# Day 1 with PowerShell

## Things I wish I'd known when I started using PowerShell

John D. Cook
http://www.johndcook.com

First released 9 April 2009, updated 1 September 2009

## Introduction

This booklet captures many of the things I wish someone had told me when I first started using PowerShell. It will not teach you how to use PowerShell, but it will tell you some of the things surrounding the use of PowerShell that can be difficult to find. For example, here are some of the questions answered.

- How do I configure PowerShell?
- How do I make PowerShell launch faster?
- How do I get documentation?
- Why did PowerShell make some of the design decisions they did?
- Once I've written some useful functions and scripts, where do I put them?
- Where can I find more PowerShell resources?

There are many excellent resources for learning PowerShell listed in the Resource section at the end. In particular, I recommend reading Bruce Payette's book Windows PowerShell in Action from cover to cover. But this eBook is about 50 times smaller than Payette's book and can help you get your environment set up right away.

Everything here applies to PowerShell version 1.0.

# Installation and configuration

## Download and install PowerShell

PowerShell ships with Windows Server 2008 and will ship with future Microsoft operating systems including Windows 7. You can download PowerShell version 1 for Windows XP Service Pack 2, Windows Server 2003, and Windows Vista here.

PowerShell 1.0 requires the .NET Framework 2.0. If you have the 2.0 framework installed, you can install PowerShell via the Windows Update service.

## Download and install the PowerShell Community Extensions

The PowerShell Community Extensions (PSCX) library is available here.

PSCX is not necessary for running PowerShell, but it does provide a large collection of useful cmdlets and functions. In addition, installing PSCX provides a nice default profile (see section on profiles below) and installs the "Open PowerShell Here" tool. This lets you right-click on a folder in the Windows File Explorer, select "Open PowerShell Here." This will create a new instance of PowerShell whose working directory will be the folder you clicked on.

Some of my favorite features from PSCX are the clipboard functions `Get-Clipboard` and `Out-Clipboard` and the email command `Send-SmtpMail`. It's also fun to play with speech synthesis via `Out-Speech`.

The only drawback to installing PSCX is that it does make PowerShell a little slower to start up.

If you would like the "PowerShell Here" feature without installing PSCX, see Scott Hanselman's blog post on the subject.

## Set the execution policy

PowerShell was designed from the beginning with security in mind. PowerShell will not run scripts by default. You have to set the execution policy before you run your first script.

From the PowerShell prompt, run

```
Set-ExecutionPolicy RemoteSigned
```

This will allow you to run PowerShell scripts that you write but will not allow scripts to run that you download from the Internet or receive as email attachments.

If you do not remember to set the execution policy you might see something like the following.

```
PSH> .\myScript.ps1

File D:\myScript.ps1 cannot be loaded. The file D:\myScript.ps1
is not digitally signed. The script will not execute on the
system. Please see "get-help about_signing" for more details..

At line:1 char:13 + .\myScript.ps1 <<<<
```

## Make PowerShell launch faster

When Microsoft released the first version PowerShell, the installation package neglected to include code to run `NGEN`. This will be fixed in Version 2 of PowerShell, but there is a work-around for the Version 1.

Jeffrey Snover, the leader of the PowerShell development team at Microsoft, explains how to perform the steps the installer left out. On my desktop, the time to launch PowerShell went from around 13 seconds to around 2 seconds after applying the fix he recommends. See the Appendix for a script you only need to run one time that will run `NGEN` and cause PowerShell to open much faster.

Unfortunately, the PowerShell Community Extensions recommended here cause PowerShell to load slower. If you use the extensions, they're worth the increased start time. However, there's no reason not to run the script that makes PowerShell itself load more quickly.

## Configure the command prompt

To configure your command prompt, simply create a function named `prompt` and put it in your profile. For example, the following will cause the working directory to be displayed.

```
function prompt { "$pwd> " }
```

Note that you could put any code you want inside the prompt function and that code will run every time you hit return.

Here's a prompt function that will display the right-most part of the working directory. This keeps long working directory names from taking up most of the space at the command line.

```
function prompt
{
    $m = 30 # maximum prompt length
    $str = $pwd.Path
    if ($str.length -ge $m)
    {
```

```
            # The prompt will begin with "...",
            # end with ">", and in between contain
            # as many of the path characters as will fit,
            # reading from the end of the path.
            $str = "..." + $str.substring($str.length - $m + 4)
        }
        $host.UI.RawUI.WindowTitle = "$env:computername $pwd.path"
        "$str> "
    }
```

For example, if the current directory were

```
C:\Documents and Settings\Administrator\My Documents\My Music
```

then the prompt would be

```
...ator\\My Documents\\My Music>
```

The prompt function above also sets the command window title to contain the computer name and the working directory.

If you want the changes to your prompt to persist, add the `prompt` function to your profile. (See the next section on profiles.)

## Profiles

Profile information can be found in four different files. Why so many? Configuration information might apply to all users on a machine or be specific to a single user. Also, you can run PowerShell via Microsoft's command prompt or through a third-party tool. The four profile files correspond to all for combinations of (all users, single user) and (all shells, just Microsoft's shell). For more information, see the MSDN article Windows PowerShell Profiles. If you're the only user on your PC and you only use Microsoft's shell, the distinction doesn't matter.

There are two additions you can make to your profile to make it easier to debug PowerShell scripts.

First, add this line

```
Set-PSDebug -Strict
```

to your profile to cause PowerShell to complain if you try to use a variable before initializing it. Also, add

```
$ErrorActionPreference = "stop"
```

to cause PowerShell to stop executing a script as soon as an error occurs.

## Getting help

To pull up help from the command prompt, type `Get-Help` followed by the name of the topic you want help on. You may want to add the `-detailed` option to get more information.

To get help about a particular object, use `Get-Member`. For example, you may want to pipe the output of a command to `Get-Member` to learn more about the type of object the command returns.

See Keith Hill's eBook Effective Windows PowerShell for more information on how to use `Get-Help` and `Get-Member`.

The PowerShell Graphical Help File provides the same information as is available from the command line, but in a CHM format that is easier to read.

## Sitting down at the command prompt

PowerShell is case-insensitive. By convention, documentation uses mixed-case in order to make sample code easier to read, but most people will simply type in lower-case at the command prompt.

You can't just type in the name of a script file and run it. You have to specify the path. If the file is in your working directory, you need to put `.\` in front of the file name, as in

```
.\foo.ps1.
```

You could also use `./` since PowerShell accepts forward and backward slashes in paths. You also have to specify full file names.

If a script name has a space in it, you'll need to include the name in quotes and preface it with an ampersand (`&`) in order to run it. For example, to run a script foo.ps1 in your `Documents and Settings` folder, the command would be

```
& "C:/Documents and Settings/foobar.ps1"
```

### Aliases

All PowerShell cmdlets follow the verb-noun naming convention. However, some cmdlets have aliases, nicknames for the same object. For example, `gcm` is a built-in alias for `Get-Command`. You can define your own aliases using the `Set-Alias` cmdlet.

PowerShell comes with numerous aliases to ease the transition to PowerShell form either Windows' `cmd.exe` command line or from Unix shells. Sometimes one cmdlet has multiple aliases. Here are a few examples.

| Alias | Cmdlet |
|---|---|
| **cd** | Set-Location |
| **dir, ls** | Get-ChildItem |
| **copy, cp** | Copy-Item |
| **del , rm** | Remove-Item |

The `Get-Alias` cmdlet can be used to cross-reference aliases and cmdlets. Calling `Get-Alias` without any parameters lists all aliases and their definitions.

To look up the definition of a particular alias, supply that alias as a parameter. For example,

```
Get-Alias cd
```

returns `Set-Location`. Going in the opposite direction takes a little more work. For example, the following command shows how to look up all aliases for the `Remove-Item` cmdlet.

```
get-alias | where-object {$_.definition -match "remove-item"}
```

That command shows that `ri`, `rm`, `rmdir`, `del`, `erase`, and `rd` are all aliases for `Remove-Item`. It does so by searching the output of `get-alias` using a regular expression match.

## Tell me why

This section looks at why the PowerShell team made some of the decisions they did. Understanding the decisions makes it easier to remember the results.

Three of the most surprising features of PowerShell are as follows.

1. Function arguments are delimited with spaces, no commas and no parentheses.
2. The back tick `` ` `` is the escape character, not the back slash `\`.
3. The comparison operators are `-eq`, `-gt`, etc. rather than = or >.

Each of these quirks has a good explanation: **PowerShell is a Windows *shell***.

PowerShell functions do not use parentheses and commas because this would become tedious when using PowerShell from the command line.

PowerShell uses the back tick because Windows uses the backslash as a path separator. PowerShell *allows* forward slashes as path separators but it does not *require* them. Disallowing back slashes in Windows paths would have caused havoc.

Ancient shell tradition dictates that > is an output redirection operator. This is the case in both the Unix and Windows worlds. The PowerShell team decided that violating this tradition would have caused great confusion. The team also placed a high priority on consistency. So if "greater than" must be represented by letters rather than by a symbol, the same should be true for other comparison operators.

There is one exception to the "no parentheses, no commas" rule for function calls: when calling methods on .NET objects, use parentheses around the argument list and separate arguments with commas.

## Next questions

### Once I've written useful functions and scripts, where do I put them?

If you've written a function that you think you may use frequently, you could put it in your profile. Or you may want to have an alias file separate from your profile but called from your profile. The way you'd do this is to "dot source" the function file from your profile: add a line to your profile consisting of a period, a space, and the name of your function file. For example:

```
. c:\foo\bar\myfunctions.ps1
```

Dot sourcing a file brings that files definitions into the current session.

Now suppose you've written some scripts that you want to call conveniently. One option would be to put your script files to a directory listed in your Windows path (or equivalently, add your scripts directory to your Windows path). Another option would be to create a PSDrive (roughly, a mapped drive that only PowerShell knows about) and put all your scripts there. For example, the command

```
New-PSdrive -name scripts -PSprovider filesystem –root c:\foo\bar
```

creates a PSDrive named `scripts` pointing to the `c:\foo\bar` directory on your computer. To run a script

```
c:\foo\bar\myscript.ps1
```

you could type

```
scripts:myscript.ps1
```

from the command prompt.

### How do I know what has been defined in memory?

Functions defined at any given time are part of the `function` PSDrive. You can work with this drive just as you'd work with any other drive. For example, the command `dir function:` will list all

functions. The command `dir function:/G*` will list all functions whose name with "G." Similarly, the `variable` drive contains all variables defined in the current session.

# Resources

## Blogs and web sites

PowerShellCommunity.org portal for many other resources

Windows PowerShell Blog by the PowerShell development team

Dmitry's PowerBlog by Dmitry Sotnikov

The PowerShell Guy by Marc van Orsouw (a.k.a `. /\/\o\/\/`)

PowerShell Community blog aggregator

Windows PowerShell Owner's Manual from Microsoft

## Podcasts

Power Scripting by Jonathan Walz and Hal Rottenberg

Get-Scripting by Alan Renouf and Jonathan Medd

## Books

Windows PowerShell in Action by Bruce Payette (2nd edition available as early access)

Mastering PowerShell by Tobias Weltner (free eBook)

Effective Windows PowerShell by Keith Hill (free eBook)

PowerShell Cookbook by Lee Holmes

## Tools

PowerGUI free graphical interface and editor

PowerShell Plus interactive console from Idera

PowerShell Toolbar by Shay Levy

# Appendix

The following code comes from the PowerShell team blog. Copy the following code into a text file named `update-gac.ps1` and run the file by typing

```
.\update-gac.ps1
```

from the PowerShell prompt. This will cause PowerShell to launch significantly faster.

```
Set-Alias ngen (Join-Path
([System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory()) ngen.exe)
[AppDomain]::CurrentDomain.GetAssemblies() |
    sort {Split-path $_.location -leaf} |
    %{
        $Name = (Split-Path $_.location -leaf)
        if ([System.Runtime.InteropServices.RuntimeEnvironment]::FromGlobalAccessCache($_))
        {
            Write-Host "Already GACed: $Name"
        }else
        {
            Write-Host -ForegroundColor Yellow "NGENing      : $Name"
            ngen $_.location | %{"`t$_"}
         }
      }
```

# About

This document is available at http://www.johndcook.com/

Thanks to the people who reviewed this booklet and made helpful suggestions:
Clift Norris, Steven Murawski, Sarah Edmonson.

I hope you've found this helpful. Please contact me if you have any comments or corrections.

I blog daily at The Endeavour.

John D. Cook