

According to Microsoft.com, *PowerShell is a cross-platform task automation solution made up of a command-line shell, a scripting language, and a configuration management framework. PowerShell runs on Windows, Linux, and macOS.*

It is a scripting language, designed and used to automate CRMs or management systems. It is also used to build, test, and deploy software solutions. PowerShell is built within a .NET Common Language Runtime (CLR). All inputs and outputs are .NET commands.


### [What is Powershell](#)

#### **Objects:**

A function of Powershell---Unix shells, the file and language is plain text. This simplicity allows for end users to input and output all scripts. The execution will likely be stable and functional.

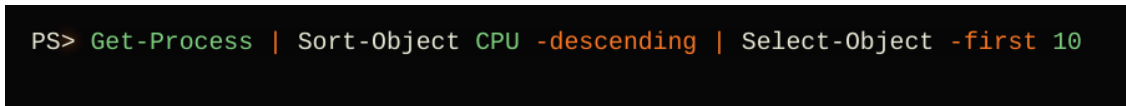
The downside is that inspecting specific data points, problematic of text parsing, and it makes working with anything other than text a challenge.

End users familiar with Python, Ruby, and JavaScript, or a similar language background will be able to implement and utilize Powershell effectively:



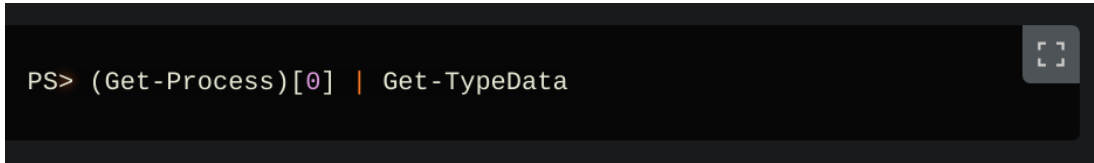
```
PS> Get-Process
```

A long string of text will emerge:



```
PS> Get-Process | Sort-Object CPU -descending | Select-Object -first 10
```

End users should see a shorter list, reverse sorted by CPU time. In the headers atop of each column each row of this table is actually a System.Diagnostics.Process object, not just a row of a table:



```
PS> (Get-Process)[0] | Get-TypeData
```

The Get-Process command returned a list of Process objects. This enabled end users to be able to select the first one through indexing, without splitting the output by \n! entry. and allowed for a

shuffle through to the `Get-TypeData` command. These items being objects allot more programming power. Now we can explore the possibility of a `ProcessName` command:

```
PS> Get-Process | Sort-Object CPU -descending | Select Object ProcessName -first 10
```

This command allowed end users to cut fields delimited by tabs or colons and count which field was desired. By implementing this command, the program was told that we wanted the `ProcessName` attribute. Away with parsing, splitting, joining, formatting output, etc.

## Objects and Types:

.NET and object must have a type:

```
PS> 2 + 2
4
```

```
PS> 4.2 / 3
1.4
```

PowerShell usually does a good job of trying to figure out the types. However, if end users really want to enforce a type, this can be done by prepending the object with its type in square brackets:

```
PS> "2" + 2
22 # A string
PS> [Int]"2" + 2
4 # An integer. The conversion only applies to the "2"
```

PowerShell commands in comparison to the Bash commands are longer. The creators of PowerShell wanted the commands to be intuitive, such that end users could almost guess the command needed.

PowerShell commands were designed after a simple pattern: "Verb-Noun." The creators of PowerShell tried to keep the number of verbs to a minimum. Common ones you'll see are *Get*,

*New, Add, Clear, Export, Remove, Set, Update, and Write*. The nouns are also usually pretty intuitive: *Process, Item, Object, Date, Job, and Command* for example.

The Get-Command is one of a few commands that is highly common and useful:

```
PS> Get-Command -noun Job
```

### Aliases:

Powershell has built-in aliases to make programming easier and more comfortable:

```
PS> Get-ChildItem

# You can also do:
PS> gci
PS> dir

# And just to make you feel a little bit more at home..
PS> ls
```

Available aliases, the command is:

```
PS> Get-Alias
```

When performing command-line work by yourself, use as many aliases as you want. However, if writing a script or sharing code with someone, it's best to type the whole command and whole flag names out.

### Useful Commands to Get Started:

Not Sure Which Command to Use

Get Command:

```
PS> Get-Command
```

This command will provide you with more information about available commands.

Format-List:

```
PS> Get-Command Get-Alias | Format-List
```

In order to get more information about one or two commands, enter the output into Format-List. This will give you the options, location, and some other useful features.

Get Help Command:

```
PS> Get-Help command  
  
# You can also get help by adding the ? parameter  
PS> command -?
```

A highly useful intuitive command. Actually, Get-Help has quite a few useful flags as well:

```
PS> Get-Command Get-Help | Format-List  
  
# Or, if you're feeling cheeky:  
PS> Get-Help Get-Help
```

```
PS> Get-Help Get-Alias -examples
```

Search for the Properties Your Object Has:

```
PS> Get-Process | Get-Member

# Another similar command:

PS> (Get-Process)[0] | Format-List
```

Search for unknown data or for possible data points available, the following commands will help end user locate objects better.

Get a Portion of the Data:

```
PS> Get-Process | Select-Object Id, ProcessName -last 20
```

Filter Your Data:

```
PS> Get-Process | Where-Object WS -gt 150MB
```

This example of the Where-Object command is the simplest one. In the example above, the data point selected was only the processes whose working set (memory usage) was greater than 150MB.

Basic Unix Commands Translated:

```
# pwd
PS> Get-Location # or gl or pwd

# ls
PS> Get-ChildItem # or gci or dir or ls

# cd
PS> Set-Location # or sl or chdir or cd

# cp
PS> Copy-Item # or copy or cpi or cp

# mv
PS> Move-Item # or move or mi or mv

# cat
PS> Get-Content # or gc or type

# mkdir
PS> New-Item -ItemType Directory # or ni -ItemType Directory or mkdir

# touch
PS> New-Item -ItemType File # or ni

# rm
PS> Remove-Item # or del or erase or ri or rm

# rm -rf
PS> Remove-Item -Recurse -Force # or rm -recurse -force

# head or tail
PS> Select-Object -first # or -last
# usage: Get-Process | Select-Object -first 10

# find
PS> Get-ChildItem -filter *.rb -recurse .
# but, for a slightly easier to read version:
PS> Get-ChildItem -filter *.rb -recurse . | Select-Object FullName
```

See link to image [here](#).

## Access the Path and Other Environment Variables:

In PowerShell, functions are like file locations and environment variables are no exception. These special groups of file-like variables are called PSDrives. In the same way you can ask the C: drive what file is at "\Users\ryan\desktop" with a `Get-ChildItem C:\Users\ryan\Desktop`, you can do the same thing with `env:`, the environment PSDrive:

```
PS> Get-ChildItem env:  
  
# and to get a specific one  
PS> Get-Content env:PATH
```

End users get to an environment variable this way as well:

```
PS> $env:PATH
```

## Customizing Your Profile:

End users can do so with the `$profile` command.

There are several profiles, depending on which "Host" used to interface with PowerShell. If end users are using the regular PowerShell command line, the name of the profile will be `Microsoft.PowerShell_profile.ps1`. However, if end users are working in the PowerShell Integrated Scripting Environment (ISE), the profile will be `Microsoft.PowerShellISE_profile.ps1`:

```
PS> $profile
```

To create a profile that will work for the ISE or the regular command line, use: `$profile.CurrentUserAllHosts`. To configure a profile for all users on your computer, implement: `$profile.AllUsersCurrentHost`. To see them all you can use this command:

```
PS> $profile | Get-Member -type NoteProperty
```

The simplest way to check if you already have a profile is:

```
PS> Test-Path $profile
```

To start creating a profile:

```
# Use whichever editor you love best
PS> code $profile
```

Examples of useful settings:

```
# Microsoft.PowerShell_profile.ps1

# You can customize the window itself by accessing $Host.UI.RawUI
$window = $Host.UI.RawUI
$window.WindowTitle = "Pa-pa-pa-pa-pa-POWERSHELL"
$window.BackgroundColor = "Black"
$window.ForegroundColor = "Gray"

# You can define functions (remember to follow the Verb-Noun convention!)
function Count-History {
    (Get-History | Measure-Object).count
}

function beep {
    echo `a
}

function Edit-Profile {
    [CmdletBinding()]
    [Alias("ep")]
    PARAM()

    vim $profile
}

# You can set aliases.
# NOTE: In PowerShell, you can only alias simple commands.
# Unlike Bash, you can't alias commands with arguments flags.
# If you want to do that, you should define a function instead.
Set-Alias touch New-Item # old habits die hard, amirite?

# You can customize your prompt!
function prompt {
    # ... see the next section for details
}
```

See a link to the image [here](#).

## The Simple Prompt:

The simplest way to customize your prompt is by defining the prompt function, either manually or in your profile. For example:

```
function prompt {
    $histCount = (Get-History | Measure-Object).count
    return "POWERSHELL LEVEL OVER $histCount THOUSAND! >"
}
```

This image is an example of how to print the number of inputs already typed inside the prompt. Whatever string returned is what gets set as the prompt.

### The Complicated Prompt:

```
function prompt {
    $loc = (Get-Location).Path.Replace("$HOME", "~")
    $gitBranch = git branch | Select-String "\*"
    if (!$gitBranch) {
        $gitBranch = ""
    } else {
        $gitBranch = $gitBranch.ToString().Replace("* ", "")
    }
    $histCount = (Get-History | Measure-Object).count
    Write-Host -ForegroundColor yellow "`n $loc"
    Write-Host -NoNewLine "PS [$histCount] $gitBranch ->"
    return " "
}
```

This is an example of a multi-line prompt by using Write-Host a several times.

The prompt yields as such:

```
~/Documents/blog
PS [102] master ->
```

### Clean Up the Commands:

As a Windows product, PowerShell runs in a window with not very many customization options. Here are two good alternatives:

1. [Cmder](#). This is built on top of ConEmu, which is a popular terminal emulator for Windows; would be a smooth transition for Notepad++ users.
2. [Hyper](#). The customization and settings are all done in JavaScript and CSS. As a JavaScript product, there are many plugins available, some of which are stable and useful. Hyper is under construction, so end users may experience some stability issues.

### More Resources:

The best resource is [microsoft sanctioned documentation](#). PowerShell is now open-source, feel free to check out their [Github files](#).

Books available would be: [Windows PowerShell Cookbook](#) by Lee Holmes and [Windows PowerShell in Action](#) by Bruce Payette and Richard Siddaway.



Resource: [Powershell Tutorial](#)

Tags: powershell, powershell101, mspowershell, microsoftpowershell, github, powershellgithub, objects, types, aliases, profiles, createaprofile, commands, variable, path, css, javascript, getcommand, powershellhelp, windowspowershell