating about arrays in scenarios like this, because in *some* cases and in *some* later versions of PowerShell, it'll secretly enumerate each item in the collection. Basically, it'll do this under the hood:

```
ForEach ($item in $MAC) {
  $item.Replace(':','-')
  }
```

But it can't do that little trick every time (or in older versions), and so you *might not realize* you're dealing with an array. Further, `Get-Member` will, by default, not show you that you've got an array–it figures you're probably interested in what's *inside* the array, creating even more confusion.

One trick:

```
$MAC.Count
```

That'll return a nonzero number if you've got an array (to be fair, it can also return zero if you're referring to an empty array, but that's rare in the specific scenario outlined here), and you'll get an error if you're not dealing with an array (unless you happen to find a rare object that has a `Count` property).

Problem 4: On Magic Quote Timing

The following is a snippet; a more complete code example is available in the GitHub repo described in the front matter to this book. The stated problem with this code is that, while it basically works fine (well, the complete version does), the email alerts never contain the error message desired.

```
Import-Module ActiveDirectory

$Smtpserver = "smtpmail.domain.com"
$From = "noreply@domain.com"
$To = "user@domain.com"
$Subject = "Delete Stale Computers Failed"
$Body = "The Delete Stale computers script failed due to \
error '$($errormessage)'.   One of the OU's listed maybe \
missing in Active Directory.  Please review the list of O\
U's and check if they are still present in Active Directo\
ry"

$date = [DateTime]::Today.AddDays(-180)

$ous = @('OU=Name,DC=domain,DC=com')


Get-AdComputer -Properties LastLogonDate -Filter {LastLog\
onDate -le $date} -Searchbase $_


if ($error[0].exception.message.Contains("Directory objec\
t not found")) {
            $errormessage = $error[0].exception.message.T\
oLower()
            Send-MailMessage -SmtpServer $Smtpserver -Fro\
m $From -To $To -Subject $Subject -BodyAsHtml $Body
            $error.Clear()
        }
```

You can see where the error message variable is included in the email body:

```
$Body = "The Delete Stale computers script failed due to \
error '$($errormessage)'.   One of the OU's listed maybe \
missing in Active Directory.  Please review the list of O\
U's and check if they are still present in Active Directo\
ry"
```

And you can see where $errormessage is being populated:

```
$errormessage = $error[0].exception.message.ToLower()
```

So that's the problem?

Spoiler!!

PowerShell uses double quotation marks as "magic quotes." In programming languages that support magic quotes, the language scans for variables or subexpressions, and replaces them with whatever they evaluate to:

```
PS C:\> $a = 'Hello'
PS C:\> $b = "$a, World'
PS C:\> $b
Hello, World
```

There are two ways that languages implement magic quotes, which I personally call “eager” and “lazy.” In an “eager” language, *the contents of the string are evaluated and replaced when you define the string*. In a “lazy” language, this happens when the string is *retrieved.*

PowerShell is an “eager” language. This means that it tried to insert `$errormessage` waaaay at the top of the script, rather than after `$errormessage` had been populated. The fix here is to move the email body definition:

```
if ($error[0].exception.message.Contains("Directory objec\
t not found")) {
            $errormessage = $error[0].exception.message.T\
oLower()
            $Body = "The Delete Stale computers script fa\
iled due to error '$($errormessage)'.   One of the OU's l\
isted maybe missing in Active Directory.  Please review t\
he list of OU's and check if they are still present in Ac\
tive Directory"
            Send-MailMessage -SmtpServer $Smtpserver -Fro\
m $From -To $To -Subject $Subject -BodyAsHtml $Body
            $error.Clear()
        }
```

This can seem inefficient, because you’re constantly re-defining the string, but that’s the way you *have* to do it in PowerShell. I mean, there are some other syntactic ways you could get to the same place of “I have to create a new string every time,” but you’ll still end up in that place.

Problem 5: On Patterns and Suppression

Consider this code, which doesn't produce the intended output. I'll give you an up-front thing to look for: there's both a technical problem here, as well as some style problems.

```
$ErrorActionPreference = 'SilentlyContinue'

$ComputerName =Get-ADComputer -Filter {(Name -like "*")} \
-SearchBase "OU=AsiaPacific,OU=Sales,OU=UserAccounts,DC=F\
ABRIKAM,DC=COM" | Select-Object -ExpandProperty Name

$results = @()

ForEach ($computer in $ComputerName) {

$Results += Get-NetAdapter -CimSession $ComputerName | Se\
lect-Object PsComputerName, InterfaceAlias, Status, MacAd\
dress

}

$results | Export-csv -path C\users\bret.hooker\desktop\m\
acaddress.csv -Append
```

What would you change?

Spoiler!!

First off, let's address the style problems. I personally have a *huge* problem with:

```
$ErrorActionPreference = 'SilentlyContinue'
```

Sitting along at the top of a script, this simply disables error reporting, and it's the main reason this script becomes harder to debug. PowerShell *is* returning a useful error message for the main technical problem in this script, but that message is being ruthlessly suppressed. This is not a good way to gracefully handle errors; a good read of *The Big Book of PowerShell Error Handling*[8], which is also available in a Spanish translation, concisely outlines the right way to handle anticipated errors without removing useful ones.

Next, let's look at the technical problem:

```
$Results += Get-NetAdapter -CimSession $ComputerName | Se\
lect-Object PsComputerName, InterfaceAlias, Status, MacAd\
dress
```

The problem here is that `$ComputerName`, if you follow the logic of the script, contains a computer name. That isn't what `-CimSession` needs, though. Many CIM-based cmdlets don't come with a `-ComputerName` parameter; instead, you're expected to use `New-CimSession` to spin up a new session, and then query against it. This line of code also reveals a major style problem that goes against a core PowerShell coding pattern: accumulating output in an array. The Big Book of PowerShell Gotchas[9] outlines why and suggests a more PowerShell-native approach. There are two main reasons: appending to arrays is memory-intensive, as the array has to be recreated each time, and

[8]https://leanpub.com/thebigbookofpowershellerrorhandling
[9]https://leanpub.com/thebigbookofpowershellgotchas

because this approach mis-uses the PowerShell pipeline. It's *very* common to see people familiar with other programming languages do this, because they're jumping in without really getting what the pipeline is all about. That's fine! It's a learning experience!

Finally, there's a kinda-technical problem here:

```
-Filter {(Name -like "*")}
```

This is a little overwrought, and it's going to take longer for the domain controller to process. I'd just use `-Filter *` instead. Neither the original script, nor my revision, seeks to address situations where the number of queried objects exceeds the domain controller's query limit.

I'd rewrite this to look something like this:

```
Function Get-NetAdapterInfo {
 [CmdletBinding()]
 Param(
  [Parameter(Mandatory=$True)]
  [string]$SearchBase
 )

 Get-ADComputer -Filter * -SearchBase $SearchBase |
 Select -Expand Name |
 ForEach-Object {
  $session = New-CimSession -ComputerName $_
  Get-NetAdapter -CimSession $session |
  Select-Object PsComputerName, InterfaceAlias, Status, M\
acAddress
 }

}

Get-NetAdapterInfo -searchbase "ou=whatever,dc=domain,dc=\
com" | Export-CSV Whatever.csv
```

I've rewritten this as a proper advanced function, which is how pretty much *everything* in PowerShell should be done. That separates the script's functionality, which is querying network adapter info, from the destination, which in this case is a CSV. By outputting objects, one at a time, to the pipeline, the function becomes self-contained and its output can be sent anywhere.

I've made the function parameterized, so that it can be self-contained while accepting new `-SearchBase` data each time it's run.

I've also corrected the problem with `-CimSession`, and removed the error suppression. I did *not* add proper error handling, which is certainly something I might want to do when spinning up the new CIM session. I'd likely do that in the next revision of the function.

Problem 6: On Formatting Numbers in Strings

Here's a snippet of code. This one, I'm not including in the GitHub repo, because there's really only a couple of lines to worry about. For context, here's the large chunk:

```
$SourceFile = "C:/Temp/File.txt"
$DestinationFile = "C:/Temp/NonexistentDirectory/File.txt"

If (Test-Path $DestinationFile) {
$i = 0
While (Test-Path $DestinationFile) {
$i += 1
$DestinationFile = "C:/Temp/NonexistentDirectory/File$i.t\
xt"
}
} Else {
New-Item -ItemType File -Path $DestinationFile -Force
}

Copy-Item -Path $SourceFile -Destination $DestinationFile\
 -Force
```

The *desire* is for the destination files to be named `File00001.txt` and so on, but obviously as-is it's producing `File1.txt`. Here's the couple of relevant lines:

```
$i += 1
$DestinationFile = "C:/Temp/NonexistentDirectory/File$i.t\
xt"
```

How would you achieve the intended outcome?

Spoiler!!

This is a place where PowerShell's oft-ignored and mostly under-used `-f` formatting operator comes into play.

```
"{0} there {1}" -f "Hello",$name
```

Like most operators, `-f` takes two *operands*. The first is a string, which can contain {0} placeholders. The second is an array of values, which are then mapped into those placeholders. Placeholder numbering starts at zero.

But those placeholders can do a lot more, because you can "tag" them with formatting instructions. This lets them format numbers and dates, particularly, in a huge variety of ways. There's a great article[10] with some very useful and specific examples, but I find myself consulting the .NET string formatting documentation[11], because although it's very developer-y, it's also very complete and it's what the `-f` operator is using under the hood.

In this case, we want to take a number and left-pad it with a certain number of zeroes.

```
"{0:d5}" -f 1
```

This says I want a 5-digit number, and PowerShell will left-pad with zeroes as needed to hit that quantity. So:

[10]https://social.technet.microsoft.com/wiki/contents/articles/7855.powershell-using-the-f-format-operator.aspx
[11]https://msdn.microsoft.com/en-us/library/system.string.format(v=vs.110).aspx

```
$DestinationFile =
 "C:/Temp/NonexistentDirectory/File{0:d5}.txt" -f $i
```

This would produce filenames like `File00001.txt`, `File00010.txt`, and so on.

Problem 7: On Parsing Strings

This one's less of a "broken script" and more of a problem that tests your knowledge of PowerShell toolmaking patterns. Suppose you have a data file that consists of multiple lines. Each line looks like this:

```
NM1*PR*2*MEDICARE COMPLETE*****PI*map1stString
```

There's no header, but each * denotes a field. You'll notice that some of the fields are empty; that isn't the case on every single line, but it's something you need to consider. Your immediate need is to read these in, and where the fourth field reads "MEDICARE COMPLETE," you need to change it to "MEDICARE ALL," and then re-write all the lines back out to the data file.

How would you do this?

Spoiler!!

A lot of folks tackle this as a string-parsing problem, and they're not wrong to do so. However, what I see a lot of people do is spend a lot of time parsing out the data they want. For example:

```
ForEach ($line in (Get-Content file.txt)) {
 If ($line -like '*MEDICARE COMPLETE*') {
  Write $line -replace 'MEDICARE COMPLETE','MEDICARE ALL'
 } else {
  Write $line
 }
}
```

Or something along those lines. This *kind of* misses the point of PowerShell, which is to *act as a wrapper around other, hard-to-do stuff.* PowerShell takes stuff which isn't structured, like text, and turns it into structured objects. I would start by writing myself a set of `Import-` and `Export-` commands which respectively read-in that text and created objects, and wrote-out those objects into the desired string state. In *this* case, the hard work's already been done:

```
Function Import-MyDataFile {
 [CmdletBinding()]
 Param(
  [Parameter(Mandatory=$True)]
  [string]$FilePath
 )
 $headers = @('Code','Type','Value',
  'Name','Amount','Extended','Allowed','Denied',
  'System','Note')
 Import-CSV $FilePath -Delim "*" -Header $headers
}

Function Export-MyDataFile {
 [CmdletBinding()]
 Param(
  [Parameter(Mandatory=$True)]
  [string]$FilePath,
  [Parameter(ValueFromPipeline=$True)]
  [string[]]$InputObject
 )
 BEGIN {
  $data = @()
 }
 PROCESS {
  $data += $InputObject
 }
 END {
  $InputObject | Export-CSV $FilePath Delim '*' -NoHead
 }
}
```

See, the `-CSV` commands can already handle any kind of delimited data, so I've just "wrapped" them into a version specific to my data, which uses * as delimiters. I've also specified headers (which I made up for this example, but presumably in a production environonement someone could tell me what each field meant), so I can have a convenient way of referring to my data. Having done that:

```
Import-MyDataFile input.txt |
ForEach {
 $_.Name = $_.Name -Replace `
 'MEDICARE COMPLETE','MEDICARE ALL'
} |
Export-MyDataFile output.txt
```

Or something broadly *like* all that. The syntax here isn't my point so much as the *pattern*. If you're going to engage in text-parsing, then always do so with the goal of producing *objects*, rather than directly manipulating text. Objects become easier to work with since PowerShell is geared around objects (and is very much not geared around text). Having done this work to parse strings into objects and back again, future tasks working with this same data structure will be vastly faster and simpler, meaning I get a big return on my investment.

All this applies, in my mind, *anytime* I'm looking at text files. For example, I see people *killing themselves* parsing log files, trying to find a line that contains some specific text, which is often partly variable, so they end up writing tortuous regular expressions. I'd rather front-load all that work, parse the entire file into *objects*, and then deal with the objects. PowerShell *loves* structured data (objects) and is much better at working with them than it is dealing with huge wedges of text.

Bear in mind that `ConvertFrom-String` exists in newer versions of Windows PowerShell, and is very powerful at turning unstructured text into structured data with a lot less work on your part. Use what's in the shell to reduce your workload!

Problem 8: Mix 'n' Match

This one's one of those "gotchas" that you either have run into and will spot immediately, or that can be extremely vexing. It catches newcomers all the time. This is a code snippet:

```
Get-ADUser : Parameter set cannot be resolved using the s\
pecified named parameters.
At C:\Work\test.ps1:10 char:34
+ Get-ADUser -Filter {Company -like "*Paul*"} -Identity $\
_.s ...
+
    + CategoryInfo          : InvalidArgument: (:) [Get-A\
DUser], ParameterBindingException
    + FullyQualifiedErrorId : AmbiguousParameterSet,Micro\
soft.ActiveDirectory.Management.Commands.GetADUser

Get-ADUser -Filter {Company -like "*Paul*"} -Identity $_.\
sAMAccountName -Properties DisplayName, LastlogonDate, En\
abled, AccountLockoutTime, LastBadPasswordAttempt, BadPwd\
Count, LockedOut, Company, Description
```

Do you see the problem?

Spoiler!!

This one not only highlights the importance of the help files in PowerShell, but the importance of knowing all the little clues they give you. Go ahead and read the documentation for Get-ADUser[12].

[12]https://docs.microsoft.com/en-us/powershell/module/addsadministration/get-aduser

The only docs actually make this a bit more clear, visually, than then in-product help files.

Notice that the first block of syntax includes a `-Filter` parameter, but no `-Identity` parameter. The second block has `-Identity`, but not `-Filter`. That means those two parameters are *mutually exclusive*; they exist in different *parameter sets*, and you cannot mix and match between parameter sets in PowerShell. You must choose either of them (or one of the other sets), but you can’t use both.

There’s also a kind of semantic problem in the original code snippet. It’s attempting to select a given user by their SAM Account Name (which is unique in all of a single directory), and then filter for just those users whose Company field contains “Paul.” That’s nonsensical; PowerShell parameters are nearly always *additive* not *alternative*, you couldn’t have more than one user with a given SAM Account Name, and so there’s no sense to “also” filter the results by Company. The `-Identity` parameter always returns zero or one user; that’s why it cannot be used alongside `-Filter`, which can return more than one.

If the intent here was to get all users matching the SAM Account name *or* having a given string in the Company field, you’d likely need to run two queries and aggregate the output of them. Alternately, if the intent was to get all the users having a specific Company field, and then narrow the results down, you’d probably perform that Company query first, and then pipe the results to `Where-Object` to filter them further. You can get the domain controller to do more complex filters using an `-LDAPFilter`, but that uses its own specialized syntax that isn’t always widely understood by everyone.

Problem 9: The Return ofâ€¦

This is another common gotcha in PowerShell. But, rather than tell you what the problem is with the code, I want you to figure it out for yourself. You'll need to predict what the problem will be, as in what someone might complain about if they wrote this themselves and then ran it. And then, create a working version. This is a tiny snippet, so there won't be a version in the download repository.

```
Function Get-ServiceStatus {
 [CmdletBinding()]
 Param(
  [Parameter(Mandatory=$True)]
  [string[]]$Name
 )
 ForEach ($item in $name) {
  $svc = Get-Service $item
  Return $svc.status
 }
}
```

Spolier!!

The `return` keyword is a tricky one in PowerShell. It's especially confusing for programmers from another language, who think they know what it does based on their past experience. Honestly, `return` is something that many on the PowerShell product team regret, for the confusion it's caused.

In PowerShell v5 classes, `return` does what it should do based on how other languages work: if PowerShell sees you using `return` in a class, it suppresses other things that are output to the pipeline. Instead, it only outputs whatever `return` tells it to, and then it exits the class method. That's how `return` "should" work.

But anywhere else in PowerShell, as in the function in this chapter, `return` simply emits one last thing to the pipeline and then exits. Anything you've *previously* output to the pipeline *still gets output*, which can be confusing. For example:

```
Function test {
 Write "one"
 Return "two"
 Write "three"
 }
```

This will output "one" and "two," but never output "three." That's because `return` is basically a special shortcut to `Write-Output`, one that does indeed output to the pipeline, but that then exits the function. In fact, this `test` example is how *most* people encounter their first `return` "gotcha," because they won't expect the function to output anything other than just "two."

In this chapter, however, we got something a bit different. The function will work *sometimes*. Specifically, it'll work fine when you run it with one service name. But give it two names, and it'll only output the status of the first service, because `return` forces it to exit afterwards. A proper rewrite of this function:

```
Function Get-ServiceStatus {
 [CmdletBinding()]
 Param(
  [Parameter(Mandatory=$True)]
  [string[]]$Name
 )
 ForEach ($item in $name) {
  $svc = Get-Service $item
  Write-Output $svc.status
 }
}
```

Generally speaking, this is the pattern you *should* see in PowerShell. PowerShell functions should be designed to work in the pipeline, which means they should output items *to* the pipeline, and be prepared to output multiple items if needed. *Generally* speaking, outside of a class, there's no reason to use the `return` keyword in PowerShell. It's "syntax sugar," and misleading sugar at that.

> There's some debate over whether to `Write-Output $something` versus just running `$something`, without the `Write-Output` command. The end result is the same; there's a tiny amount of additional overhead in using the command, but unless you're enumerating across thousands of objects, you'll never notice, and including the command makes the code more readable.

Whenever I see `return` in a function, I immediately suspect that the author is working based on knowledge of other programming languages. I'll often see them accumulating output in an array, rather than immediately outputting each item to the pipeline–another "no-no" in most PowerShell situations.

Problem 10: True Equality

Here's the code - spot the problem!

```
do {
   get-aduser -filter * | select SamAccountName | ForEach\
-Object {

   if ($_.SamAccountName -eq $username) {
   $username = $firstname+($lastname.Substring(0,$i))

   $i++
   write-host $username
   }
   else {
   $usernameexists = $false

   }
  }
}
until ($usernameexists = $false)

Write-Host "Username is $username" -ForegroundColor Green
```

Spoiler!!

Did you catch it? Being able to statically read code and find a problem is a really key skill, one that takes years, sometimes, to really build up to.

```
until ($usernameexists = $false)
```

That's the problem. SO easy to make, and one I made a LOT in the early days of PowerShell when I was switching from VBScript. You see, = isn't an equality operator in PowerShell, it's an assignment operator. The equality operator is `-eq`:

```
until ($usernameexists -eq $false)
```

Or, if you know `$usernameexists` will always be True or False:

```
until (-not $usernameexists)
```

Fun trivia: the reason PowerShell uses `-gt` and similar operators, versus the more traditional < and such, is because Microsoft wanted to use > for redirection, like shells commonly do. It was difficult to get the parser to understand that sometimes > meant redirection and other times it meant "less than," and so they switched to a different syntax for operators. Pretty much all comparison operators now start with a `-` in PowerShell.

Problem 11: Argumentative

Try to solve this one without running the code. But, it's okay to look at help files, if you want. Because this one is a long, single line, I'm going to kind of arbitrarily break it up to make it easier to read.

```
Start-Process -FilePath "C:\BootStrap\$($Using:MSIFile.Na\
me)"
 -ArgumentList "/passive /norestart"
 -Wait
 -PassThru
```

This probably won't work. Do you know why?

Spoiler!

The clue is in the help files for `Start-Process`:

```
Start-Process
     [-FilePath] <String>
     [[-ArgumentList] <String[]>]
     [-WorkingDirectory <String>]
     [-PassThru]
     [-Verb <String>]
     [-WindowStyle <ProcessWindowStyle>]
     [-Wait]
     [<CommonParameters>]
```

As you see here, `-ArgumentList` expects an *array of strings*. We've just given it one string. The `<String[]>` part is the giveaway, with the `[]` indicating that an array of strings is expected. We'd type that as a comma-separated list, and PowerShell would convert those to an array for us. So, like this, right?

```
-ArgumentList "/passive,/norestart"
```

Nope! With the comma inside the quotes, we're still passing a single string. We need this:

```
-ArgumentList "/passive","/norestart"
```

Where the comma *separates* two distinct strings. So that's probably better. More specifically, Example 7 in the help file shows exactly this usage, highlighting how important it is to explore those help files!

Problem 12: Assignment and Output

Why does this output “pony”?

```
($a = "pony")
```

Spoiler!

This is actually *really* tricky, in terms of what’s happening. Let’s start by just looking at a simpler version:

```
"pony"
```

Whenever PowerShell encounters a command-line that consists of an expression, it has to figure out what to *do* with it. So it calls its default command, `Write-Output`. The above is exactly the same as this, in terms of functionality:

```
Write "pony"
```

Of course, `Write` is just an alias to `Write-Output`, which means it’s functionally the same as:

```
Write-Output "pony"
```

Functionally the same. In fact, these three examples have different actual performance:

- `"pony"` .512ms
- `Write "pony"` 31.37ms
- `Write-Output "pony"` 7.137ms

The use of the alias taking longer kind of makes sense, since PowerShell has to look up the alias first. But it surprises a lot of folks to learn that *using `Write-Output` actually takes longer* than just letting it happen "by default." Obviously, the numbers will differ from system to system based on system load, but the relative differences are what's interesting.

Anyway, consider this:

```
$a = "pony"
```

This will place the string "pony" into `$a` and return nothing. So why does this return something?

```
($a = "pony")
```

There's kind of a little under-the-hood sneakiness going on. In this case, `$a` will still contain "pony:"

```
PS /> ($a = "pony")                                              \

pony
PS /> $a                                                         \

pony
PS />
```

It's just in that specific situation, where you've created a parenthetical expression, PowerShell both executes it *and* returns the result. It's intended to be useful I cases like `if` or `while` constructs, where you use a parenthetical expression to determine logical flow, but you *also* want the assignment to actually take effect. It's kind of a bit niche, meaning you don't see folks use it a lot, but it's one of the "syntax sugar" things that PowerShell has stashed around inside its brain.

Honestly, if you're thinking, "this makes no sense to me," I get it. It's a kind of programmer-y thing, and there's no reason you *have* to use this syntax.

Problem 13: Tricky Parameter Sets

Here's the code (which is in the code samples in GitHub, too):

```
function Test-ParameterSet
{
[CmdletBinding(
DefaultParameterSetName = "all"
)]
Param
(
[Parameter(Mandatory = $false,
Position = 0)]
[ValidateNotNullorEmpty()]
[String]$apiKey,
[ValidateSet("Prod", "Dev")]
[Parameter(Mandatory = $false)]
$environment = "Dev",
[Parameter(Mandatory = $false)]
[int]$maxItems,
[Parameter(Mandatory = $false,
ParameterSetName = "all")]
[switch]$all,
[Parameter(Mandatory = $false,
ParameterSetName = "byCustomerId")]
[Parameter(ParameterSetName = "byId")]
[Parameter(ParameterSetName = "byName")]
[string]$customerId,
[Parameter(Mandatory = $false,
ParameterSetName = "byId")]
[Parameter(ParameterSetName = "byCustomerId")]
[string]$id,
[Parameter(Mandatory = $false,
ParameterSetName = "byName")]
[Parameter(ParameterSetName = "byCustomerId")]
[string]$name
)

Write-Output $PSCmdlet.ParameterSetName

}
```

Now look, that's a lot to read, and it's not especially well-formatted. So let's break it down a bit in English. We have four parameters: `-All`, `-CustomerId`, `-Id`, and `-Name`. Here's how we want them used:

1. We use `-All`, and cannot use any other parameter.
2. We use `-CustomerId`, and do not use any other parameter.
3. Use use `-CustomerId` and also use `-Name`.
4. Use use `-CustomerId` and also use `-Id`.

That right there is a tricky set of requirements. Parameter set #1 is easy, but #2-4 create a problem. You see, in #2, we want only `-CustomerId`, but we don't want any other parameter. So we might be inclined to make `-Name` and `-Id` optional parameters. But in that case, PowerShell *can't differentiate* between sets 2-4. It's let us use `-Name` and `-Id` at the same time, since they're both optional. Optional parameters, basically, can't be used to "put yourself" into a parameter set, because PowerShell can't distinguish between "a set where a parameter isn't available" and "a set where a parameter is available, but not used."

So how could we fix this?

Spoiler!!

You could *probably* define four sets. Set one makes `-All` mandatory, set 2 includes only `-CustomerId` as mandatory, set 3 makes both `-CustomerId` and `-Name` mandatory, and set 4 makes `-CustomerId` and `-Id` mandatory. If you had other usage scenarios, though, things would start to get more complex.

Parameter sets can be super-complex. You kind of have to think about them in terms of how *PowerShell* thinks about them.

PowerShell starts by looking at the parameters you have typed. It looks for any parameter sets *which do not include* those parameters, and basically wipes them from its mind. The only parameter sets in consideration, then, are all the sets that contain every parameter you've typed so far.

You need to keep typing parameters, then, until there's only one set remaining in consideration.

So, consider this:

```
[Parameter(Mandatory, ParameterSetName = "A")]
[switch]
$All,

[Parameter(Mandatory, ParameterSetName = "B")]
[Parameter(Mandatory, ParameterSetName = "C")]
[Parameter(Mandatory, ParameterSetName = "D")]
[string]
$CustomerID,

[Parameter(Mandatory, ParameterSetName = "B")]
[string]
$ID,

[Parameter(Mandatory, ParameterSetName = "C")]
```

```
[string]
$Name
```

Now suppose we type:

```
Whatever-Command -Name "BLAH"
```

Which sets are in consideration? Only "C." It's the only set that contains `-Name`. But we could also add `-CustomerId`, since it "lives" in set "C" also. In fact, we *must* add `-CustomerId`, because it is *mandatory* in all sets in which it exists. Now, let's change the definitions a bit:

```
[Parameter(Mandatory, ParameterSetName = "A")]
[switch]
$All,

[Parameter(Mandatory, ParameterSetName = "B")]
[Parameter(Mandatory, ParameterSetName = "C")]
[Parameter(Mandatory, ParameterSetName = "D")]
[string]
$CustomerID,

[Parameter(ParameterSetName = "B")]
[string]
$ID,

[Parameter(ParameterSetName = "C")]
[string]
$Name
```

Now suppose we type:

```
Whatever-Command -CustomerId "BLAH"
```

Which sets are in play? “B,” “C,” and “D” are. After all `-CustomerId` lives in all three of those. If we added `-Id`, we’d be locked into set “B.” If we added `-Name`, we’d be locked into set “C.” *But there’s no way to lock ourselves into set “D!”* Because `-Name` and `-Id` aren’t mandatory, PowerShell can’t decide which set we’re trying to use! See the difference?

Problem 14: The Business Case

For this problem, we're going to take a slightly different tack. Consider this business need:

> I have a script that needs to send a command to a remote machine. We've enabled PowerShell Remoting, so I can use Invoke-Command to do this. However, the script won't be running under a user account that has permission to connect via Remoting. How can I have the script, which needs to run unattended on a schedule, send an alternate credential?

How would you solve this? Here's a hint: perambulations with the `-Credential` parameter of `Invoke-Command` *are not the best solution.*

Spoiler!!

The problem with the `-Credential` parameter is that, almost no matter what you do, you're storing potentially powerful credentials someplace where they can be retrieved with relative ease. Perhaps not in clear text, but in reality not much better than clear text.

The solution here is PowerShell's ability to create *constrained endpoints* in Remoting. PowerShell's Just Enough Administration, or JEA, module can make these easier to set up, but you can set them up manually, too. *Secrets of PowerShell Remoting*, available at PowerShell.org, covers the basics.

To explain this, let's put some names to things.

- We want to run a **Scheduled Task** that is a PowerShell script.
- The Scheduled Task will run under a **Task User Account** of some kind. This can't be LocalSystem, though; it needs to be a legit user account that both machines can authenticate, like a domain user. But it doesn't need to be an admin of any kind.
- The Scheduled Task needs to use `Invoke-Command` to send commands to a **Remote Computer**. The command(s) being sent are known in advance, and we call that the **Command List**.
- The commands on the Command List need to be executed by a **Privileged User Account** in order to work. This might be an admin.

Here's what we do:

1. On the Remote Computer, we create a new PowerShell Remoting endpoint.
2. We configure the endpoint with the Command List, so it can only run those commands.
3. We configure the endpoint to only provide access to the Task User Account.
4. We configure the endpoint to "Run As" the Privileged User Account.
5. We configure the Scheduled Task to use `Invoke-Command` to send its commands. There's no need to use the `-Credential` parameter.

A safe way of remotely using alternate credentials in a variety of situations, including a scheduled task.

Problem 15: Not Your Father's Programming Language

PowerShell is a *shell*, and it *contains* a programming language. But most programmers wouldn't really love PowerShell as a programming language; it takes a lot of liberties that most languages don't. Here's a good example of where you can go wrong:

```
function GetDriver([string]$RegKey, [string]$oracle_home)\
 {
  rrlog 2 "(L# 102):   RegKey: '$($RegKey)'"
  $theDriverWeWant="some string"
  return $theDriverWeWant
}
$theDriverWeWant = GetDriver $RegKey $oracle_home
```

Go ahead and pick that apart a bit. As-is, it will *not* produce a string output, which is what the author seems to expect. Why or why not, and what else needs fixing?

Spoiler!!

This kind of function declaration almost always tells me we're dealing with a "real programmer" who might be new to PowerShell:

```
function GetDriver([string]$RegKey, [string]$oracle_home)\
 {
```

Now, that's totally *legal* in PowerShell, but it's not how we'd *normally* declare a function and its parameters. This isn't a problem *per se*, but it does make me alert for some of the other "I've been programming in real languages for years" gotchas I'm likely to see. In PowerShell, we'd normally write this as:

```
function GetDriver {
 Param([string]$RegKey,
  [string]$oracle_home)
}
```

So now let's look at where the real problem lies:

```
  rrlog 2 "(L# 102):   RegKey: '$($RegKey)'"
  $theDriverWeWant="some string"
  return $theDriverWeWant
```

That `return` keyword is a huge red flag, because it *suggests* that the author thinks PowerShell functions return a single value. You know, like in every other programming language ever. But that's not the case. The output of a PowerShell command (including functions, which are a kind of command) goes to the *pipeline*, and the pipeline is capable of–nay, is *delighted* to be–handling many items of output.

In our case, the external command `rrlog` is doubtless producing some textual output. Because that hasn't been captured to a variable, it winds up in the pipeline *as the initial output of the function*. The `return` keyword then indeed puts something more in the pipeline and exits the function, but by that time out output is already hopeless corrupted.

The fix?

```
rrlog 2 "(L# 102):   RegKey: '$($RegKey)'" > $null
$theDriverWeWant="some string"
Write $theDriverWeWant
```

Forcing the `rrlog` output to the $null variable basically suppresses that output, and switching to `write` from `return` clears up any suggestion that we might not be familiar with PowerShell's under-the-hood ticks.

Problem 16: Shall I Compare Theeâ€¦

Consider:

```
$string = "11/28/2018"
$string1 = Get-Date $string -Hour 00 -Minute 00 -Second 00
$string2 = Get-Date -Hour 00 -Minute 00 -Second 00
$string1.ToString() -ge $string2.ToString()
```

This works as-expected. The following, however, does not:

```
$string = "11/28/2018"
$string1 = Get-Date $string -Hour 00 -Minute 00 -Second 00
$string2 = Get-Date -Hour 00 -Minute 00 -Second 00
$string1 -ge $string2
```

How come?

Spoiler!!

What PowerShell displays on the screen after you run a command is the result of a complex formatting system, meaning the actual objects produced by the command might not be visible to you. For example, the object produced by `Get-Date` has a milliseconds property which isn't ordinarily displayed. So even a straightforward comparison:

```
(Get-Date) -eq (Get-Date)
```

That'll return False sometimes, because a millisecond elapses between the first `Get-Date` executing and the second one running. So to walk through our "failing" example:

```
$string = "11/28/2018"
```

Our variable now contains a date-only string. PowerShell can parse that into a proper DateTime object:

```
$string1 = Get-Date $string -Hour 00 -Minute 00 -Second 00
```

But this isn't a "full resolution" DateTime, so the millisecond property is unknown.

```
$string2 = Get-Date -Hour 00 -Minute 00 -Second 00
```

We've produced what SHOULD be today's date, but, again, we've missed the milliseconds.

```
$string1 -ge $string2
```

And so we get unexpected output. Always remember: *what you see on the screen isn't necessarily the truth.* Always pipe things to `fl *` and `gm` to make sure you're seeing the *fully real thing* in front of your weak organic eyeballs.

Problem 17: Debug Me

Just start with this code:

```
$Userdata = Import-Csv C:\empID.csv
ForEach ($User in $Userdata) {
  $User = $User.User
  $Department = $Department.Department
  Get-ADUser -Filter "EmployeeNumber -eq '$User'" -Proper\
ties * |
  set-AdUser -Add "'$User' -eq 'Department'"  -Properties\
 Department
}
```

That’s also in the downloadable code samples.

There are three main problems. Can you spot them?

Spoiler!!

First off, the ForEach loop is using `$User` as its enumerator variable, but then immediately overwriting that:

```
ForEach ($User in $Userdata) {
  $User = $User.User
```

The `$User` variable is no longer a single item from $Userdata, which messes up the code a bit further down. Second is this:

```
$Department = $Department.Department
```

The `$Department` variable is undefined to this point; it's probably meant to be:

```
$Department = $User.Department
```

Presuming that `$User` had not been improperly overwritten as noted previously. A correct version might be:

```
$Userdata = Import-Csv C:\empID.csv
ForEach ($User in $Userdata) {
  $User2 = $User.User
  $Department = $User.Department
  Get-ADUser -Filter "EmployeeNumber -eq '$User2'" -Prope\
rties * |
  set-AdUser -Add "'$User2' -eq 'Department'"  -Propertie\
s Department
}
```

Except that the `-Add` parameter is being used improperly. This is probably correct:

```
Set-ADUser -Department $Department
```

Did you get them all?

Problem 18: Literally the Registry

Here's the code:

```
function Test-RegistryValue {
 param (
  [parameter(Mandatory=$true)]
  [ValidateNotNullOrEmpty()]$Path,
  [parameter(Mandatory=$true)]
  [ValidateNotNullOrEmpty()]$Name
 )
 try {
  Get-ItemProperty -Path $Path `
                   -Name $Name `
                   -ErrorAction Stop |
  Out-Null
  return $true
 } catch {
  return $false
 }
}
```

Now, here's that command in use, and it'll generate the wrong results:

```
Test-RegistryValue -Path "HKLM:\SOFTWARE\Misc\*" `
-Name "test"
```

The asterisk is deliberate; the example presumes that there is a key named SOFTWAREMisc*. So what's wrong?

Spoiler!!

The problem with using a "file system" paradigm for things other than the file system is that certain characters in the actual file system, like `?` and `*`, are illegal, and used for special purposes; elsewhere, however, those characters are perfectly legal and shouldn't be treated as special.

The trick here is this:

```
Get-ItemProperty -Path $Path `
                 -Name $Name `
                 -ErrorAction Stop |
```

Should be this:

```
Get-ItemProperty -LiteralPath $Path `
                 -Name $Name `
                 -ErrorAction Stop |
```

The `-LiteralPath` parameter forces all characters to be treated as literals, not as special wildcards. My other suggestion with this function is to change this:

```
return $true
```

And the following `return $false` to be:

```
Write $True
```

And, of course, `Write $False`. The `return` keyword, in anything but a v5+ class, is misleading. It doesn't do what `return` in literally any other language does, and including it suggests that the person using it isn't aware of PowerShell's somewhat inimical use of `return`, meaning they're likely to get caught in a common "gotcha."