26 Tips, tricks, and techniques

We're nearing the end of your month of lunches, so we'd like to share a few random extra tips and techniques to round out your education.

26.1 Profiles, prompts, and colors: Customizing the shell

Every PowerShell session starts out the same: the same aliases, the same PSDrives, the same colors, and so forth. Why not customize the shell a little bit more?

26.1.1 PowerShell profiles

We've explained before that there's a difference between a PowerShell hosting application and the PowerShell engine itself. A hosting application, such as the console or the VS Code, is a way for you to send commands to the PowerShell engine. The engine executes your commands, and the hosting application is responsible for displaying the results. The hosting application is also responsible for loading and running *profile scripts* each time the shell starts.

These profile scripts can be used to customize the PowerShell environment by loading modules, changing to a different starting directory, defining functions that you'll want to use, and so forth. For example, here is the profile script that Sarah uses on her computer:

```
Import-Module ActiveDirectory
Import-Module DBATools
cd c:\
```

The profile loads the two modules that Sarah uses the most, and it changes to the root of her `C:` drive, which is where Sarah likes to begin working. You can put any commands you like into your profile.

NOTE You might think there's no need to load the Active Directory module, because PowerShell will implicitly load it as soon as Sarah tries to use one of the commands in that module. But that particular module also maps an `AD:` PSDrive, and Sarah likes to have that available as soon as the shell starts.

There's no default profile, and the exact profile script that you create will depend on how you want it to work. Details are available if you run `help about_profiles`, but you mainly need to consider whether you'll be working in multiple different hosting applications. For example, we tend to switch back and forth between the regular console, Windows Terminal, and VS Code. We like to have the same profile running for all three, so we have to be careful to create the right profile script file in the right location. We also have to be careful about what goes into that profile, as some commands that tweak console-specific settings such as colors can cause errors in VS Code or Windows Terminal. Here are the files that the console host tries to load, and the order in which it tries to load them:

1. $pshome\profile.ps1—This will execute for all users of the computer, no matter which host they're using (remember that `$pshome` is predefined within PowerShell and contains the path of the PowerShell installation folder).
2. $pshome\Microsoft.PowerShell_profile.ps1—This will execute for all users of the computer if they're using the console host.
3. $pshome/Microsoft.VSCode_profile.ps1—If you are using the VS Code with the PowerShell extension, this script will be executed instead.
4. $home\Documents\WindowsPowerShell\profile.ps1—This will execute only for the current user (because it lives under the user's home directory), no matter which host they're using.
5. $home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1—This will execute for the current user if they're using the console host. If they're using VS Code with the PowerShell extension, the $home\Documents\WindowsPowerShell\Microsoft.VSCode_profile.ps1 script will be executed instead.

If one or more of these scripts doesn't exist, there's no problem. The hosting application will simply skip it and move on to the next one.

On 64-bit systems, there are variations for both 32- and 64-bit scripts, because there are separate 32- and 64-bit versions of PowerShell itself. You won't always want the same commands run in the 64-bit shell as you do in the 32-bit shell—that is, some modules and other extensions are available for only one or the other architecture, so you wouldn't want a 32-bit profile trying to load a 64-bit module into the 32-bit shell, because it won't work.

TRY IT NOW Run `$Profile | Format-List -force` and list out all your profiles.

Note that the documentation in `about_profiles` is different from what we've listed here, and our experience is that the preceding list is correct. Here are a few more points about that list:

- `$pshome` is a built-in PowerShell variable that contains the installation folder for PowerShell itself; on most systems, that's in C:\Program Files\PowerShell\7.
- `$home` is another built-in variable that points to the current user's profile folder (such as C:\Users\Sarah).
- We've used *Documents* to refer to the Documents folder, but on some versions of Windows it will be *My Documents*.
- We've written "no matter which host they're using," but that technically isn't true. It's true of hosting applications (e.g., VS Code) written by Microsoft, but there's no way to force the authors of non-Microsoft hosting applications to follow these rules.

Because we want the same shell extensions to load whether we're using the console host or the VS Code, we chose to customize $home\Documents\WindowsPowerShell\profile.ps1, because that profile is run for both of the Microsoft-supplied hosting applications.

> **TRY IT NOW** Take your profile for a test drive by creating one or more profile scripts for yourself. Even if all you put in them is a simple message, such as `Write-Output "It Worked"`, this is a good way to see the different files in action. Remember that you have to close the shell (or VS Code) and reopen it to see the profile scripts run.

Keep in mind that profile scripts are scripts and are subject to your shell's current execution policy. If your execution policy is `Restricted`, your profile won't run; if your policy is `AllSigned`, your profile must be signed. Chapter 4 discussed the execution policy.

> **TIP** In VS Code, you can run the command `code $profile` and it will open the VS Code profile. Similarly, in the console, you can run `notepad $profile` and it will open your console-specific profile.

26.1.2 Customizing the prompt

The PowerShell prompt—the `PS C:\>` that you've seen through much of this book—is generated by a built-in function called `Prompt`. If you want to customize the prompt, you can replace that function. Defining a new `Prompt` function is something that can be done in your profile script so that your change takes effect each time you open the shell. Here's the default prompt:

```
function prompt
{
    $(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
    else { '' }) + 'PS ' + $(Get-Location) `
    + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

This prompt first tests to see whether the `$DebugContext` variable is defined in the shell's `VARIABLE:` drive. If it is, this function adds `[DBG]:` to the start of the prompt. Otherwise, the prompt is defined as `PS` along with the current location, which is returned by the `Get-Location` cmdlet. If the shell is in a nested prompt, as defined by the built-in `$nestedpromptlevel` variable, the prompt will have >> added to it.

Here's an alternative prompt function. You could enter this directly into any profile script to make it the standard prompt for your shell sessions:

```
function prompt {
 $time = (Get-Date).ToShortTimeString()
 "$time [$env:COMPUTERNAME]:> "
}
```

This alternative prompt displays the current time, followed by the current computer name (which is contained within square brackets):

```
6:07 PM [CLIENT01]:>
```

Note that this uses PowerShell's special behavior with double quotation marks, in which the shell will replace variables (such as `$time`) with their contents.

One of the most useful pieces of code to add to your profile is to change the title bar of your PowerShell Windows:

```
$host.UI.RawUI.WindowTitle = "$env:username"
```

26.1.3 Tweaking colors

In previous chapters, we mentioned how stressed out we can get when a long series of error messages scrolls by in the shell. Sarah always struggled in English class when she was a kid, and seeing all that red text reminds her of the essays she'd get back from Ms. Hansen, all marked up with a red pen. Yuck. Fortunately, PowerShell gives you the ability to modify most of the default colors it uses (figure 26.1).

The default text foreground and background colors can be modified by clicking the control box in the upper-left corner of PowerShell's window. From there, select Properties, and then select the Colors tab.

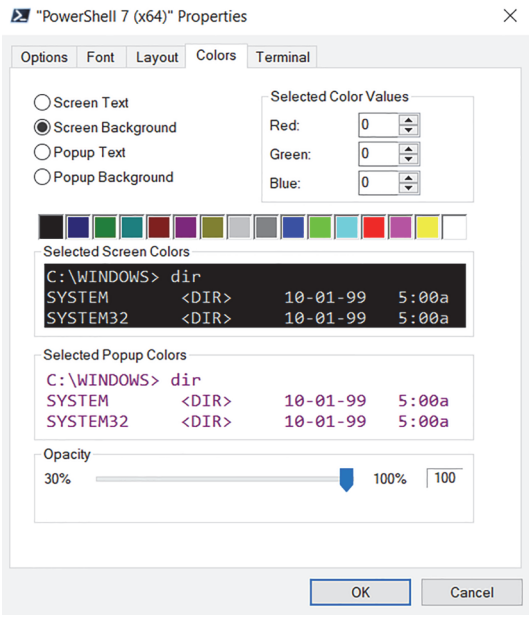

Figure 26.1 Configuring the default shell screen colors

Modifying the colors of errors, warnings, and other messages is a bit trickier and requires you to run a command. But you could put this command into your profile so that it executes each time you open the shell. Here's how to change the error message foreground color to green, which we find a lot more soothing:

```
(Get-Host).PrivateData.ErrorForegroundColor = "green"
```

You can change colors for the following settings:

- `ErrorForegroundColor`
- `ErrorBackgroundColor`
- `WarningForegroundColor`
- `WarningBackgroundColor`
- `DebugForegroundColor`
- `DebugBackgroundColor`
- `VerboseForegroundColor`
- `VerboseBackgroundColor`
- `ProgressForegroundColor`
- `ProgressBackgroundColor`

And here are some of the colors you can choose:

- `Red`
- `Yellow`
- `Black`
- `White`
- `Green`
- `Cyan`
- `Magenta`
- `Blue`

There are also dark versions of most of these colors: `DarkRed`, `DarkYellow`, `DarkGreen`, `DarkCyan`, `DarkBlue`, and so on.

26.2 Operators: -as, -is, -replace, -join, -split, -contains, -in

These additional operators are useful in a variety of situations. They let you work with data types, collections, and strings.

26.2.1 -as and -is

The `-as` operator produces a new object in an attempt to convert an existing object into a different type. For example, if you have a number that contains a decimal (perhaps from the result of a division operation), you can drop the decimal portion by converting, or *casting*, the number to an integer:

```
1000 / 3 -as [int]
```

The object to be converted comes first, then the `-as` operator, and then, in square brackets, the type you want to convert to. Types can include `[string]`, `[xml]`, `[int]`, `[single]`, `[double]`, `[datetime]`, and others, although those are probably the ones you'll use the most. Technically, this example of converting to an integer will round the fractional number to an integer, rather than just truncating the fractional portion of the number.

The `-is` operator works similarly: it's designed to return `True` or `False` if an object is of a particular type or not. Here are a few one-line examples:

```
123.45 -is [int]
"SRV02" -is [string]
$True -is [bool]
(Get-Date) -is [datetime]
```

TRY IT NOW Try running each of these one-line commands in the shell to see the results.

26.2.2 -replace

The `-replace` operator uses regex and is designed to locate all occurrences of one string within another and replace those occurrences with a third string:

```
PS C:\> "192.168.34.12" -replace "34","15"
192.168.15.12
```

The source string comes first, followed by the `-replace` operator. Then you provide the string you want to search for within the source, followed by a comma and the string you want to use in place of the search string. In the preceding example, we replace `34` with `15`.

This is not to be confused with the `string replace()` method, which is a static text replace. While they work similarly, they are very different.

26.2.3 -join and -split

The `-join` and `-split` operators are designed to convert arrays to delimited lists, and vice versa. For example, suppose you create an array with five elements:

```
PS C:\> $array = "one","two","three","four","five"
PS C:\> $array
one
two
three
four
five
```

This works because PowerShell automatically treats a comma-separated list as an array. Now, let's say you want to join this array together into a pipe-delimited string. You can do that with `-join`:

```
PS C:\> $array -join "|"
one|two|three|four|five
```

Saving that result into a variable will let you reuse it, or even pipe it out to a file:

```
PS C:\> $string = $array -join "|"
PS C:\> $string
one|two|three|four|five
PS C:\> $string | out-file data.dat
```

The `-split` operator does the opposite: it takes a delimited string and makes an array from it. For example, suppose you have a tab-delimited file containing one line and four columns. Displaying the contents of the file might look like this:

```
PS C:\> gc computers.tdf
Server1 Windows East    Managed
```

Keep in mind that `gc` is an alias for `Get-Content`.

You can use the `-split` operator to break that into four individual array elements:

```
PS C:\> $array = (gc computers.tdf) -split "`t"
PS C:\> $array
Server1
Windows
East
Managed
```

Notice the use of the escape character, a backtick, and a *t* (`` `t ``) to define the tab character. This has to be in double quotes so that the escape character will be recognized. The resulting array has four elements, and you can access them individually by using their index numbers:

```
PS C:\> $array[0]
Server1
```

26.2.4 -contains and -in

The `-contains` operator causes much confusion for PowerShell newcomers. You'll see folks try to do this:

```
PS C:\> 'this' -contains '*his*'
False
```

In fact, they mean to use the `-like` operator instead:

```
. PS C:\> 'this' -like '*his*'
True
```

The `-like` operator is designed for wildcard string comparisons. The `-contains` operator is used to test whether a given object exists within a collection. For example,

create a collection of string objects, and then test whether a given string is in that collection:

```
PS C:\> $collection = 'abc','def','ghi','jkl'
PS C:\> $collection -contains 'abc'
True
PS C:\> $collection -contains 'xyz'
False
```

The `-in` operator does the same thing, but it flips the order of the operands so that the collection goes on the right and the test object on the left:

```
PS C:\> $collection = 'abc','def','ghi','jkl'
PS C:\> 'abc' -in $collection
True
PS C:\> 'xyz' -in $collection
False
```

26.3 String manipulation

Suppose you have a string of text, and you need to convert it to all uppercase letters. Or perhaps you need to get the last three characters from the string. How would you do it?

In PowerShell, strings are objects, and they come with a great many methods. Remember that a method is a way of telling the object to do something, usually to itself, and that you can discover the available methods by piping the object to `gm`:

```
PS C:\> "Hello" | get-member
   TypeName: System.String
Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
CompareTo        Method               int CompareTo(System.Object value...
Contains         Method               bool Contains(string value)
CopyTo           Method               System.Void CopyTo(int sourceInde...
EndsWith         Method               bool EndsWith(string value), bool...
Equals           Method               bool Equals(System.Object obj), b...
GetEnumerator    Method               System.CharEnumerator GetEnumerat...
GetHashCode      Method               int GetHashCode()
GetType          Method               type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode()
IndexOf          Method               int IndexOf(char value), int Inde...
IndexOfAny       Method               int IndexOfAny(char[] anyOf), int...
Insert           Method               string Insert(int startIndex, str...
IsNormalized     Method               bool IsNormalized(), bool IsNorma...
LastIndexOf      Method               int LastIndexOf(char value), int ...
LastIndexOfAny   Method               int LastIndexOfAny(char[] anyOf),...
Normalize        Method               string Normalize(), string Normal...
PadLeft          Method               string PadLeft(int totalWidth), s...
PadRight         Method               string PadRight(int totalWidth), ...
Remove           Method               string Remove(int startIndex, int...
Replace          Method               string Replace(char oldChar, char...
```

```
Split            Method                 string[] Split(Params char[] sepa...
StartsWith       Method                 bool StartsWith(string value), bo...
Substring        Method                 string Substring(int startIndex),...
ToCharArray      Method                 char[] ToCharArray(), char[] ToCh...
ToLower          Method                 string ToLower(), string ToLower(...
ToLowerInvariant Method                 string ToLowerInvariant()
ToString         Method                 string ToString(), string ToStrin...
ToUpper          Method                 string ToUpper(), string ToUpper(...
ToUpperInvariant Method                 string ToUpperInvariant()
Trim             Method                 string Trim(Params char[] trimCha...
TrimEnd          Method                 string TrimEnd(Params char[] trim...
TrimStart        Method                 string TrimStart(Params char[] tr...
Chars            ParameterizedProperty  char Chars(int index) {get;}
Length           Property               System.Int32 Length {get;}
```

Some of the more useful `String` methods include the following:

- `IndexOf()`—Tells you the location of a given character within the string:

```
PS C:\> "SRV02".IndexOf("-")
6
```

- `Split()`, `Join()`, *and* `Replace()`—Operate similarly to the `-split`, `-join`, and `-replace` operators we described in the previous section. We tend to use the PowerShell operators rather than the `String` methods.
- `ToLower()` *and* `ToUpper()`—Convert the case of a string:

```
PS C:\> $computername = "SERVER17"
PS C:\> $computername.tolower()
server17
```

- `Trim()`—Removes whitespace from both ends of a string.
- `TrimStart()` and `TrimEnd()`—Remove whitespace from the beginning or end of a string, respectively:

```
PS C:\> $username = "    Sarah "
PS C:\> $username.Trim()
Sarah
```

All of these `String` methods are great ways to manipulate and modify `String` objects. Note that all of these methods can be used with a variable that contains a string—as in the `ToLower()` and `Trim()` examples—or they can be used directly with a static string, as in the `IndexOf()` example.

26.4 Date manipulation

Like `String` objects, `Date` (or `DateTime`, if you prefer) objects come with a great many methods that allow date and time manipulation and calculation:

```
PS C:\> get-date | get-member
   TypeName: System.DateTime
Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(System.TimeSpan ...
AddDays              Method         System.DateTime AddDays(double value)
AddHours             Method         System.DateTime AddHours(double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(doub...
AddMinutes           Method         System.DateTime AddMinutes(double va...
AddMonths            Method         System.DateTime AddMonths(int months)
AddSeconds           Method         System.DateTime AddSeconds(double va...
AddTicks             Method         System.DateTime AddTicks(long value)
AddYears             Method         System.DateTime AddYears(int value)
CompareTo            Method         int CompareTo(System.Object value), ...
Equals               Method         bool Equals(System.Object value), bo...
GetDateTimeFormats   Method         string[] GetDateTimeFormats(), strin...
GetHashCode          Method         int GetHashCode()
GetType              Method         type GetType()
GetTypeCode          Method         System.TypeCode GetTypeCode()
IsDaylightSavingTime Method         bool IsDaylightSavingTime()
Subtract             Method         System.TimeSpan Subtract(System.Date...
ToBinary             Method         long ToBinary()
ToFileTime           Method         long ToFileTime()
ToFileTimeUtc        Method         long ToFileTimeUtc()
ToLocalTime          Method         System.DateTime ToLocalTime()
ToLongDateString     Method         string ToLongDateString()
ToLongTimeString     Method         string ToLongTimeString()
ToOADate             Method         double ToOADate()
ToShortDateString    Method         string ToShortDateString()
ToShortTimeString    Method         string ToShortTimeString()
ToString             Method         string ToString(), string ToString(s...
ToUniversalTime      Method         System.DateTime ToUniversalTime()
DisplayHint          NoteProperty   Microsoft.PowerShell.Commands.Displa...
Date                 Property       System.DateTime Date {get;}
Day                  Property       System.Int32 Day {get;}
DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}
DayOfYear            Property       System.Int32 DayOfYear {get;}
Hour                 Property       System.Int32 Hour {get;}
Kind                 Property       System.DateTimeKind Kind {get;}
Millisecond          Property       System.Int32 Millisecond {get;}
Minute               Property       System.Int32 Minute {get;}
Month                Property       System.Int32 Month {get;}
Second               Property       System.Int32 Second {get;}
Ticks                Property       System.Int64 Ticks {get;}
TimeOfDay            Property       System.TimeSpan TimeOfDay {get;}
Year                 Property       System.Int32 Year {get;}
DateTime             ScriptProperty System.Object DateTime {get=if ((& {...
```

Note that the properties enable you to access just a portion of a `DateTime`, such as the day, year, or month:

```
PS C:\> (get-date).month
10
```

The methods enable two things: calculations and conversions to other formats. For example, to get the date for 90 days ago, we like to use `AddDays()` with a negative number:

```
PS C:\> $today = get-date
PS C:\> $90daysago = $today.adddays(-90)
PS C:\> $90daysago
Saturday, March 13, 2021 11:26:08 AM
```

The methods whose names start with `To` are designed to provide dates and times in an alternative format, such as a short date string:

```
PS C:\> $90daysago.toshortdatestring()
3/13/2021
```

These methods all use your computer's current regional settings to determine the correct way of formatting dates and times.

26.5 Dealing with WMI dates

While WMI isn't available in PowerShell 7, we know some of you are still using Windows PowerShell 5.1, so we want to share a tidbit of knowledge about how WMI tends to store date and time information in difficult-to-use strings. For example, the `Win32_OperatingSystem` class tracks the last time a computer was started, and the date and time information looks like this:

```
PS C:\> get-wmiobject win32_operatingsystem | select lastbootuptime
lastbootuptime
--------------
20101021210207.793534-420
```

PowerShell's designers knew you wouldn't be able to easily use this information, so they added a pair of conversion methods to every WMI object. Pipe any WMI object to `gm` and you can see those methods at or near the end:

```
PS C:\> get-wmiobject win32_operatingsystem | gm
   TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingS
ystem
Name                                       MemberType   Definition
----                                       ----------   ----------
Reboot                                     Method       System.Management...
SetDateTime                                Method       System.Management...
Shutdown                                   Method       System.Management...
Win32Shutdown                              Method       System.Management...
Win32ShutdownTracker                       Method       System.Management...
BootDevice                                 Property     System.String Boo...
...
PSStatus                                   PropertySet  PSStatus {Status,...
ConvertFromDateTime                        ScriptMethod System.Object Con...
ConvertToDateTime                          ScriptMethod System.Object Con...
```

We cut out most of the middle of this output so that you can easily find the `Convert-FromDateTime()` and `ConvertToDateTime()` methods. In this case, what you start with is a WMI date and time, and you want to convert that to a normal date and time, so you do it like this:

```
PS C:\> $os = get-wmiobject win32_operatingsystem
PS C:\> $os.ConvertToDateTime($os.lastbootuptime)
Thursday, October 20, 2015 9:02:07 PM
```

If you want to make that date and time information part of a normal table, you can use `Select-Object` or `Format-Table` to create custom, calculated columns and properties:

```
PS C:\> get-wmiobject win32_operatingsystem | select BuildNumber,__SERVER,
[CA]@{l='LastBootTime';e={$_.ConvertToDateTime($_.LastBootupTime)}}
BuildNumber               __SERVER                 LastBootTime
-----------               --------                 ------------
7600                      SRV02                10/20/2015 9:02:07 PM
```

Dates are less of a hassle if you're using the CIM commands, because they automatically translate most date/time values into something human readable.

26.6 Setting default parameter values

Most PowerShell commands have at least a few parameters that include default values. For example, run `Dir` by itself and it defaults to the current path, without you having to specify a `-Path` parameter.

Defaults are stored in a special built-in variable named `$PSDefaultParameterValues`. The variable is empty each time you open a new shell window, and it's meant to be populated with a hash table (which you could do in a profile script, to have your defaults always in effect).

For example, let's say you want to create a new credential object containing a username and password, and have that credential automatically apply to all commands that have a `-Credential` parameter:

```
PS C:\> $credential = Get-Credential -UserName Administrator
-Message "Enter Admin credential"
PS C:\> $PSDefaultParameterValues.Add('*:Credential',$credential)
```

Or, you might want to force only the `Invoke-Command` cmdlet to prompt for a credential each time it's run. In this case, rather than assigning a default value, you'd assign a script block that executes the `Get-Credential` command:

```
PS C:\> $PSDefaultParameterValues.Add('Invoke-Command:Credential',
(Get-Credential -Message 'Enter administrator credential'
-UserName Administrator})
```

You can see that the basic format for the `Add()` method's first argument is `<-cmdlet>:<parameter>`, and `<cmdlet>` and can accept wildcards such as `*`. The second argument

for the `Add()` method is either the value you want to make the default, or a script block that executes another command or commands.

You can always examine `$PSDefaultParameterValues` to see what it contains:

```
PS C:\> $PSDefaultParameterValues
Name                           Value
----                           -----
*:Credential                   System.Management.Automation.PSCredenti
Invoke-Command:Credential      Get-Credential -Message 'Enter administ
```

You can learn more about this feature by reading the shell's `about_parameters_default_values` help file.

> **Above and beyond**
>
> PowerShell variables are controlled by something called *scope*. We offered a brief introduction to scope in chapter 16, and it is something that plays into these default parameter values.
>
> If you set `$PSDefaultParameterValues` at the command line, it'll apply to all scripts and commands run within that shell session. But if you set `$-PSDefaultParameter-Values` within a script, it'll apply only to things done by that script. That's a useful technique, because it means you can start a script with a bunch of defaults, and they won't apply to other scripts, or to the shell in general.
>
> This concept of "what happens in the script, stays in the script" is the heart of scope. You can read more about scope in the shell's `about_scope` help file, if you'd like to explore further on your own.

26.7 Playing with script blocks

Script blocks are a key part of PowerShell, and you've been using them quite a bit:

- The `-FilterScript` parameter of `Where-Object` takes a script block.
- The `-Process` parameter of `ForEach-Object` takes a script block.
- The hash table used to create custom properties with `Select-Object`, or custom columns with `Format-Table`, accepts a script block as the value of the `E,` or `Expression,` key.
- Default parameter values, as described in the previous section, can be set to a script block.
- Some remoting and job-related commands, including `Invoke-Command` and `Start-Job`, accept script blocks on their `-ScriptBlock` parameter.

So what is a script block? In general, it's anything surrounded by curly brackets `{}`, with the exception of hash tables, which use curly brackets but are preceded by the `@` symbol. You can even enter a script block right from the command line and assign it to a variable. You can then use the call operator, `&`, to run the block:

```
PS C:\> $block = {
Get-process | sort -Property vm -Descending | select -first 10 }
PS C:\> &$block
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    680      42    14772      13576  1387     3.84    404 svchost
    454      26    68368      75116   626     1.28   1912 powershell
    396      37   179136      99252   623     8.45   2700 powershell
    497      29    15104       6048   615     0.41   2500 SearchIndexer
    260      20     4088       8328   356     0.08   3044 taskhost
    550      47    16716      13180   344     1.25   1128 svchost
   1091      55    19712      35036   311     1.81   3056 explorer
    454      31    56660      15216   182    45.94   1596 MsMpEng
    163      17    62808      27132   162     0.94   2692 dwm
    584      29     7752       8832   159     1.27    892 svchost
```

You can do quite a bit more with script blocks. If you'd like to explore the possibilities on your own, read the shell's `about_script_blocks` help file.

26.8 More tips, tricks, and techniques

As we said at the outset of this chapter, this is an overview of some random little things that we want to show you but that did not fit neatly into one of the previous chapters. Of course, you'll continue to pick up tips and tricks with the shell as you learn more about it and gain more experience with it.

You can check out our Twitter feeds too—@TylerLeonhardt, @TravisPlunk, and @PSJamesP—where we routinely share tips and techniques that we discover and find useful. And don't forget the forums at PowerShell.org. Sometimes, learning bit by bit can be an easy way to become more proficient in a technology, so consider these and any other sources you run across as a way to incrementally and continually improve your PowerShell expertise.