2

PowerShell Scripting Fundamentals

Now that you have learned how to get started with PowerShell, let's have a closer look at PowerShell scripting fundamentals to refresh our knowledge.

We will start with the basics, such as working with variables, operators, and control structures. Then, we will dive deeper, putting the big picture together when it comes to cmdlets, functions, and even modules.

After working through this chapter, you should be able to create your very own scripts and even know how to create your own modules.

In this chapter, we are going to cover the following topics:

- Variables
- Operators
- Control structures
- Naming conventions
- Cmdlets
- Functions
- Aliases
- Modules

Technical requirements

For this chapter, you will need the following:

- PowerShell 7.3 and above
- Visual Studio Code
- Access to the GitHub repository for `Chapter02`: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter02**

Variables

A **variable** is a storage location that developers can use to store information with a so-called *value*. Variables always have names that allow you to call them independently of the values that are stored within. In PowerShell, the **$** sign at the beginning indicates a variable:

```
> $i = 1
> $string = "Hello World!"
> $this_is_a_variable = "test"
```

Variables are great for storing simple values, strings, and also the output of commands:

```
> Get-Date
Monday, November 2, 2020 6:43:59 PM
> $date = Get-Date
> Write-Host "Today is" $date
Today is 11/2/2020 6:44:40 PM
```

As you can see in these examples, not only can we store strings and numbers within a variable, we can also store the output of a cmdlet such as **`Get-Date`** and reuse it within our code.

Data types

In contrast to other scripting or programming languages, you don't necessarily need to define the data type for variables. When defining a variable, the data type that makes the most sense is automatically set:

```
> $x = 4
> $string = "Hello World!"
> $date = Get-Date
```

You can find out which data type was used with the **`GetType()`** method:

```
> $x.GetType().Name
Int32
> $string.GetType().Name
String
> $date.GetType().Name
DateTime
```

In PowerShell, data types are automatically set. When defining variables in an automated way, sometimes it can happen that the wrong variable type is set. For example, it can happen that an integer was defined as a string. If you spot a conflict, the **`GetType()`** method helps you to find out which data type was set.

Overview of data types

The following table shows a list of variable data types with their description:

`[string]`	`System.String`. A simple string, which is a commonly used data type.
`[char]`	Unicode 16-bit character
`[byte]`	8-bit unsigned character
`[int], [int32]`	32-bit signed integer
`[long]`	64-bit signed integer
`[bool]`	Boolean: Can be `True` or `False`
`[decimal]`	128-bit decimal value
`[single], [float]`	Single-precision 32-bit floating point number
`[double]`	Double-precision 64-bit floating point number
`[datetime]`	Date and time
`[array]`	Array of values
`[hashtable]`	Hashtable object
`[guid]`	Globally unique identifier (GUID) – example, created by `New-Guid`
`[psobject], [PSCustomObject]`	PowerShell object
`[scriptblock]`	PowerShell script block
`[regex]`	Regular expression
`[timespan]`	Timespan object – example, created by `New-TimeSpan`

Table 2.1 – Variable data types

These are the most common data types that you will come across when working with PowerShell. This is not a complete list, so there might also be other variables that you will encounter: using **`GetType()`** helps you identify the variable data type.

In PowerShell, all data types are based on .NET classes; to get more information on each class, you can refer to the official Microsoft documentation:

- **https://learn.microsoft.com/en-us/dotnet/api/system**
- **https://learn.microsoft.com/en-us/dotnet/api/system.management.automation**

Casting variables

Normally, there's no need to declare data types, as PowerShell does it by itself. But sometimes there might be a need to change the data type – for example, if a list of imported number values is treated like a string instead of **`int`**:

```
> $number = "4"
> $number.GetType().Name
String
```

If you are processing values that have the wrong data type declared, you will either see nasty error messages (because only another input is accepted) or your code will not work as expected.

If the **`$number`** variable was declared as a string and we perform an addition, a mathematical operation will not be performed. Instead, both are concatenated as a string:

```
> $number + 2
42
```

Although 42 might be the answer to the ultimate question of life, the universe, and everything, it is not the expected answer for our equation:

when adding *4 + 2*, we expect the result *6*, but since *4* is treated as a string, *2* will be concatenated and the string *42* is shown as a result:

```
> ($number + 2).GetType().Name
String
```

Especially when parsing files or input, it can happen that variables are not set correctly. If that happens, error messages or wrong operations are the results. Of course, this behavior is not strictly limited to integers and strings: it can basically occur with every other data type as well.

If you discover that a wrong data type is set, you can convert the data type by **casting** it to another type.

If we want, for example, to process **`$number`** as a normal integer, we need to cast the variable type to **`[int]`**:

```
> $int_number = [int]$number
> $int_number.GetType().Name
Int32
```

Now, **`$int_number`** can be processed as a normal integer, and performing mathematical operations works as expected:

```
> $int_number + 2
6
```

You can also cast a Unicode hex string into a character in PowerShell by using the hex value of the Unicode string and casting it to **`[char]`**:

```
> 0x263a
9786
> [char]0x263a
☺
```

Most of the time, the right variable data type is already set automatically by PowerShell. Casting data types helps you to control how to process the data, avoiding wrong results and error messages.

Automatic variables

Automatic variables are built-in variables that are created and maintained by PowerShell.

Here is just a small collection of commonly used automatic variables that are important for beginners. You might find other automatic variables used in later chapters:

- **`$?`**: The execution status of the last command. If the last command succeeded, it is set to **`True`**, otherwise, it is set to **`False`**.
- **`$_`**: When processing a pipeline object, **`$_`** can be used to access the current object (**`$PSItem`**). It can also be used in commands that execute an action on every item, as in the following example:

```
Get-ChildItem -Path C:\ -Directory -Force -ErrorAction SilentlyContinue | ForEach-Object {
    Write-Host $_.FullName
```

```
}
```

- **`$Error`**: Contains the most recent errors, collected in an array. The most recent error can be found in **`$Error[0]`**.
- **`$false`**: Represents the traditional Boolean value of **`False`**.
- **`$LastExitCode`**: Contains the last exit code of the program that was run.
- **`$null`**: Contains **`null`** or an empty value. It can be used to check whether a variable contains a value or to set an undefined value when scripting, as **`$null`** is still treated like an object with a value.
- **`$PSScriptRoot`**: The location of the directory from which the script is being run. It can help you to address relative paths.
- **`$true`**: Contains **`True`**. You can use **`$true`** to represent **`True`** in commands and scripts.

For a complete list of automatic variables, please review the official documentation: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables**.

Environment variables

Environment variables store information about the operating system and paths that are frequently used by the system.

To show all environment variables within your session, you can leverage **`dir env:`**, as shown in the following screenshot:

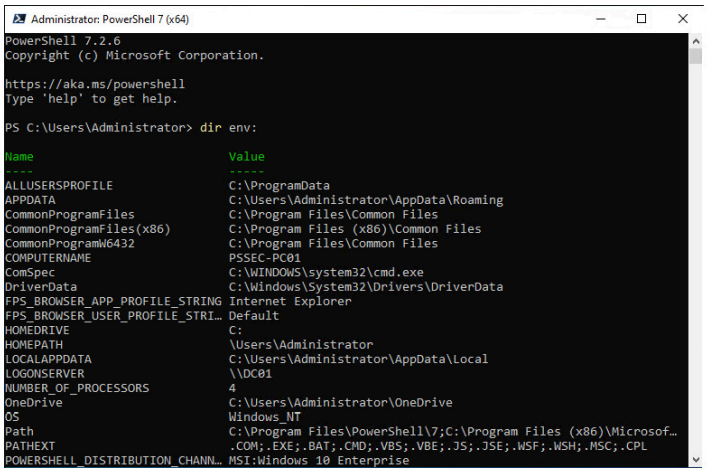

Figure 2.1 – Environment variables

You can directly access and reuse those variables by using the prefix **`$env:`**:

```
> $env:PSModulePath
C:\Users\PSSec\Documents\WindowsPowerShell\Modules;C:\Program Files\WindowsPowerShell\Modules;C:\W
```

To learn more about how to access and process environment variables, have a look at the official documentation: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_environment_variables**.

Reserved words and language keywords

Some words are reserved by the system and should not be used as variables or function names, as this would lead to confusion and unexpected behavior of your code.

By using **`Get-Help`**, you can get a list and more information on reserved words:

```
> Get-Help about_reserved_words
```

Also see the **`about_Language_Keywords`** help pages to get a detailed overview and explanation of all language keywords:

```
> Get-Help about_Language_Keywords
```

Here's an overview of all the language keywords that were available when this book was written:

```
Begin               Enum              Param
Break               Exit              Process
Catch               Filter            Return
Class               Finally           Static
Continue            For               Switch
Data                ForEach           Throw
Define              From              Trap
Do                  Function          Try
DynamicParam        Hidden            Until
Else                If                Using
Elseif              In                Var
End                 InlineScript      While
```

To learn more about a certain language keyword, you can use **`Get-Help`**:

```
> Get-Help break
```

Some reserved words (such as **`if`**, **`for`**, **`foreach`**, and **`while`**) have their own help articles. To read them, add **`about_`** as a prefix:

```
> Get-Help about_If
```

If you don't find a help page for a certain reserved word, as not every one has its own page, you can use **`Get-Help`** to find help pages that write about the word you are looking for:

```
> Get-Help filter -Category:HelpFile
```

Keep those reserved words in mind and avoid using them as function, variable, or parameter names. Using reserved words can and will lead to a malfunction of your code.

Variable scope

When working with PowerShell variables, you want to restrict access. If you use a variable in a function, you don't want it to be available by de-

fault on the command line – especially if you are processing protected values. PowerShell variable scopes protect access to variables as needed.

In general, variables are only available in the context in which they were set, unless the scope is modified:

```
$script:ModuleRoot = $PSScriptRoot
# Sets the scope of the variable $ModuleRoot to script
```

Scope modifier

Using the scope modifier, you can configure the scope in which your variables will be available. Here is an overview of the most commonly used scope modifiers:

- **`global`**: Sets the scope to **`global`**. This scope is effective when PowerShell starts or if you create a new session.

For example, if you set a variable to **`global`** within a module, once the module is loaded and the part is run in which the variable is set to **`global`**, this variable will be available in the session – even if you don't run other functions of this module.

- **`local`**: This is the current scope. The **`local`** scope can be the **`global`** scope, the **`script`** scope, or any other scope.
- **`script`**: This scope is only effective within the script that sets this scope. It can be very useful if you want to set a variable only within a module that should not be available after the function was called.

To demonstrate how variable scopes work, I have prepared a little script, **`Get-VariableScope.ps1`**, which can be found in **`Chapter02`** of this book's GitHub repository: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter02/Get-VariableScope.ps1**.

In the script, the **`Set-Variables`** function is declared first. If this function is called, it sets variables of three scopes – **`local`**, **`script`**, and **`global`** – and then outputs each variable.

Then, the **`Set-Variable`** function is called by the same script. After calling the function, the variables are written to the output:

```
PS C:\Users\Administrator\Documents\GitHub\PowerShell-Automation-and-Scripting-for-CyberSecurity\Chapter02
> .\Get-VariableScope.ps1
########################################################
This is how our variables look in the function, where we defined the variables - in a LOCAL SCOPE:
  Local:  Hello, I'm a local variable.
  Script:  Hello, I'm a script variable.
  Global:  Hello, I'm a global variable.
########################################################
This is how our variables look in the same script - in a SCRIPT SCOPE:
  Local:
  Script:  Hello, I'm a script variable.
  Global:  Hello, I'm a global variable.
```

Figure 2.2 – Calling variables with a local, script, and global scope

While the variables were just set in the **`local`** scope, all configured variables are available when called in this context (**local scope**).

If the same script tries to access the defined variables outside of the function in which the variables were configured, it can still access the vari-

ables that were configured for the **`script`** and **`global`** scope. The variable with the **`local`** scope is inaccessible, as the variables were called in the **script scope**.

After running the **`Get-VariableScope.ps1`** script, try to access the variables on the command line yourself (**global scope**):

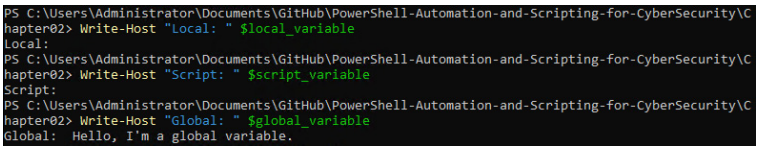

Figure 2.3 – Accessing the variables on the command line

You can imagine scopes as *containers for variables* therefore, in this case, we can only access variables within the **`global`** scope container. The variables with the **`local`** and **`script`** scopes are inaccessible from the command line when not called from the script they were defined in.

When working with scopes, it is advisable to *choose the scope that offers the minimum required privileges* for your use case. This can help prevent accidental script breakage when running scripts multiple times in the same session. While using the **`global`** scope is not necessarily problematic from a security standpoint, it is still best to avoid it when not strictly necessary.

WORKING WITH MODIFIED SCOPE VARIABLES

When you are working with **`script`** *and* **`global`** *scope variables, it is a good practice to always use the variable with the modifier:* **`$script:script_variable`** / **`$global:global_variable`**.

*Although it is possible to use the variable without the modifier (***`$script_variable`** / **`$global_variable`***), using it with the modifier helps you to see at one glance whether the scope of a variable was changed, helps you with your troubleshooting, and avoids confusion.*

Scopes are not only restricted to variables; they can also be used to restrict functions, aliases, and PowerShell drives. Of course, there are also many more use cases for scopes than the ones I described in this section.

If you are interested to learn more about scopes (not only variable scopes) and advanced use cases, have a look at the official documentation: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes**.

Operators

Operators help you not only to perform mathematical or logical operations but they are also a good way to compare values or redirect values.

Arithmetic operators

Arithmetic operators can be used to calculate values. They are as follows:

- **Addition** (`+`):

```
> $a = 3; $b = 5; $result = $a + $b
> $result
8
```

- **Subtraction** (`-`):

```
> $a = 3; $b = 5; $result = $b - $a
> $result
2
```

- **Multiplication** (`*`):

```
> $a = 3; $b = 5; $result = $a * $b
> $result
15
```

- **Division** (`/`):

```
> $a = 12; $b = 4; $result = $a / $b
> $result
3
```

- **Modulus** (`%`): In case you have never worked with modulus in the past, `%` is a great way to check whether there is a remainder if a number is divided by a divisor. Modulus provides you with the remainder:

```
> 7%2
1
> 8%2
0
> 7%4
3
```

Of course, you can also combine different arithmetic operators as you are used to:

```
> $a = 3; $b = 5; $c = 2
> $result = ($a + $b) * $c
> $result
16
```

When combining different arithmetic operators in PowerShell, the operator precedence is respected, as you are used to from regular mathematic operations.

SEMICOLONS, (CURLY) BRACES, AND AMPERSANDS

In this example, we are using the semicolon to execute multiple commands on a single line: in PowerShell, a **semicolon** *(`;`) is functionally equivalent to a carriage return.*

It is also worth noting that the use of reserved characters such as **curly braces** `{}`*,* **parentheses** `()`*, and* **ampersands** `&` *can have a significant impact on script execution. Specifically,* **curly braces** *denote a code block, while* **parentheses** *are used to group expressions or function parameters. The* **ampersand** *is used to invoke an executable or command as if it were a cmdlet.*

To avoid issues with script execution, it is essential to be aware of these reserved characters and their specific use cases.

Comparison operators

Often, it is necessary to compare values. In this section, you will find an overview of comparison operators in PowerShell:

- Equal (`-eq`): Returns `True` if both values are equal:

```
> $a = 1; $b = 1; $a -eq $b
True
> $a = 1; $b = 2; $a -eq $b
False
```

In an **array context**, operators behave differently: when an array is used as the left-hand operand in a comparison, PowerShell performs the comparison operation against each element in the array.

When using comparison operators in an array context, the operation will return the elements selected by the operator:

```
> "A", "B", "C", "D" -lt "C"
A
B
```

When used in an array context, the `-eq` operator behaves differently from its typical comparison behavior. Instead of checking whether the two operands are equal, it returns all elements in the left-hand operand array that are equal to the right-hand operand. If no matches are found, the operation will still return `False`:

```
> "A","B","C" -eq "A"
A
```

- Not equal (`-ne`): Returns `True` if both values are not equal:

```
> $a = 1; $b = 2; $a -ne $b
True
> $a = 1; $b = 1; $a -ne $b
False
> "Hello World!" -ne $null
True
> "A","B","C" -ne "A"
B
C
```

- Less equal (`-le`): Returns `True` if the first value is less than or equal to the second value:

```
> $a = 1; $b = 2; $a -le $b
True
> $a = 2; $b = 2; $a -le $b
True
> $a = 3; $b = 2; $a -le $b
False
```

```
> "A","B","C" -le "A"
A
```

- Greater equal (`-ge`): Returns `True` if the first value is greater than or equal to the second value:

```
> $a = 1; $b = 2; $a -ge $b
False
> $a = 2; $b = 2; $a -ge $b
True
> $a = 3; $b = 2; $a -ge $b
True
> "A","B","C" -ge "A"
A
B
C
```

- Less than (`-lt`): Returns `True` if the first value is less than the second value:

```
> $a = 1; $b = 2; $a -lt $b
True
> $a = 2; $b = 2; $a -lt $b
False
> $a = 3; $b = 2; $a -lt $b
False
> "A","B","C" -lt "A" # results in no output
```

- Greater than (`-gt`): Returns `True` if the first value is greater than the second value:

```
> $a = 1; $b = 2; $a -gt $b
False
> $a = 2; $b = 2; $a -gt $b
False
> $a = 3; $b = 2; $a -gt $b
True
> "A","B","C" -gt "A"
B
C
```

- `-like`: Can be used to check whether a value matches a wildcard expression when used with a scalar. If used in an array context, the `-like` operator returns only the elements that match the specified wildcard expression:

```
> "PowerShell" -like "*owers*"
True
> "PowerShell", "Dog", "Cat", "Guinea Pig" -like "*owers*"
PowerShell
```

It is important to note that the array version of the operator does not return a Boolean value indicating whether any elements in the array match the expression, as the scalar version does.

- `-notlike`: Can be used to check whether a value does not match a wildcard expression when used with a scalar. If used in an array context, the `-notlike` operator returns only the elements that do not match the specified wildcard expression:

```
> "PowerShell" -notlike "*owers*"
False
```

```
> "PowerShell", "Dog", "Cat", "Guinea Pig" -notlike "*owers*"
Dog
Cat
Guinea Pig
```

- `-match`: Can be used to check whether a value matches a regular expression:

```
> "PowerShell scripting and automation for Cybersecurity" -match "shell\s*(\d)"
False
> "Cybersecurity scripting in PowerShell 7.3" -match "shell\s*(\d)"
True
```

- `-notmatch`: Can be used to check whether a value does not match a regular expression:

```
> "Cybersecurity scripting in PowerShell 7.3" -notmatch "^Cyb"
False
> "PowerShell scripting and automation for Cybersecurity" -notmatch "^Cyb"
True
```

Also refer to the official PowerShell documentation to read more about comparison operators: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators**.

Assignment operators

When working with variables, it is vital to understand assignment operators:

- `=`: Assigns a value:

```
> $a = 1; $a
1
```

- `+=`: Increases the value by the amount defined after the operator and stores the result in the initial variable:

```
> $a = 1; $a += 2; $a
3
```

- `-=`: Decreases the value by the amount defined after the operator and stores the result in the initial variable:

```
> $a
3
> $a -= 1; $a
2
```

- `*=`: Multiplies the value by the amount defined after the operator and stores the result in the initial variable:

```
> $a
2
> $a *= 3; $a
6
```

- `/=`: Divides the value by the amount defined after the operator and stores the result in the initial variable:

```
> $a
6
```

```
> $a /= 2; $a
3
```

- **`%=`**: Performs a modulo operation on the variable using the amount after the operator and stores the result in the initial variable:

```
> $a
3
> $a %= 2; $a
1
```

- **`++`**: Increases the variable by **`1`**:

```
> $a= 1; $a++; $a
2
```

- **`--`**: Decreases the variable by **`1`**:

```
> $a = 10; $a--; $a
9
```

Please refer to the official documentation to see more examples of how to use assignment operators: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_assignment_operators**.

Logical operators

If you work with multiple statements, you will need logical operators to add, compare, or exclude. In this section, you will find an overview of common logical operators in PowerShell:

- **`-and`**: Can be used to combine conditions. The defined action is triggered only if both conditions are met:

```
> $a = 1; $b = 2
> if (($a -eq 1) -and ($b -eq 2)) {Write-Host "Condition is true!"}
Condition is true!
```

- **`-or`**: If one of the defined conditions is met, the action is triggered:

```
> $a = 2; $b = 2
> if (($a -eq 1) -or ($b -eq 2)) {Write-Host "Condition is true!"}
Condition is true!
```

- **`-not`** or **`!`**: Can be used to negate a condition. The following example tests whether the folder specified using the **`$path`** variable is available. If it is missing, it will be created:

```
$path = $env:TEMP + "\TestDirectory"
if( -not (Test-Path -Path $path )) {
    New-Item -ItemType directory -Path $path
}
if (!(Test-Path -Path $path)) {
    New-Item -ItemType directory -Path $path
}
```

- **`-xor`**: Logical exclusive **`-or`**. Is **`True`** if *only one* statement is **`True`** (but returns **`False`** if both are **`True`**):

```
> $a = 1; $b = 2; ($a -eq 1) -xor ($b -eq 1)
True
> ($a -eq 1) -xor ($b -eq 2)
```

```
False
> ($a -eq 2) -xor ($b -eq 1)
False
```

Now that you have learned how to work with operators in PowerShell, let's have a look at control structures in our next section.

Please also refer to the **about_operators** documentation to learn more about PowerShell operators in general: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators**.

Control structures

A control structure is some kind of programmatic logic that assesses conditions and variables and decides which defined action will be taken if a certain condition is met.

Use the operators that we learned about in the last section to define the conditions, which will be assessed using the control structures introduced in this section.

Conditions

If you want to select which action is performed if a certain condition is met, you can use one of the following selection control structures: either an **if/elseif/else** construct or the **switch** statement.

If/elseif/else

if, **elseif**, and **else** can be used to check whether a certain condition is **True** and run an action if the condition is fulfilled:

```
if (<condition>)
{
    <action>
}
elseif (<condition 2>)
{
    <action 2>
}
...
else
{
    <action 3>
}
```

You can use the **if** statement to check whether a condition is **True**:

```
> if (1+2 -eq 3) { Write-Host "Good job!" }
 Good job!
> if (1+2 -eq 5) { Write-Host "Something is terribly wrong!" }
# returns no Output
```

You can also check whether one of several conditions is **`True`** by using **`elseif`**. The action of the first condition that is met will be executed:

```
$color = "green"
if ($color -eq "blue") {
     Write-Host "The color is blue!"
}
elseif ($color -eq "green"){
     Write-Host "The color is green!"
}
# returns: The color is green!
```

In this example, the control structure checks whether one of the specified conditions is met (either **`$color -eq "blue"`** or **`$color -eq "green"`**). If **`$color`** would be **`red`**, no action would be performed.

But since **`$color`** is **`green`**, the **`elseif`** condition is **`True`** and the **`The color is green!`** string will be written to the console.

If you want to specify an action that will be triggered if none of the specified conditions are met, you can use **`else`**. If no condition from **`if`** or **`elseif`** is met, the action specified in the **`else`** block will be executed:

```
$color = "red"
if ($color -eq "blue") {
     Write-Host "The color is blue!"
}
elseif ($color -eq "green"){
     Write-Host "The color is green!"
}
else {
     Write-Host "That is also a very beautiful color!"
}
# returns: That is also a very beautiful color!
```

In this example, we check whether **`$color`** is either **`blue`** or **`green`**. But since **`$color`** is **`"red"`**, none of the defined conditions are **`True`**, and therefore the code defined in the **`else`** block will be executed, which writes **`That is also a very beautiful color!`** to the output.

Switch

Sometimes, it can happen that you want to check one variable against a long list of values.

To solve this problem, you could – of course – create a long and complicated list of **`if`**, **`elseif`**, ..., **`elseif`**, and **`else`** statements.

But instead, you can use the more elegant **`switch`** statement to test a value against a list of predefined values and react accordingly:

```
switch (<value to test>) {
     <condition 1> {<action 1>}
     <condition 2> {<action 2>}
     <condition 3> {<action 3>}
     ...
```

```
    default {}
}
```

Here is an example:

```
$color = Read-Host "What is your favorite color?"
switch ($color) {
    "blue"   { Write-Host "I'm BLUE, Da ba dee da ba di..." }
    "yellow" { Write-Host "YELLOW is the color of my true love's hair." }
    "red"    { Write-Host "Roxanne, you don't have to put on the RED light..." }
    "purple" { Write-Host "PURPLE rain, purple rain!" }
    "black" { Write-Host "Lady in BLACK... she came to me one morning, one lonely Sunday morning.
    default  { Write-Host "The color is not in this list." }
}
```

In this example, the user is prompted to enter a value: **`What is your favorite color?`**.

Depending on what the user enters, a different output will be shown: if **`purple`** is entered, a line from a famous Prince song, *Purple Rain*, will be displayed. If **`red`** is entered, a line of the Police song *Roxanne* is cited.

But if **`green`** is entered, the **`default`** output will be shown, as there's no option for the **`green`** value defined and the message **`The color is not in this list`** will be displayed.

In addition to using the **`switch`** statement to evaluate simple conditions based on the value of a variable or expression, PowerShell also supports **more advanced modes**. These modes allow you to use regular expressions, process the contents of files, and more.

For example, you can use the **`-Regex`** parameter to use a regular expression to match against the input, like this:

```
switch -Regex ($userInput) {
    "^[A-Z]" { "User input starts with a letter." }
    "^[0-9]" { "User input starts with a number." }
    default { "User input doesn't start with a letter or number." }
}
```

If **`$userInput`** was defined as **`"Hello World!"`**, then **`"User input starts with a letter."`** would be written to the output. If **`$userInput`** started with a number (for example, **`"1337"`**), the output would be **`"User input starts with a number."`**. And if **`$userInput`** started with a different character, (for example, **`"!"`**), then the **`default`** condition would be met and **`"User input doesn't start with a letter or number."`** would be written to the output.

You can also use the **`-File`** parameter to process the contents of a file with the **`switch`** statement. The **`-Wildcard`** parameter enables you to use the wildcard logic with **`switch`**:

```
$path = $env:TEMP + "\example.txt"
switch -Wildcard -File $path {
```

```
    "*Error*" { Write-Host "Error was found!: $_" }
}
```

In this example, we're using the **`switch`** statement to process the contents of a file named **`"example.txt"`**. We're looking for the **`"*Error*"`** pattern within the file, and then taking an action based on whether that pattern was found. If the specified file contains the pattern, **`"Error was found!:"`** will be written to the output, followed by the line that contained the error. It's important to note that the wildcard pattern is processed line by line and not for the entire file, so there will be an **`"Error was found!: "`** line written to the output for every line in the file that contained the **`"*Error*"`** pattern.

Loops and iterations

If you want to run an action over and over again until a certain condition is met, you can do that using loops. A loop will continue to execute as long as the specified condition is **`True`** unless it is terminated with a loop-breaking statement such as **`break`**. Depending on the loop construct used, the loop may execute at least once, or may not execute at all if the condition is initially **`False`**.

In this section, you will find an overview of how to work with loops.

ForEach-Object

`ForEach-Object` accepts a list or an array of items and allows you to perform an action against each of them. **`ForEach-Object`** is best used when you use the pipeline to pipe objects to **`ForEach-Object`**.

As an example, if you want to process all files that are in a folder, you can use **`Foreach-Object`**. **`$_`** contains the value of every single item of each iteration:

```
> $path = $env:TEMP + "\baselines"
> Get-ChildItem -Path $path | ForEach-Object {Write-Host $_}
Office365-ProPlus-Sept2019-FINAL.zip
Windows 10 Version 1507 Security Baseline.zip
Windows 10 Version 1607 and Windows Server 2016 Security Baseline.zip
Windows 10 Version 1803 Security Baseline.zip
Windows 10 Version 1809 and Windows Server 2019 Security Baseline.zip
Windows 10 Version 1903 and Windows Server Version 1903 Security Baseline - Sept2019Update.zip
Windows 10 Version 1909 and Windows Server Version 1909 Security Baseline.zip
Windows 10 Version 2004 and Windows Server Version 2004 Security Baseline.zip
Windows Server 2012 R2 Security Baseline.zip
```

If you want to perform specific actions before processing each item in the pipeline or after processing all the items, you can use the **`-Begin`** and **`-End`** advanced parameters with the **`ForEach-Object`** cmdlet: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/foreach-object**.

Additionally, you can use the **`-Process`** parameter to specify the script block that is run for each item in the pipeline.

Foreach

To iterate through a collection of items in PowerShell, you can use the **`Foreach-Object`** *cmdlet*, the **`foreach`** *statement*, or the **`foreach`** *method*. The **`Foreach-Object`** *cmdlet* accepts pipeline objects, making it a useful tool for working with object-oriented data. The **`foreach`** *method* and the **`foreach`** *statement* are very similar to **`Foreach-Object`** but they do not accept pipeline objects. You will get error messages if you try to use it in the same way as **`Foreach-Object`**.

The **`foreach`** *statement* loads all items into a collection before they are processed, making it quicker but consuming more memory than **`ForEach-Object`**.

The following example shows how to use the **`foreach`** *statement*:

```
$path = $env:TEMP + "\baselines"
$items = Get-ChildItem -Path $path
foreach ($file in $items) {
     Write-Host $file
}
```

In this example, the **`$path`** path is examined similarly as in our example before. But in this case, it uses a **`foreach`** *statement* to iterate through each item in the **`$items`** array, assigning the current item to the **`$file`** variable on each iteration. The **`$file`** variable is defined by the author of the script – every other variable name can be added here and, of course, processed. For each item, it outputs the value of **`$file`** to the console using the **`Write-Host`** cmdlet.

You can use the **`.foreach({})`** *method* to iterate through a collection of items. Here's an example of how to use it:

```
$path = $env:TEMP + "\baselines"
$items = Get-ChildItem -Path $path
$items.foreach({
    Write-Host "Current item: $_"
})
```

In this example, **`$path`** is examined; for each file in that folder, the filename will be written to the command line. The **`.foreach({})`** *method* is used to iterate through each item in the **`$items`** collection and write a message to the console that includes the item's name. The **`$_`** variable is used to reference the current item being iterated over. So, for each item in the **`$items`** collection, the script will output a message such as **`"Current item: filename"`**.

while

`while` does something (**`<actions>`**) as long as the defined *condition* is fulfilled:

```
while ( <condition> ){ <actions> }
```

In this example, user input is read, and as long as the user doesn't type in **`quit`**, the **`while`** loop still runs:

```
while(($input = Read-Host -Prompt "Choose a command (type in 'help' for an overview)") -ne "quit")
    switch ($input) {
        "hello" {Write-Host "Hello World!"}
        "color" {Write-Host "What's your favorite color?"}
        "help" {Write-Host "Options: 'hello', 'color', 'help' 'quit'"}
    }
}
```

In this example, if the user types in either **`hello`**, **`color`**, or **`help`**, different output options will be shown, but the program still continues, as the condition for the **`while`** statement is not fulfilled.

Once the user types in **`quit`**, the program will be terminated, as the condition is fulfilled.

for

This defines the initializing statement, a condition, and loops through until the defined condition is not fulfilled anymore:

```
for (<initializing statement>; <condition>; <repeat>)
{
    <actions>
}
```

If you need iterating values, **`for`** is a great solution:

```
> for ($i=1; $i -le 5; $i++) {Write-Host "i: $i"}
i: 1
i: 2
i: 3
i: 4
i: 5
```

In this example, **`$i=1`** is the starting condition, and in every iteration, **`$i`** is increased by **`1`**, using the **`$i++`** statement. As long as **`$i`** is smaller than or equal to **`5`** – that is, **`($i -le 5)`** – the loop continues and writes **`$i`** to the output.

do-until/do-while

Compared to other loops, **`do-until`** or **`do-while`** already starts running the defined commands and then checks whether the condition is still met or not met:

```
do{
     <action>
}
<while/until><condition>
```

Although **`do-until`** and **`do-while`** have the same syntax, they differ in how the condition is treated.

do-while runs as long as the condition is **True** and stops as soon as the condition is not met anymore. **do-until** runs only as long as the condition is *not* met: it ends when the condition is met.

break

break can be used to exit the loop (for example, **for**/**foreach**/**foreach-object**/...):

```
> for ($i=1; $i -le 10; $i++) {
    Write-Host "i: $i"
    if ($i -eq 3) {break}
}
i: 1
i: 2
i: 3
```

Consult the official documentation to learn more about the advanced usage of **break**: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break**.

continue

The **continue** statement is used to skip the current iteration of a loop and move to the next one. It does not affect the loop's condition, which will be re-evaluated at the beginning of the next iteration:

```
> for ($i=1; $i -le 10; $i++) {
    if (($i % 2) -ne 0) {continue}
    Write-Host "i: $i"
}
i: 2
i: 4
i: 6
i: 8
i: 10
```

In this example, we use the modulus (**%**) operator to calculate whether a division by **2** returns a remainder. If the remainder of **$i % 2** is non-zero, then the condition returns **True**, and **continue** is triggered.

This behavior causes **$i** to be only written to the console if no remainder is returned.

DID YOU KNOW?

The preceding example demonstrates that every time the remainder returned is not 0, the current iteration is skipped. This code could also be simplified by writing the following:

for ($i=1; $i -le 10; $i++) {

if ($i % 2){ continue }

Write-Host "i: $i"

```
}
```

You can use control structures not only to solve a single instance but also to solve problems by combining multiple control structures to build complex logic.

After reading this section, you should have a basic knowledge of what control structures exist and how to use them.

Naming conventions

Cmdlets and functions both follow the schema *verb-noun*, such as **`Get-Help`** or **`Stop-Process`**. So, if you write your own functions or cmdlets, make sure to follow the name guidelines and recommendations.

Microsoft has released a list of approved verbs. Although it is not technically enforced to use approved verbs, it is strongly recommended to do so in order to comply with PowerShell best practices and avoid conflicts with automatic variables and reserved words. Additionally, using approved verbs is required when publishing PowerShell modules to the PowerShell Gallery, as it will trigger a warning message if non-approved verbs are used. Here is the link for the approved verbs:

https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands

Finding the approved verbs

If you are in the process of writing your code and quickly want to check which approved verbs exist, you can leverage the **`Get-Verb`** command.

If you want to sort the list of available verbs, you can pipe the output to **`Sort-Object`**. By default, the verbs are sorted into traditional categories of use, such as **`Common`**, **`Data`**, and **`Lifecycle`**. However, you can also sort them alphabetically by name by specifying the **`Name`** property with the **`Sort-Object`** command. Use the following command to sort the output of **`Get-Verb`** by the name **`Verb`**:

```
Get-Verb | Sort-Object Verb
```

You can also use wildcards to prefilter the list:

```
> Get-Verb re*
Verb     Group
----     -----
Redo     Common
Remove   Common
Rename   Common
Reset    Common
Resize   Common
Restore  Data
Register Lifecycle
Request  Lifecycle
```

```
Restart  Lifecycle
Resume   Lifecycle
Repair   Diagnostic
Resolve  Diagnostic
Read     Communications
Receive  Communications
Revoke   Security
```

If you just want to get all approved verbs from a certain group (in this case, **`Security`**), you can filter **`Group`** using **`Where-Object`**:

```
> Get-Verb | Where-Object Group -eq Security
Verb      Group
----      -----
Block     Security
Grant     Security
Protect   Security
Revoke    Security
Unblock   Security
Unprotect Security
```

Although naming conventions are not enforced in PowerShell, they should be respected nevertheless. Microsoft also strongly encourages following those guidelines when writing your cmdlets to ensure that users have a consistent user experience.

Please also have a look at the development guidelines when writing your own functions and cmdlets: https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/strongly-encouraged-development-guidelines.

PowerShell profiles

PowerShell profiles are configuration files that allow you to personalize your PowerShell environment. These profiles can be used to customize the behavior and environment of PowerShell sessions. They are scripts that are executed when a PowerShell session is started, allowing users to set variables, define functions, create aliases, and more.

Any variables, functions, or aliases defined in the appropriate PowerShell profile will be loaded every time a PowerShell session is started. This means you can have a consistent and personalized PowerShell environment across all your sessions.

There are several different types of profiles and more than one can be processed by PowerShell. PowerShell profiles are stored as plain text files on your system, and there are several types of profiles available:

- **All Users, All Hosts** (**`$profile.AllUsersAllHosts`**): This profile applies to all users for all PowerShell hosts.
- **All Users, Current Host** (**`$profile.AllUsersCurrentHost`**): This profile applies to all users for the current PowerShell host.
- **Current User, All Hosts** (**`$profile.CurrentUserAllHosts`**): This profile applies to the current user for all PowerShell hosts.

- **Current User, Current Host** (`$profile.CurrentUserCurrentHost`): This profile applies only to the current user and the current PowerShell host.

A **PowerShell host** is an application that hosts the PowerShell engine. Examples of PowerShell hosts include the Windows PowerShell console, the PowerShell **Integrated Scripting Environment** (**ISE**), and the PowerShell terminal in Visual Studio Code.

The location of your PowerShell profile(s) depends on your system and configuration, but you can easily find out where they are stored by running the following command in PowerShell:

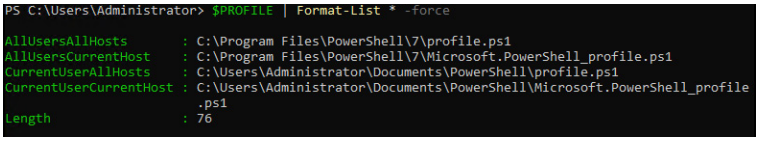

Figure 2.4 – Finding out the location of the local PowerShell profile(s)

It is important to note that there are also more profile paths available, including those used by the system and not just by individual users (which would be included in the `AllUsers` profile):

- Applies to local shells and all users: `%windir%\system32\WindowsPowerShell\v1.0\profile.ps1`
- Applies to all shells and all users: `%windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1`
- Applies to all local ISE shells and all users: `%windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1`

This profile is loaded when using the PowerShell ISE and can be viewed by running the `$profile | fl * -force` command within the ISE

- Applies to current user ISE shells on the local host: `%UserProfile%\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1`

For example, in Windows PowerShell, there are profiles for `AllUsers` and `AllHosts`, which apply to all users and all PowerShell hosts on a system. In PowerShell Core, there are profiles for `AllUsers` and `AllHosts` as well, but they do not load the Windows PowerShell profiles from the `system32` directory by default. It's also worth noting that while PowerShell Core supports loading Windows PowerShell profiles, the reverse is not true.

To access the file path of one particular profile, such as the one for `CurrentUserCurrentHost`, you can use the variable that is defined in `$profile.CurrentUserCurrentHost`:

```
> $profile.CurrentUserCurrentHost
C:\Users\pssecuser\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
```

Use the following code snippet to check whether the file already exists; if it does not yet, the file is created:

```
if ( !( Test-Path $profile.CurrentUserCurrentHost ) ) {
    New-Item -ItemType File -Path $profile.CurrentUserCurrentHost
}
```

Finally, add the commands, functions, or aliases to the user profile:

```
> Add-Content -Path $profile -Value “New-Alias -Name Get-Ip -Value ‘ipconfig.exe’”
```

In addition to customizing your PowerShell environment, profiles are also a crucial aspect of PowerShell security. By modifying your profiles, you can set policies and restrictions to enforce security best practices, such as preventing the execution of unsigned scripts or setting execution policies. But also, adversaries can use PowerShell profiles to their advantage – for example, to establish persistence.

Understanding PSDrives in PowerShell

PowerShell includes a feature called **PowerShell drives** (**PSDrives**). PSDrives in PowerShell are similar to filesystem drives in Windows, but instead of accessing files and folders, you use PSDrives to access a variety of data stores. These data stores can include directories, registry keys, and other data sources, which can be accessed through a consistent and familiar interface.

PSDrives are powered by **PSProviders**, which are the underlying components that provide access to data stores. PSProviders are similar to drivers in Windows, which allow access to different hardware devices. In the case of PowerShell, PSProviders allow you to access different data stores in a uniform way, using the same set of cmdlets and syntax.

For example, the **`Env:\`** PSDrive is a built-in PowerShell drive that provides access to environment variables. To retrieve all environment variables that have the **`path`** string in their name, you can use the **`Get-ChildItem`** cmdlet with the **`Env:\`** PSDrive:

```
> Get-ChildItem Env:\*path*
```

To access a PSDrive, you use a special prefix in the path. For example, to access the filesystem drive, you use the prefix **`C:`**, and to access the registry drive, you use the prefix **`HKLM:`**. In the case of the **`Env:\`** PSDrive, the prefix is **`Env:`**, which allows you to access environment variables as if they were files or folders.

There are several built-in PSDrives in PowerShell, including the following:

- **`Alias`**: Provides access to PowerShell aliases
- **`Environment`**: Provides access to environment variables
- **`Function`**: Provides access to PowerShell functions
- **`Variable`**: Provides access to PowerShell variables
- **`Cert`**: Provides access to certificates in the Windows certificate store

- `Cert:\CurrentUser`: Provides access to certificates in the current user's certificate store
- `Cert:\LocalMachine`: Provides access to certificates in the local machine's certificate store
- `WSMan`: Provides access to **Windows Remote Management** (**WinRM**) configuration data
- `C:` and `D:` (*and other drive letters*): Used to access the filesystem, just like in Windows Explorer
- `HKCU`: Provides access to the `HKEY_CURRENT_USER` registry hive
- `HKLM`: Provides access to the `HKEY_LOCAL_MACHINE` registry hive

Making your code reusable

In this section, we will explore the concept of making your code reusable in PowerShell. Reusability is an important aspect of coding that allows you to create a function, cmdlet, or module once and use it multiple times without having to rewrite the same code again and again. Through this, you can save time and effort in the long run.

We will start by discussing cmdlets, followed by functions and aliases, and finally, we will explore PowerShell modules, which are collections of PowerShell commands and functions that can be easily shared and installed on other systems, which is a great way to package and distribute your reusable code.

Cmdlets

A cmdlet (pronounced as *commandlet*) is a type of PowerShell command that performs a specific task and can be written in C# or in another .NET language. This includes advanced functions, which are also considered cmdlets but have more advanced features than regular functions.

`Get-Command` can help you to differentiate cmdlets from functions. Additionally, you can also see the version and the provider:

```
> Get-Command new-item
CommandType   Name      Version    Source
-----------   ----      -------    ------
Cmdlet        New-Item  3.1.0.0    Microsoft.PowerShell.Management
```

To find out all cmdlets that are currently installed on the machine you are using, you can leverage `Get-Command` with the `CommandType` parameter:

```
Get-Command -CommandType Cmdlet
```

If you want to dig deeper into cmdlets, I recommend reviewing the official PowerShell documentation. Microsoft has published a lot of advice, as well as recommendations and guidelines:

- **https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/cmdlet-overview**
- **https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/windows-powershell-**

cmdlet-concepts

Functions

Functions are a collection of PowerShell commands that should be run following a certain logic.

As with other programming and scripting languages, if you are typing in the same commands over and over again, and if you find yourself modifying the same one-liners for different scenarios, it is definitely time to create a function.

When you choose a name, make sure it follows the verb-noun naming convention and only uses approved verbs. Read more about approved verbs and naming conventions in the *Naming conventions* section covered earlier in this chapter.

This skeleton function using pseudocode should demonstrate the basic structure of a function:

```
function Verb-Noun {
<#
        <Optional help text>
#>
param (
    [data_type]$Parameter
)
<...Code: Function Logic...>
}
```

Once the function is loaded into the session, it needs to be called so that it will be executed:

```
Verb-Noun -Parameter "test"
```

You can find a demo function with demo help that simply writes the output **Hello World!** and accepts a parameter to generate additional output, as well as the calling of it on GitHub:

https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter02/Write-HelloWorld.ps1

Parameters

A function does not necessarily need to support parameters, but if you want to process input within the function, parameters are required:

```
function Invoke-Greeting {
    param (
        [string]$Name
    )
    Write-Output "Hello $Name!"
}
```

In this example, the **`Invoke-Greeting`** function provides the possibility to supply the **`$Name`** parameter, while specifying the data type as **`[string]`** will attempt to convert any input to a *string*, allowing for flexibility in the parameter input. You can also use other data types (for example, **`int`**, **`boolean`**, and so on) depending on your use case.

If the parameter is specified, the provided value is stored in the **`$Name`** variable and can be used within the function:

```
> Invoke-Greeting -Name "Miriam"
Hello Miriam!
```

If the parameter is not specified, it will be replaced by **`$null`** (which is **`""`**/*nothing*):

```
> Invoke-Greeting
Hello !
```

In this case, the **`$Name`** parameter is not mandatory, so it does not have to be specified to run the function.

Adding parameters enables you to cover many of your use case's complex scenarios. You might have already seen functions that allow only some type of input or that require a certain parameter – functions that will not be run until the user confirms and functions that provide the possibility to run them verbosely.

Let's explore how these behaviors can be configured in our next sections about **`cmdletbinding`**, **`SupportsShouldProcess`**, input validation, and mandatory parameters.

cmdletbinding

`cmdletbinding` is a feature in PowerShell that allows you to add common parameters (such as **`-Verbose`**, **`-Debug`**, or **`-ErrorAction`**) to your functions and cmdlets without defining them yourself. This can make your code more consistent with other PowerShell commands and easier to use for users.

One way to use **`cmdletbinding`** is to declare a parameter as mandatory, positional, or in a parameter set, which can automatically turn your function into a cmdlet with additional common parameters. For example, if you want to make the **`-Name`** parameter mandatory in your function, you can add **`[Parameter(Mandatory)]`** before the parameter definition, like this:

```
function Invoke-Greeting {
    [cmdletbinding()]
    param (
        [Parameter(Mandatory)]
        $Name
    )
    Write-Output "Hello $Name!"
}
```

This will automatically add the **[<CommonParameters >]** section to the output of **Get-Command**, and you will see all the common parameters that are also available in many other cmdlets, such as **Verbose**, **Debug**, **ErrorAction**, and others.

To learn more about **cmdletbinding** and its functionality, check out the following link: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_cmdletbindingattribute**.

SupportsShouldProcess

If a function makes changes, you can use **SupportsShouldProcess** to add an additional layer of protection to your function. By adding **[CmdletBinding(SupportsShouldProcess)]**, you can enable the **-WhatIf** and **-Confirm** parameters in your function, which help users understand the effect of their actions before executing the function. To use **SupportsShouldProcess** effectively, you will also need to call **ShouldProcess()** for each item being processed. Here's an example of what your code could look like:

```
function Invoke-Greeting {
    [CmdletBinding(SupportsShouldProcess)]
    param (
        $Name
    )
    foreach ($item in $Name) {
        if ($PSCmdlet.ShouldProcess($item)) {
            Write-Output "Hello $item!"
        }
    }
}
```

With this code, the function can be executed with the **-Confirm** parameter to prompt the user for confirmation before processing each item, or with the **-WhatIf** parameter to display a list of changes that would be made without actually processing the items.

```
> Get-Command -Name Invoke-Greeting -Syntax
Invoke-Greeting [[-Name] <Object>] [-WhatIf] [-Confirm] [<CommonParameters>]
```

Once you have added **SupportsShouldProcess** to your function, you can also see that the syntax has changed, by using **Get-Command** as shown in the preceding example.

Accepting input via the pipeline

It is also possible to configure parameters to accept user input to use it in our code. In addition to accepting input from the user, we can also accept input from the pipeline. This can be done in two ways: by value or by property name.

When accepting input by value, we receive the entire object passed through the pipeline. We can then use the parameter in our function to filter or manipulate the object.

When accepting input by property name, we receive only the specified property of the object passed through the pipeline. This can be useful when we only need to work with a specific property of the object.

To configure a function to accept input by value, we can use **`ValueFromPipeline`**; to accept input by property name use **`ValueFromPipelineByPropertyName`**. Of course, both can be combined with each other and with other parameter options as well, such as **`Mandatory`**.

The following example shows the **`Invoke-Greeting`** function, which accepts input both by value and property name for its mandatory **`$Name`** parameter:

```
function Invoke-Greeting {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline, ValueFromPipelineByPropertyName)]
        [string]$Name
    )
    process {
        Write-Output "Hello $Name!"
    }
}
```

You can now pass input by value to this function, as shown in the following example:

```
> "Alice","Bob" | Invoke-Greeting
Hello Alice!
Hello Bob!
```

But it also works to pass input by property name, as the following code snippet demonstrates:

```
> [pscustomobject]@{Name = "Miriam"} | Invoke-Greeting
Hello Miriam!
```

If you want to dive deeper into accepting input from the pipeline and how to troubleshoot issues, you may refer to the following resources:

- *PowerShell Basics for Security Professionals Part 6 – Pipeline* by Carlos Perez: **https://youtube.com/watch?v=P3ST3lat9bs**
- *About Pipelines*: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipelines**

As this book focuses on PowerShell security and not on expert function creation, it can barely scratch the surface of advanced functions. So, if you are interested in learning more about advanced functions and parameters, I have added some links in the *Further reading* section at the end of this chapter.

Comment-based help

Writing comment-based help for your functions is crucial; others might reuse your function or if you want to adjust or reuse the function yourself some months after you wrote it, having good comment-based help will simplify the usage:

```
<#
.SYNOPSIS
<Describe the function shortly.>
.DESCRIPTION
<More detailed description of the function.>
.PARAMETER Name
<Add a section to describe each parameter, if your function has one or more parameters.>
.EXAMPLE
<Example how to call the funtion>
<Describes what happens if the example call is run.>
#>
```

Please also have a look at the **`Write-HelloWorld.ps1`** demo script on GitHub to see an example: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter02/Write-HelloWorld.ps1**.

Error handling

If you are not sure whether your command will succeed, use **`try`** and **`catch`**:

```
try {
    New-PSSession -ComputerName $Computer -ErrorAction Stop
}
catch {
    Write-Warning -Message "Couldn't connect to Computer: $Computer"
}
```

Setting **`ErrorAction`** to **`Stop`** will treat the error as a terminating error. As only terminating errors are caught, the action defined in the **`catch`** block is triggered.

If **`ErrorAction`** is not defined and if no terminating error is triggered, the **`catch`** block will be ignored.

The difference between cmdlets and script cmdlets (advanced functions)

When I heard for the first time about cmdlets and advanced functions, I was like *Okay great, but what's the difference? They both sound pretty alike*.

One significant difference is that cmdlets can be written in a .NET language such as C# and reside within a compiled binary. Script cmdlets, also known as advanced functions, are similar to cmdlets, but they are written in PowerShell script rather than a .NET language. Script cmdlets are a way to create custom cmdlets using PowerShell script instead of compiling code in a .NET language.

One advantage of script cmdlets is that they can be easily modified and debugged without requiring compilation, making them more accessible to users who may not be comfortable with .NET languages. Additionally, script cmdlets can be distributed and shared just like compiled cmdlets.

For software vendors and developers, it is easier to package compiled cmdlets than to package libraries of functions and scripts, as well as to write and package help files.

However, it is just a matter of preference what you want to use – if you prefer writing your functions in C# or other .NET-based languages, cmdlets might be your preferred choice; if you prefer using PowerShell only, you might want to create PowerShell functions.

Aliases

An alias is some kind of a nickname for a PowerShell command, an alternate name. You can set aliases to make your daily work easier – for example, if you are repeatedly working with the same long and complicated command, setting an alias and using it instead will ease your daily work.

For example, one of the most used aliases is the famous **`cd`** command, which administrators use to change the directory on the command line. But **`cd`** is only an alias for the **`Set-Location`** cmdlet:

```
PS C:\> cd 'C:\tmp\PSSec\'
PS C:\tmp\PS Sec>
PS C:\> Set-Location 'C:\tmp\PSSec\'
PS C:\tmp\PS Sec>
```

To see all available cmdlets that have the word **`Alias`** in their name, you can leverage **`Get-Command`**:

Figure 2.5 – Getting all available cmdlets that have the word Alias in their name

Next, let's have a closer look at how to work with aliases, using the **`Get-Alias`**, **`New-Alias`**, **`Set-Alias`**, **`Export-Alias`**, and **`Import-Alias`** cmdlets.

Get-Alias

To see all aliases that are currently configured on the computer you are working on, use the **`Get-Alias`** cmdlet:

Figure 2.6 – Output of the Get-Alias command

You can either use **`Get-Alias`** to inspect the entire list of aliases that are available, or you can check whether a specific alias exists using the **`-Name`** parameter.

New-Alias

You can use **`New-Alias`** to create a new alias within the current PowerShell session:

```
> New-Alias -Name Get-Ip -Value ipconfig
> Get-Ip
Windows IP Configuration
Ethernet adapter Ethernet:
   Connection-specific DNS Suffix  . : mshome.net
   IPv4 Address. . . . . . . . . . . : 10.10.1.10
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.10.1.1
```

This alias is not set permanently, so once you exit the session, the alias will not be available anymore.

If you want to use aliases multiple times in multiple sessions, you can either export them and import them in every new session or you can configure them to be permanently set for every new PowerShell session by using the PowerShell profile.

If you want to add parameters to the command that your alias runs, you can create a function and use **`New-Alias`** to link the new function to your existing command.

Set-Alias

`Set-Alias` can be used to either create or change an alias.

So if you want to change, for example, the content of the formerly created **`Get-Ip`** alias to **`Get-NetIPAddress`**, you would run the following command:

```
> Set-Alias -Name Get-Ip -Value Get-NetIPAddress
```

Export-Alias

Export one or more aliases with **`Export-Alias`** – either as a **`.csv`** file or as a script:

```
Export-Alias -Path "alias.csv"
```

Using this command, we first export all aliases to a **`.csv`** file:

```
Export-Alias -Path "alias.ps1" -As Script
```

The **`-As Script`** parameter allows you to execute all currently available aliases as a script that can be executed:

```
Export-Alias -Path "alias.ps1" -Name Get-Ip -As Script
```

If you plan to re-import the aliases later, it's important to be aware that executing the script without re-importing the function may cause issues. Therefore, make sure to also import the script on the new system on which you plan to import the alias.

Of course, it is also possible to only export a single alias by specifying its **`-Name`** parameter, in the last example.

alias.csv

The **`alias.csv`** file that we created using the **`Export-Alias`** command can now be reused to create or import all aliases of this session in another session:

```
# Alias File
# Exported by : PSSec
# Date/Time : Sunday, July 9, 2023 1:39:50 PM
# Computer : PSSEC-PC
"foreach","ForEach-Object","","ReadOnly, AllScope"
"%","ForEach-Object","","ReadOnly, AllScope"
"where","Where-Object","","ReadOnly, AllScope"
"?","Where-Object","","ReadOnly, AllScope"
"ac","Add-Content","","ReadOnly, AllScope"
"clc","Clear-Content","","ReadOnly, AllScope"
...
"stz","Set-TimeZone","","None"
"Get-Ip","Get-NetIPAddress","","None"
```

alias.ps1

If you export your aliases using the **`-As Script`** option (as in the example from earlier), an executable **`.ps1`** file (**`alias.ps1`**) is created.

You can now use the file to set your aliases automatically whenever you run the **`.ps1`** script, or you can use the code to edit your profile file (see **`New-Alias`**) to configure permanent aliases:

```
# Alias File
# Exported by : PSSec
# Date/Time : Sunday, July 9, 2023 1:34:31 PM
# Computer : PSSEC-PC
set-alias -Name:"Get-Ip" -Value:"Get-NetIPAddress" -Description:"" -Option:"None"
```

If you use functions to define aliases, make sure to also save those functions and execute them in the session in which you want to import your aliases.

Import-Alias

You can use **`Import-Alias`** to import aliases that were exported as **`.csv`**:

```
> Set-Alias -Name Get-Ip -Value Get-Iponfig
> Export-Alias -Name Get-Ip -Path Get-Ip_alias.csv
```

Import the file to make the alias available in your current session:

```
> Import-Alias -Path .\Get-Ip_alias.csv
> Get-Ip
Windows IP Configuration
Ethernet adapter Ethernet:
   Connection-specific DNS Suffix  . : mshome.net
   IPv4 Address. . . . . . . . . . . : 10.10.1.10
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.10.1.1
```

Further information on aliases can be found at the following link: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_aliases**.

Modules

Modules are a collection of PowerShell commands and functions that can be easily shipped and installed on other systems. They are a great way to enrich your sessions with other functionalities.

FIND MODULE-RELATED CMDLETS

To find module-related cmdlets, leverage **`Get-Command`** *and have a look at their help pages and the official documentation to understand their function:*

```
Get-Command -Name "*Module*"
```

All modules that are installed on the system can be found in one of the **`PSModulePath`** folders, which are part of the **`Env:\`** PSDrive:

```
> Get-Item -Path Env:\PSModulePath
Name           Value
----           -----
PSModulePath   C:\Users\PSSec\Documents\WindowsPowerShell\Modules;
               C:\Program Files\WindowsPowerShell\Modules;
               C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

Query the content with **`Env:\PSModulePath`** to find out which paths were set on your system.

Working with modules

To use a module efficiently, the following sections will help you to make the module available, to find out how to work with it, and to finally remove or unload it.

Finding and installing modules

To search for a certain module in a repository, you can leverage **`Find-Module -Name <modulename>`**. It queries the repositories that are configured on your operating system:

```
> Find-Module -Name EventList
Version    Name       Repository           Description
-------    ----       ----------           -----------
2.0.1      EventList  PSGallery            EventList - The Event Analyzer. This tool helps you to
```

Once you have found the desired module, you can download and install it to your local system using **`Install-Module`**:

```
> Install-Module <modulename>
```

If you have already installed a module for which a newer version exists, update it with **`Update-Module`**:

```
> Update-Module <modulename> -Force
```

To see which repositories are available on your system, use the following:

```
> Get-PSRepository
```

One of the most commonly used repositories is the **PowerShell Gallery** (shown as **`PSGallery`** in the previous example).

The PowerShell Gallery

The PowerShell Gallery is the central repository for PowerShell content: **https://www.powershellgallery.com/**. In this repository, you'll find thousands of helpful modules, scripts, and **Desired State Configuration (DSC)** resources.

To leverage the PowerShell Gallery and to install modules directly from the repository, **`NuGet`** and **`PowerShellGet`** need to be installed.

If you haven't installed the required packages, when you try to install a module for the first time from the PowerShell Gallery, you will be prompted to install it:

Figure 2.7 – Installing a module from the PowerShell Gallery using Windows PowerShell

As you can see in the preceding screenshot, you will not only be prompted to install the module itself but also the NuGet provider if you are installing modules from the PowerShell Gallery for the first time.

If you are using PowerShell Core, both **`NuGet`** and **`PowerShellGet`** are usually already preinstalled:

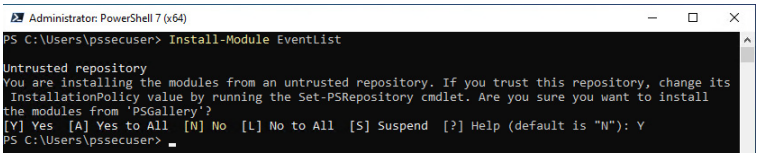

Figure 2.8 – Installing a module from the PowerShell Gallery using PowerShell Core

CONFIGURE POWERSHELL GALLERY AS A TRUSTED REPOSITORY

When you install modules from the PowerShell Gallery, you may receive a warning that the repository is not trusted. This warning is displayed to ensure that you are aware that you are installing code from an external source that has not been verified by Microsoft. The warning is intended to protect you from potentially malicious code that could harm your system.

To avoid the warning, you can configure the repository as a trusted repository. By doing this, you are indicating that you trust the source and that you accept the potential risks associated with installing code from it. To configure a repository as a trusted repository, you can use the following code snippet: **`Set-PSRepository -Name 'PSGallery' -InstallationPolicy Trusted`**.

By configuring the repository as a trusted repository, you are indicating that you trust the code provided by that repository and that you are willing to take responsibility for any risks associated with using it.

Working with modules

To find out which modules are already available in the current session, you can use **`Get-Module`**:

```
> Get-Module
```

To see which modules are available to import, including those that come pre-installed with Windows, you can use the **`ListAvailable`** parameter with the **`Get-Module`** cmdlet. This will display a list of all available modules on the computer, including their version numbers, descriptions, and other information:

```
> Get-Module -ListAvailable
```

Find out which commands are available by using **`Get-Command`**:

```
> Get-Command -Module <modulename>
```

And if you want to know more about the usage of a command that is available in a module, you can use **`Get-Help`**. You can see how important it is to write proper help pages for your function:

Figure 2.9 – Getting the help pages of a command

If you have, for example, an old version loaded in your current session and you want to unload it, **`Remove-Module`** unloads the current module from your session:

```
> Remove-Module <modulename>
```

When you are developing and testing your own modules, this command is especially helpful.

Creating your own modules

To make your functions easier to ship to other systems, creating a module is a great way. As the description of full-blown modules would exceed the scope of this book, I will describe the basics of how to quickly get started.

Please also have a look at the official PowerShell module documentation to better understand how modules work and how they should be created: https://docs.microsoft.com/en-us/powershell/scripting/developer/module/writing-a-windows-powershell-module.

When working more intensively with PowerShell modules, you might also come across many different files, such as files that end with **.psm1**, **.psd1**, **.ps1xml**, or **.dll**, help files, localization files, and many others.

I will not describe all the files that can be used in a module, but I will describe the most necessary files – the **.psm1** file and the **.psd1** file.

.psm1

The **.psm1** file contains the scripting logic that your module should provide. Of course, you can also use it to import other functions within your module.

.psd1 – the module manifest

The **.psd1** file is the manifest of your module. If you only create a PowerShell script module, this file is not mandatory, but it allows you to control your module functions and include information about the module.

Developing a basic module

Creating a basic PowerShell module can be as simple as writing a script containing one or more functions, and saving it with a **.psm1** file extension.

First, we define the path where the module should be saved in the **$path** variable and create the **MyModule** folder if it does not exist yet. We then use the **New-ModuleManifest** cmdlet to create a new module manifest file named **MyModule.psd1** in the **MyModule** folder. The **-RootModule** parameter specifies the name of the PowerShell module file, which is **MyModule.psm1**.

Using the **Set-Content** cmdlet, we create the **MyModule.psm1** file and define the **Invoke-Greeting** function, which we wrote earlier in this chapter:

```
$path = $env:TEMP + "\MyModule\"
if (!(Test-Path -Path $path)) {
    New-Item -ItemType directory -Path $path
}
New-ModuleManifest -Path $path\MyModule.psd1 -RootModule MyModule.psm1
Set-Content -Path $path\MyModule.psm1 -Value {
    function Invoke-Greeting {
        [CmdletBinding()]
        param(
            [Parameter(Mandatory=$true)]
            [string]$Name
        )
        "Hello, $Name!"
    }
}
```

When you want to use a module in your PowerShell session, you can either import it directly into your session or copy it into one of the **PSModule** paths. To ensure that the module is easily accessible for future

use, it's recommended to copy it to one of the **PSModule** paths. The **PSModule** paths are directories that are searched for modules when you use the **Import-Module** cmdlet. To see the **PSModule** paths, you can run the following command:

```
> $env:PSModulePath
```

Once you have determined which **PSModule** path to use, you can copy the module directory to that location. After copying the module to the appropriate **PSModule** path, you can then import the module using the **Import-Module** cmdlet:

```
> Import-Module MyModule
```

Alternatively, when you are in the development phase, you can import the module directly into your session, without having it copied in one of the **PSModule** paths, using **Import-Module**:

```
> Import-Module $env:TEMP\MyModule\MyModule.psd1
```

By copying the module to a **PSModule** path, you can easily import it into any PowerShell session without having to specify the full path to the module.

Now, you can call the function that was defined in the **MyModule** module:

```
> Invoke-Greeting -Name "Miriam"
```

Congratulations, you just created and executed your first very own module!

You can compare your own module with the demo module of this chapter: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter02/MyModule**.

MODULE MANIFEST OPTIONS

Have a closer look at the options that are available within the module manifest. For example, you can also specify the author, the description, or modules that are required to install this module, using the **RequiredModules** *hashtable.*

As you become more familiar with module development and want to take your code to the next level, you can explore tools such as **PSModuleDevelopment**, which can help you with your development tasks, and also with later CI/CD tasks: **https://psframework.org/documentation/documents/psmoduledevelopment.html**.

Summary

In this chapter, you have learned the fundamentals of PowerShell scripting. After refreshing the basics of variables, operators, and control struc-

tures, you are able to create your very own scripts, functions, and modules.

Now that you are familiar with the PowerShell basics and you are able to work with PowerShell on your local system, let's dive deeper into PowerShell remoting and its security considerations in the next chapter.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, check out these resources:

- Everything you want to know about arrays: **https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-arrays**
- Everything you want to know about hashtables: **https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-hashtable**
- Everything you want to know about `$null`: **https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-null**
- Everything you want to know about `PSCustomObject`: **https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-pscustomobject**
- About functions: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions**
- Functions 101: **https://docs.microsoft.com/en-us/powershell/scripting/learn/ps101/09-functions**
- About functions' advanced parameters: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters**
- Cmdlets versus functions: **https://www.leeholmes.com/blog/2007/07/24/cmdlets-vs-functions/**
- Modules help pages: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_modules

You can also find all links mentioned in this chapter in the GitHub repository for ***Chapter 2*** – no need to manually type in every link: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter02/Links.md**