CHAPTER 2

The PowerShell Language

In this chapter, we will introduce the PowerShell language. We will start by considering some of the challenges faced when learning PowerShell, which can be discouraging. Then we will review a simple script so as to get a feel for the language. From there we will discuss comparison and mathematical operators and logical operands. PowerShell commands are called cmdlets, pronounced *command-lets*. We'll discuss Microsoft's cmdlet naming conventions, then review of some of the most prominent cmdlets. After this, we'll discuss how to use variables and go over the supported data types. PowerShell has advanced support for string manipulation, and we will review some of these features, including regular expressions, which are used for string matching, and here strings, which allow long string values that include special characters. Then we'll discuss PowerShell's control flow statements that support conditional code execution and various looping constructs. We will close the chapter by reviewing PowerShell's support for arrays and a special object called a hash table. Hash tables are arrays of key/value pairs ideal for code translations, such as a state abbreviation code for the state name. By the end of this chapter, you will have a basic grasp of the PowerShell language.

Challenges to Learning PowerShell

There are a number of challenges to learning PowerShell that can discourage beginners. First, there is the architecture of PowerShell as a scripting language; to use it, we need to learn to think differently than we would in languages like C#. For example, piping data from one command to the next is ubiquitous in PowerShell but is not something C# supports. Second, there is a competing legacy shell, CMD.Exe, which, though limited in its use, still has loyal users who resist changing to PowerShell. Third, PowerShell started as a Windows administration tool and therefore most examples available focus on Windows administration. Getting examples of typical development scenarios is difficult. Fourth, because PowerShell is so versatile, there are many ways to accomplish a given task, and this can be overwhelming to a beginner.

Thinking Differently

If you are used to programming languages like T-SQL, C#, or VB.Net, PowerShell's architecture takes some getting used to. For example, piping the output of one command into another, which is fundamental to PowerShell, is not something inherently supported by most languages. Rather, you would loop through a collection and process each item. In PowerShell you can usually avoid the work of setting up loops and individually accessing object properties. To really leverage PowerShell, you need to integrate piping into your thinking. Similarly, as a database developer, you may tend to think of variables as distinct data types, like strings and integers. After all, T-SQL variables are simple data types. PowerShell variables are always objects, which you need to bear in mind when you code. Otherwise, you could end up doing a lot more work than is needed. For example, say you want to read a text file and write out the lines that contain the string "stuff" to another file. There are a few ways to do this. Let's review two of them to contrast the different methods. First, let's set up the text file as shown in Listing 2-1.

Listing 2-1. Creating a text file

```
Set-Location $env:USERPROFILE

@"
This is just a file
stuff1
stuff2
lastline
"@ > "$env:USERPROFILE\file1.txt"
```

The first line in Listing 2-1 sets the current folder to the user's default. The second line uses a PowerShell feature called a here string to create the file. We will discuss what here strings are later, but for now just know this will create a text file with exactly what is in the quotes. Note: The following two code examples require the default location be set to `$env:USERPROFILE`.

One way we can perform the task is to read the text file and write out lines that contain the string "`stuff`" to another file, as shown in Listing 2-2.

Listing 2-2. Writing output to a file using StreamWriter

```
Set-Location $env:USERPROFILE
$streamin = new-object System.IO.StreamReader("file1.txt")
$streamout = new-object System.IO.StreamWriter("file2.txt", 'Append')

[string] $inline = $streamin.ReadLine()

Do
{
   if ($inline.Contains("stuff") )
     {
        $inline
        $streamout.WriteLine($inline)
    }
     $inline=$streamin.ReadLine()
  }  Until ($streamin.EndOfStream)

$streamout.Close()
$streamin.Close()
```

This example is intentionally complex so as to show how difficult you can make your work. We could simply pipe everything to the `Add-Content` cmdlet as shown here:

```
Get-Content file1.txt | Where-Object { $_.Contains("stuff") } | Add-Content file2.txt
```

This statement uses the cmdlet `Get-Content` to load the file, pipes this into the `Where-Object` cmdlet that extracts lines with the word "`stuff`" and pipes that into `Add-Content`, which adds the lines to `file2.txt`. The lesson here is to leverage what PowerShell does best; it is worth your time searching to find an efficient method to do the work.

Again, compared to other languages you may have used, PowerShell's syntax is very different. For example, languages like VB and C# do not allow variables to start with a $, but in PowerShell all variables start with a $. In the old BASIC programming days, a variable ending with a $ meant it was a string. Comparison operators in many languages use standard mathematical syntax, i.e., > for greater than, < for less than, = for equal, etc. PowerShell uses a dash, followed by several letters for operators, i.e., `-gt` for greater than, `-lt` for less than, `-eq` for equals. Logical operands are `-and` and `-or`. And to make the adjustment to PowerShell even more awkward, comparisons must be enclosed in parentheses. Finally, even calling functions is different than what you may be used to. Functions and cmdlets are called without parentheses, and parameters are separated by spaces. Let's look at the PowerShell Syntax versus T-SQL and C#.

Declaring a variable...

```
PowerShell:    $myvar = "value"
C#:            string myvar = "value";
T-SQL:         declare @myvar varchar(50) = "value";
```

If condition...

```
PowerShell:    if ($myvar -eq "value") -and ($myothervar = "value") {  .. some code ..}
C#:            if (myvar   == "value" and mothervar == "value") {...some code...}
T-SQL          IF @myvar = 'value' and @myothervar = 'value'  BEGIN  ...some code...
               END
```

Making a function call...

```
PowerShell:    ufn_myfunction parm1 parm2
C#:            ufn_myfunciton(parm1, parm2);
T-SQL
  Scalar       select dbo.ufn_myfunction(@parm1, @parm2);
  Table Valued  select * from dbo.ufn_myfunction(@parm1, @parm2);
```

Just Got to Get the Job Done

One of the main reasons I put off learning PowerShell was the pressure to get the job done as soon as possible. When I mentioned learning and using PowerShell, the typical response would be that there was not time, so we should do the task the tried and proven way, such as using old command batch files. The truth is that there is never a good time to slow down development by learning and applying new tools. However, how can anything ever improve if you don't take the time to do so? In my case, I was developing a new, complex ETL process with a tool other than SQL Server Integration Services (*SSIS*). It required loading a number of flat files in various formats. The ETL tool had a number of limitations. For example, all flat files must have column headings, and you must assign a single fixed file name for the ETL to reference. I needed to merge multiple files into one file and insert column headings at the top of the merged file. I also needed to decrypt and unzip the files coming in and move them to an archive folder after processing. My client was comfortable with the old DOS-style command files and had already developed some to handle these tasks. When I considered developing dozens of batch command files with the archaic syntax, limitations, and lack of reusability, I knew that was heading in the wrong direction. I sold the client on PowerShell. In retrospect, it was a great decision, because I left behind a clean library of reusable functions, thus making the ETL process more maintainable and extensible. I relate this story to you to encourage you to sell management and maybe even yourself on learning and using PowerShell.

Getting Good Examples

One of the challenges I found in learning PowerShell was the lack of examples targeted to development or ETL tasks. Since PowerShell was initially created to support Windows administration, most books, blogs, and online articles are about administrative tasks. This is starting to change, and I hope this book is a big help in that direction. More and more writing about using PowerShell for development is being published. I am convinced PowerShell is too useful to stay limited to Windows administration tasks and that developers will need to learn it.

The Paradox of Power

Ironically one of the things that makes PowerShell a challenge to master is that it is so incredibly flexible, extensible, and integrated into the Windows environment that it's easy to get overwhelmed by all the things you can do and various ways you can code it. No matter how well I think I have mastered PowerShell, I am humbled by just searching the Internet for some new applications of the language. Each release of Windows, .Net, SQL Server, SharePoint, and other Microsoft applications is quickly integrated into PowerShell. My advice is to take your time learning and focus on basic tasks you need to get done in your job.

The PowerShell Language

In this section we will discuss the PowerShell language. We'll start by reviewing a simple script in order to provide an overview of the language components and syntax. Then we will review the elements of the language, starting with comparison and mathematical operators and logical operands. We'll discuss PowerShell commands and how to use them. This will be followed by a discussion of variables and variable types. Because PowerShell's support for strings is so extensive, we will review strings and string manipulation in great detail. Then we will discuss reading and writing to flat files. After this, we'll discuss the control flow statements, which include conditional and looping constructs. An interesting feature of PowerShell is its built-in support for storing scripts in variables know as script blocks. We will discuss this useful feature in detail. PowerShell has excellent support for variable arrays, and we'll review how to make use of these features, including the special array type called a hash table, which has built-in support to look up values based on a key.

A Brief Introduction Using a Script

Now that we've considered the challenges to learning PowerShell, let's start to look at the language itself. I find the best way to learn a language is to jump right in and look at the code in a small program. This shows the syntax and also provides a sense of context on how various elements are used together. So let's take a look at Listing 2-3. It's a long script, but don't worry. We're going to discuss how it works in detail.

Listing 2-3. This script uses the Windows speech API to speak

```
<# Author:  Bryan Cafferky      Date:  2013-11-29
   Purpose:  Speaks the input...
  Fun using SAPI - the text to speech thing....
#>

# Variable declarations...
$speaker = new-object -com SAPI.SpVoice  # PowerShell defaults variables to objects

[string]$saythis = ""    # Note: declare the type as string to have the string methods available

[string] $option = "x"

while ($option.toUpper() -ne "S")
```

```
{
# --> Open Brace defines the start of a code block.
   $option = read-host "Enter d for read directory, i for say input, s to stop"

   if ($option -eq "d")
   {
     $saythis = Get-ChildItem
     $saythis = $saythis.substring(1, 50)
   }
   elseif ($option -eq "i") {
          $saythis = read-host "What would you like me to say?"
         }
   else {
          $saythis = "Stopping the program."
      }

   $speaker.Speak($saythis, 1) | out-null   # We are piping the return value to null
}                                           # --> Closing Brace defines the end of a code block.
```

This script in Listing 2-3 creates an instance of a Windows Speech Application Interface (SAPI) object and stores it in the variable `$speaker`. It then prompts you with a message "`Enter d for read directory, i for say input, s to stop`" to enter an option. If you enter 'I', you are prompted for what you want SAPI to say and then SAPI says it. If you enter a 'd', a partial list of the current directory contents, i.e., the first one hundred characters, is read. A `while` loop will cause the script to keep looping back until the user enters an 's' or 'S'. When you enter an 's' or 'S', the script will say "Stopping the program" and stop. This short program covers a lot of elements in the PowerShell language. I encourage you to look at the code closely to see what you can figure out about the language. How is it different from other languages you have used?

Let's review the code more closely to discover more about the PowerShell language. You can see comments in the program lines copied below. Multi-line comments start with <# and end with #>, and a single-line comment starts with #. Comments are very important for documenting what the code does.

So a comment block is coded as shown below.

```
<# Author:  Bryan Cafferky      Date:  2013-11-29
   Purpose:  Speaks the input...
  Fun using SAPI - the text to speech thing....
#>
```

And a single-line comment is coded with a # character as shown below.

```
# Variable declarations...

Of course, you can include a single-line comment on the same line as a statement...
{                                                   # --> Open Brace defines the start of a
                                                          code block.

Let's look closer at the variables...

$speaker = New-Object -com SAPI.SpVoice
[string]$saythis = ""
[string]$option = "x"
```

Variables start with a $. Notice that $speaker does not have a type preceding it as the other two variables do. When you state a type in braces, such as [string], you are telling PowerShell to treat the variable as a string and to only allow string values to be stored there. It also makes the methods and properties of a string class available to the variable, which is why we can use the substring method later on. If no type is specified when you create a variable, PowerShell creates the variable as an object. You can create variables on the fly just by storing something in it; for instance, $myvar = Get-ChildItem would automatically create the $myvar variable and store the list of file properties in the current folder in it. Let's look again at $speaker:

```
$speaker = new-object -com SAPI.SpVoice
```

This line creates an instance of the COM object SAPI.SpVoice, which enables speech. PowerShell makes it very easy to create instances of COM and .Net objects. Notice that we did not have to add a reference or namespace prior to referencing the object. Many of the Windows object libraries are known to PowerShell without you having to do anything. All PowerShell variables are objects and provide a rich set of methods and properties. Avoid doing things the hard way and don't write unnecessary code to do something already provided by the object.

Next we have a loop, copied below. A loop will repeat some block of code until it reaches an end condition.

```
while ($option.toUpper() -ne "S")
{
```

This statement says to start a loop—beginning with the following open brace and ending with the matching closing brace—that will not end until the variable $option = "s" or "S". Wait a minute, it doesn't say that! Actually, rather than have to test for an upper case and lower case value, we use the built-in string function ToUpper() to convert the value to upper case before testing it. Notice that the condition is in parentheses. Although this format may be unfamiliar to you, if you omit them, the code will fail. Finally, notice the operand -ne. In T-SQL and many other languages, this would be coded as <>, but operands in PowerShell all start with a dash followed by letters. This may take some time to get used to. If you are getting a syntax error, check to make sure your operands are coded correctly. Not only are comparison operands coded this way, but so too are the logical operands—but more on that later.

```
$option = read-host "Enter d for read directory, i for say input, s to stop"
```

This line prompts the user with the message "Enter d for read directory, i for say input, s to stop" and waits for a reply to be entered. Once the user hits Enter, the response is stored in the variable $option:

```
if ($option -eq "d")
   {
     $saythis = Get-ChildItem
     $saythis = $saythis.substring(1, 50)
   }
   elseif ($option -eq "i") {
          $saythis = read-host "What would you like me to say?"
   }
   else {
          $saythis = "Stopping the program."
      }
```

The above if block does most of the work. It tests the value of $option, and if it is equal to "d", executes the immediately following statements found between the { and the }. If $option is not equal to "d", the code checks for a value of "i" and, if true, executes the immediately following statements found between the { and the }. Finally, if it reaches the else statement, the other two tests were false and it executes the statements after the `else` statement found between the { and the }. There are some PowerShell nuances to get used to here. First, the parentheses around the condition are required, and you will get errors if you omit them. Second, the conditional operand is not the usual = sign we might see in other languages. Finally, the braces to make off script blocks is similar to C#. Be careful when in your code that you line them up so you know you have a closing brace for each open brace. The result of this `if` block is that the variable `$saythis` is assigned a value.

```
$speaker.Speak($saythis, 1) | out-null   # We are piping the return value to null
```

This statement uses the `Speak` method of the object `$speaker` to read the string `$saythis` out loud. The parameter, `1`, is the voice and | `out-null` absorbs the return of the method, much like a void statement in C#. This is done fairly often in PowerShell.

We can't be complete without our program's ending brace, copied below.

```
}
```

Operators and Operands

PowerShell comparison operators can be a bit difficult to get used to because they are not the usual mathematical operators most languages use. Instead they look like parameters with a dash prefix followed by the operand name. Logical operands are more intuitive, i.e., `and`, `not`, `and !`, but these also must be prefixed with a dash. Fortunately, mathematical operations use the traditional operators.

Comparison operators are used to compare values such as if A equals B. PowerShell has a comprehensive set of comparison operators. By default all operands are case insensitive. To make an operand case sensitive, prefix it with a `c`. The complete list of comparison operators is shown in Table 2-1.

Table 2-1. *PowerShell Comparison Operators*

Case Insensitive	Case Sensitive	Description	Case Insensitive Example
`-eq`	`-ceq`	Test for equal values.	`$s -eq "test"`
`-ne`	`-cne`	Test for non-equal values.	`$s -ne "test"`
`-ge`	`-cge`	Test for greater than or equal value.	`$s -ge 5`
`-gt`	`-cgt`	Test for greater than value.	`$s -gt 5`
`-lt`	`-clt`	Test for less than value.	`$s -lt 5`
`-le`	`-cle`	Test for less than or equal to value.	`$s le 5`
`-like`	`-clike`	Test for pattern in value.	`$s -like "smith*"`
`-notlike`	`-cnotlike`	Test for pattern not being in a value.	`$s -notlike "smith*"`
`-match`	`-cmatch`	Test for pattern using a Regular Expression.	`"Megacorp" -match "corp"`
`-notmatch`	`-cnotmatch`	Test for no match of a pattern using a Regular Expression.	`"Megacorp" -notmatch "corp"`
`-replace`	`-creplace`	Replace a set of characters with another set of characters.	`$s -replace "old", "new"`
`-contains`	`-ccontains`	Test if a array contains an item.	`"a","b" -contains "b"`
`-notcontains`	`-cnotcontains`	Test if a string does not contain a character or set of characters.	`"a","b" -notcontains "b"`

Logical operands connect expressions and are also prefixed with a dash. Table 2-2 shows PowerShell's logical operands with examples.

***Table 2-2.** PowerShell Logical Operands*

Case Insensitive	Description	Example	Result
`-and`	`Logical and`	`(5 -eq 5) -and (5 -eq 6)`	`false`
`-or`	`Logical or`	`(5 -eq 5) -or (5 -eq 6)`	`true`
`-not`	`Logical not`	`(5 - eq 5) -and -not (5 -eq 6)`	`true`
`!`	`Logical not`	`(5 - eq 5) -and !(5 -eq 6)`	`true`

Fortunately, mathematical operators are the same as used in most languages. Table 2-3 lists the operators with a description and coding example.

***Table 2-3.** PowerShell Mathamatical Operators*

Operator	Description	Example
`=`	Assigns a value to a variable.	`$x = 5`
`+`	Adds two values or append strings.	`$x = 1 + 5`
`+=`	Increments a variable or appends a string 'r to a string variable.	`$x+=5`
`-`	Subtracts two values.	`$x = 5 - 1`
`-=`	Decrements a variable.	`$x-=5`
`/`	Performs division on a number.	`$x = 6/2`
`/=`	Divide a variable.	`$x /= 3`
`%`	Performs division and returns the remainder.	`$x = (3/10)`
`%=`	Perform division on a variable, returning the remainder to the variable.	`$x %= 3`

Cmdlets

The heart of PowerShell is made up of a vast array of commands called cmdlets, pronounced *Command-lets*. We can identify command-lets by their naming, i.e., a verb and a noun separated by a dash. Taken from Listing 2-3, cmdlets are highlighted in bold on the lines below.

`New-Object` returns an object of the type specified, i.e., COM in this case.

```
$speaker = New-Object -com SAPI.SpVoice
```

`Get-ChildItem` returns the list of items in the current directory.

```
$saythis = Get-ChildItem
```

`Read-Host` prompts the user for input, which is stored in the variable `$saythis`.

```
$saythis = Read-Host "What would you like me to say?"
```

There are many command-lets, and more are added with each PowerShell release. You can even write your own command-lets. Naming follows a convention, which helps the developer get an idea of what a command-let does. Command-lets that get values start with `Get`, while command-lets that set a value start with `Set`. Here is a list of some common command-let prefixes and their meanings. For a complete list, see `http://msdn.microsoft.com/en-us/library/ms714428%28v=vs.85%29.aspx`

Table 2-4. *Cmdlet Verb Prefixes*

Cmdlet Verb Prefix	Meaning/Example Cmdlet
`Add`	Adds something on to an object. `Add-Content` adds to a file.
`Clear`	Clears out an object. `Clear-Content` empties a file.
`Copy`	Copies an item. `Copy-Content` copies data.
`Export`	Reformat or extract data. `Export-CSV` outputs the data in CSV format.
`Format`	Formats output. `Format-Table` formats the output as a table.
`Import`	Loads something into memory. `Import-CSV` reads in a file in CSV format.
`Move`	Moves something from a source to a target. `Move-Item` moves an item such as a file.
`New`	Creates an instance of an object. `New-Object` creates a new object instance.
`Out`	Outputs data. `Out-GridView` writes data to a windows grid.
`Read`	Takes input. `Read-Host` takes input from the keyboard.
`Remove`	Removes an item. `Remove-Item` deletes an object, such as a file.
`Rename`	Renames something. `Rename-Item` renames an object, such as a file.
`Write`	Outputs data. `Write-Host` outputs to the console.

Variables

In my work I find that most of the time my variables are either an object or a string. However, PowerShell supports a number of data types. In reality, all PowerShell variables are objects. However, by casting these objects to a type, PowerShell will treat the variable consistently with that type and expose properties and methods that are useful for the type. Table 2-5 provides a list of PowerShell variable types supported, with a description and example.

Table 2-5. *PowerShell Data Types*

Type	Description	Example Declaration
`string`	A string of Unicode characters.	`[string] $mvar = "somestring"`
`char`	A single Unicode character.	`[char] $myvar = "A"`
`datetme`	A datetime stamp.	`[datetime] $myvar = Get-Date`
`single`	Single-precision floating decimal point number.	`[single] $myvar = 12.234`
`double`	Double-precision floating decimal point number.	`[double] $myvar = 123.5676`
`Int`	32-bit integer.	`[int] $myvar = 10`
`wmi`	Windows Management Instrumentation instance or collection.	`[wmi] $myvar = GetWMI-Object Win32_Bios`
`adsi`	Active Directory Services object.	`[adsi]$myvar = [ADSI]"WinNT://BRYANCAFFERKYPC/BryanCafferky,user"`
`wmiclass`	Windows Management Instrumentation Class.	`[wmiclass] $my1 = [wmiclass]'win32_service'`
`boolean`	True or False.	`[boolean] $myvar = $true`

Cmdlet Output

If a cmdlet returns a value, you can either capture the output of it to a variable or push it through the pipe into another cmdlet. Values returned are objects, which means you can access the properties and methods through the variable. For example, the code below will get information about the current folder, store it in `$myvar`, which is then piped into `Get-Member`, which will list all the properties and methods available to `$myvar`.

```
$myvar = Get-ChildItem
$myvar | Get-Member
```

Now that we have the object loaded, we can do a lot of things with it. `$myvar` will display the properties of the folder in a list in the format shown below. Note: The values will be for whatever you have the current path point set at.

```
    Directory: C:\Users\BryanCafferky

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d-r--         8/15/2014  12:06 PM            Contacts
d-r--         8/15/2014  12:06 PM            Desktop
d-r--          9/3/2014   3:10 PM            Documents
d-r--          9/1/2014   6:38 PM            Downloads
d-r--         8/15/2014  12:06 PM            Favorites
d-r--         8/15/2014  12:06 PM            Links
d-r--         8/15/2014  12:06 PM            Music
d----         4/25/2013   2:13 PM            My Backup Files
d----         4/24/2013   1:32 PM            New folder
d-r--          9/3/2014   2:45 PM            Pictures
```

We can filter rows in `$myvar` using the `Where-Object` cmdlet.

```
$myvar | Where-Object  {$_.name -like "d*"}

The statement above will list just the lines that have a name that start with the letter 'd'.

  Directory: C:\Users\BryanCafferky

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d-r--        8/15/2014  12:06 PM            Desktop
d-r--         9/3/2014   3:10 PM            Documents
d-r--         9/1/2014   6:38 PM            Downloads
-a---         1/9/2014   9:13 AM          7 db.txt
```

You can also select just the properties you want and have them appear in the order you specify.

```
$myvar | Select-Object -Property Name, LastWriteTime

Name                                 LastWriteTime
----                                 -------------
Contacts                             8/15/2014 12:06:54 PM
Desktop                              8/15/2014 12:06:54 PM
Documents                            9/3/2014 3:10:54 PM
Downloads                            9/1/2014 6:38:20 PM
Favorites                            8/15/2014 12:06:54 PM
Links                                8/15/2014 12:06:54 PM
Music                                8/15/2014 12:06:54 PM
My Backup Files                      4/25/2013 2:13:02 PM
New folder                           4/24/2013 1:32:26 PM
Pictures                             9/3/2014 2:45:39 PM
```

These are just some examples. Let your imagination go wild as you play with variables that hold cmdlet return values. The possibilities are endless.

Strings

PowerShell's support for strings sets it above most languages. First, we have the basic string support that we would expect; i.e., we can create string variables and manipulate them. However, there is a special string type called a *here string* that is designed to store exactly what you assign it; formatting, line feeds, special characters, and all. Additionally, the string object type exposes a number of useful methods for searching, comparing, and manipulation. Some languages support a feature called *regular expressions*. Regular expressions are a standardized, powerful string pattern-matching and manipulation language. PowerShell supports regular expressions, and we will discuss how to apply this technology later.

Basic Strings

When you want to write out strings, you can enclose the value in double quotes or in single quotes. There is a big difference between what you choose. When you use double quotes, PowerShell will replace any variables or expressions in the string before it is output. See the examples below.

```
$myvar = get-date

Write-host "The date and time is $myvar `t more stuff."
```

Displays the output below.

```
The date and time is 08/09/2014 14:27:25       more stuff.
```

The variable is replaced with its value and `` `t `` is replaced with a tab.
If we code this with single quotes as shown below:

```
Write-host 'The date and time is $myvar `t more stuff.'
```

We get the output below.

```
The date and time is $myvar `t more stuff.
```

Single quotes tell PowerShell to write exactly what is in the quotes. The `` `t `` is called an *espace sequence*. That is a character that controls formatting. Some common escape characters are:

```
`b      Alert/beep
`n      New line
`r      Carriage return
`r`n    Carriage reurn and line feed, which is often used for writing to files
```

Sometimes you need a single quote to be output, and the string is already encased in single quotes. To tell PowerShell this you double the character, i.e., use two single quotes to output one single quote, as follows:

```
write-host 'This has ''single quotes''.'
```

This will display the output below.

```
This has 'single quotes'.
```

If the string is enclosed in double quotes and you need to include a double quote, put two double quotes in the string:

```
write-host "This has ""double quotes"" "
```

Writes the output below to the console.

```
This has "double quotes"
```

If we are enclosing the string in double quotes, we can just use single quotes within the string and it will be output correctly. Similarly, if our string is enclosed in single quotes and we need to include a double quote, we can include double quotes within the string and it will be output correctly. This is because the overall string quote delimiter we use is different than the quote character we want to include in the string. The examples below shows how this works.

```
write-host "This has 'single quotes'."
```

Writes the output below to the console.

```
This has 'single quotes'.
```

```
write-host 'This has "double quotes". '
```

Writes the output below to the console.

```
This has "double quotes".
```

Here Strings

Sometimes we need to assign a value to a string that has multiple lines, embedded single and double quotes, and various special characters. Doing this in C# or T-SQL or most languages is a pain, and you have to build the string by concatenating pieces. PowerShell has a special way to handle this called a here string. Let's look of using here strings in Listing 2-4.

Listing 2-4. Using here strings

```
$myvar = "Bryan"

$myherestring = @"
" Four score and $ *

Seven years... ,,, ~` ' ago

$myvar

!
"
"@

Write-Host $myherestring
```

The above script displays the output shown below.

```
" Four score and $ *

Seven years... ,,, ~` ' ago

Bryan

!
"
```

A here string is marked by the beginning and ending `@` with the value in quotes. You can enter anything you want just as you want it to be retained, with carriage returns and special characters. Not only that, but embedded variables will still be expanded for you, as we can see above where `$myvar` is replaced with the value "`Bryan`". Here strings are very useful, indeed.

String Manipulation

As a database developer I often need to do some pretty hairy string manipulation. T-SQL is limited in this area, but PowerShell is the best tool for string manipulation I've ever used. Let's start with the typical things you might need to do. The comment above each statement explains what the method does. Listing 2-5 shows some examples of string manipulation.

Listing 2-5. Manipulating strings

```
[string] $mystring = "  This is a nifty nifty string. "

# Get a part of the string.
Write-Host $mystring.Substring(0,5)

# Get the length of the string.
Write-Host $mystring.Length

# Comparing...
Write-Host $mystring.CompareTo("This is a nifty nifty string.")   # 0 = a match and
-1 = no match.
Write-Host $mystring.Equals("This is a nifty string.")      # returns True or False

# Search for set of characters in the string.
Write-Host $mystring.Contains("nifty") # returns True or False

# Does the string end with the characters passed into the method?
Write-Host $mystring.EndsWith(".")   # returns True or False

# insert the set of characters in the second parameter starting at the position specified in
the first parameter.
Write-Host $mystring.Insert(5, "was ")

# Convert to Upper Case
Write-Host $mystring.ToUpper()

# Convert to Lower Case
Write-Host $mystring.ToLower()

# Strip off beginning and trailing spaces.
Write-Host $mystring.Trim()

# Replace occurences of the set of characters specified in the first parameter with the set
in the second parameter.
Write-Host $mystring.Replace("nifty", "swell")
```

Regular Expressions

Something lacking in T-SQL but available in PowerShell is built-in support of regular expressions. If you have never used regular expressions, you're in for a treat, because they support amazingly flexible pattern matching on strings.

To test if a string has only digits, you would code something like statements below:

```
$var1 = "4017771234"
$var1 -match "^[0-9]+$"   #  Returns true as there are only digits.
$var2 = "401B7771234"
$var2 -match "^[0-9]+$"  # Returns false as there is a non-digit, B, in the string.
```

Regular expressions are powerful, but that power comes with complexity; regular expressions are practically a language in itself. I found the free website below that has, among other useful things, a searchable regular expression library. You can just search for the expression that fits your needs and copy it.

http://www.regxlib.com/?AspxAutoDetectCookieSupport=1

On that site, contributor Steven Smith supplied a number of useful expressions, a few of which I would like to share:

This expression tests for a phone number being in the U.S. format:

```
"333-444-5555" -match "^[2-9]\d{2}-\d{3}-\d{4}$"   # Returns True
```

This expression tests for a string matching the U.S. 5- or 9-digit zip code format:

```
"12345-1234" -match "^\d{5}$|^\d{5}-\d{4}$"   # Returns True.
```

This expression tests for string in date format MM/DD/YYYY:

```
"01/01/2014" -match "^\d{1,2}\/\d{1,2}\/\d{4}$"    # Returns True

"01/15/14" -match "^\d{1,2}\/\d{1,2}\/\d{4}$"    # Returns False
```

Editing Strings Using Regular Expressions

We can use regular expressions with the `replace` operator to edit strings. Some examples of this are below. The comment to the right of each example gives the result.

Replaces all occurrences of the letter e with the letter b.

```
"ppowershell" -replace 'e', 'b'  # Result is 'ppowbrshbll'
```

Replaces the letter p with the letter b only if p is the first character.

```
"ppowershell" -replace '^p', 'b' # Result is 'bpowershell'
```

Replaces the letter l with the letter b only if l is the last character.

```
"ppowershell" -replace 'l$', 'b' #. Outputs ppowershelb
```

Files

Reading in and writing to flat files is so simple in PowerShell that you may do what I did when I started, which is do a lot more work than necessary. Let's do some things with files to give you a sense of how simple it is. First let's create a text file to play with, as shown in Listing 2-6.

Listing 2-6. Creating a file for testing

```
# Create a file...
"this is a test file created for demostration purposes" > infilepsbook.txt

$myfile = Get-Content infilepsbook.txt

$myfile
```

The first line above creates a new file and inserts the quoted string into it, as the single carrot, `>`, is a piping symbol that tells PowerShell to create a new file or overwrite a file if one is there already. Then we use the `Get-Content` cmdlet to load the file into the variable `$myfile`. That's all you need in order to do anything you want to load a flat file into a variable.

If you enter a period just after `$myfile`, you will get a list of methods and properties available. Right off the bat, you have a lot of things you can do.

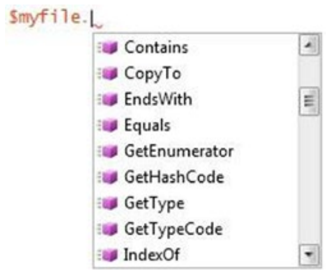


Figure 2-1. *Intellisense showing a variable's methods and properties*

I encourage you to play with the properties and methods to see what you can do. Search the Internet for examples of how people are using them.

Let's make a copy of our file.

```
$myfile > outfilepsbook.txt
```

How about concatenating these files?

```
Get-Content infilepsbook.txt > newfilepsbook.txt
Get-Content outfilepsbook.txt >> newfilepsbook.txt
Get-Content newfilepsbook.txt
```

We use `Get-Content` to load the file's data and then redirect it to `newfilepsbook.txt`. Notice that the second line uses `>>`. Using `>>` causes the data to be appended to existing file contents. Another way to do this is:

```
Add-Content newfilepsbook.txt (Get-Content outfilepsbook.txt)
```

The parentheses cause `Get-Content` to load `outfilepsbook.txt` before trying to add it to `newfilepsbook`.

Control Flow

In this section, we will discuss PowerShell's control flow statements, which include condition expressions and looping constructs. As expected, we have the traditional `If/Else` support, but we also have more concise ways of performing such conditional executions, such as by using the `Switch` statement. Looping constructs include a number of ways to conditionally iterate over a block of code, and we'll cover these in detail. The following link provides some nice coverage of PowerShell loops:

`http://powershell.com/cs/blogs/ebookv2/archive/2012/03/15/chapter-8-loops.aspx`

If/Elseif/Else

The `if` conditional tests for a condition, and if it is true, executes the block of code following the `if`. Consider this example:

```
if (test-path C:\Users\) {
   write-host "The path exists."
   }
else {
   write-host "The path does not exist."
   }
```

The statement above uses the `Test-Path` cmdlet to verify the path `C:\Users\` exists. If it does, true is returned and the first code block executes. If the path does not exist, false is returned and the `else` code block executes. The braces define the beginning and end of the code block.

For more complex conditions, we can include the `elseif` statement to add other conditions to test for. Listing 2-7 shows an example of this.

Listing 2-7. Using If/ElseIf

```
$datetime = Get-Date
$answer = Read-Host "Enter t for time, d for date or b for both"

if ($answer -eq 't') {
   Write-Host "The time is " $datetime.ToShortTimeString()
   }
elseif ($answer -eq "d") {
  Write-Host "The date is " $datetime.ToShortDateString()
   }
else {
  Write-Host "The date and time is $datetime"
}
```

The first statement in Listing 2-7 gets the current date/time and stores it in variable `$datetime`. An interesting thing in the above example is that usually you can incorporate variables into a string implicitly just by inserting the variable name. However, with the object method `datetime.ToShortDateString()` and `$datetime.ToShortTimeString()` this does not work, as you get a value like `"08/09/2014 08:01:01.ToShortDateString()"`. By rewording the `Write-Host` statement to close the quoted string and have the object method call separated by a space, we get the desired result. Note how many different ways the same thing can be coded.

Switch

The Switch statement provides an efficient way to test for multiple conditions with minimal code. It starts with evaluating an expression followed by each possible result and related code to be executed.

```
$datetime = Get-Date
switch (Read-Host "Enter t for time, d for date or b for both")
{
"t"      {write-host "The time is " $datetime.ToShortTimeString()}
"d"      {write-host "The date is " $datetime.ToShortDateString()}
"b"      {write-host "The date and time is $datetime"}
default  {write-host "The date and time is $datetime"}
}
```

The value entered is compared to several possible values. If a match is found, the code in between the braces will execute. Notice how compact and readable this is compared to the equivalent If statement example earlier. We even made it more concise by including the Read-Host prompt within the switch expression. The default condition will execute if none of the prior conditions are met.

This next example demonstrates how the switch statement works using the test expression with multiple True value results. Unlike if statements, switch statements continue running when True value matches are found. So it is possible that more than one code block will be executed. The results shows that test expressions that equal 10 will run:

```
switch (10)
{
  (1 + 9) {Write-Host "Congratulations, you applied addition correctly"}
  (1 + 10) {Write-Host "This script block better not run"}
  (11 - 1) {Write-Host "Congratulations, you found the difference correctly"}
  (1 - 11) {Write-Host "This script block better not run"}
}
```

When a condition is met in the switch statement, the switch statement is not exited after running the associated condition code. Rather, PowerShell continues to evaluate the other conditions, and if another condition is met, executes the associated code. In the first example, this cannot happen, but consider any case in which multiple conditions can be met, but you want the switch statement to exit once a condition has been met. This can be accomplished by adding the break statement, as shown in the example below.

```
$salesamount = 1000.00

switch ($salesamount)
{
{$_ -ge 1000}   {write-host "Awesome sales."; break}
{$_ -ge 100}    {write-host "Good sales."; break}
{$_ -lt 100}    {write-host "Poor sales."; break}
default         {write-host "Bad value"}
}
```

You may be wondering what $_ means. It is a special name PowerShell gives to the current object being processed. In the case above, $_ -ge 1000, means $salesamount -ge 1000.

For Loop

The For loop is a simple looping construct that iterates a code block a certain number of times, which is determined by comparing a counter value to a termination value and incrementing the counter value with each iteration.

```
for ($counter = 1; $counter -le 5; $counter++)
{
   Write-Host "Counter is $counter"
}
```

This code initializes the variable $counter to 1 and will execute the code in braces while $counter is less than or equal to 5. On each loop execution, $counter is incremented by 1.

For Each

The foreach loop differs from the For loop in that it iterates over an object collection, and since virtually everything in PowerShell is an object, this looping construct is very handy.

```
$objectarray = (0..3)

foreach ($number in $objectarray) {
Write-Host "Counter value is $number"
}
```

In the code above, the first line creates an array, $objectarray, consisting of integers with values 0, 1, 2, and 3. The foreach statement iterates once for each element in $objectarray.

An interesting thing about the foreach loop is that it can be coded in a very compressed manner. For example, the above code could be written as follows

```
(0..3) |% {
Write-Host "Counter value is $_"
}
```

Notice the piping | symbol followed by %, which is a placeholder for an object instance. So (0..3) implicitly creates a four-element array of integers and then pipes it into the code block that follows.

While

While and Do Until loops are very similar, but apply the test condition a little differently. The While loop will interact as long as the test condition is true. The Do Until loop will iterate until the test condition is true. Let's look at Listing 2-8, which demonstrates how to use the While loop.

Listing 2-8. Using the while loop

```
$datetime = Get-Date
$answer = "x"

While ($answer -ne "e")
{
   $answer = Read-Host "Enter d for date, t for time or e to exit"
   if ($answer -eq "t")
      {
      write-host "The time is " $datetime.ToShortTimeString()
      }
    elseif ($answer -eq "d")
      {
      write-host "The date is " $datetime.ToShortDateString()
      }
}
```

Listing 2-8 shows a good example of when you might choose to use a `While` loop. The first line initializes `$datetime` as the current timestamp. Then `$answer` is initialized to "x". The loop will keep repeating until the user decides to end it by entering "e" when prompted by the `Read-Host` cmdlet. If "t" is entered, the time is displayed. If "d" is entered, the date will display. The `While` and `Do Until` loops can provide more flexibility than the `For` loop we saw earlier can. Now let's look at an example of using a `Do Until` Loop, as shown in Listing 2-9.

Listing 2-9. Using the Do Until loop

```
$datetime = Get-Date
$answer = "e"

Do
{
   $answer = Read-Host "Enter d for date, t for time or e to exit"
   if ($answer -eq "t")
      {
      write-host "The time is " $datetime.ToShortTimeString()
      }
    elseif ($answer -eq "d")
      {
      write-host "The date is " $datetime.ToShortDateString()
      }
}
until ($answer -eq "e")
```

As you can see, there is not a big difference between the `While` loop and the `Do Until` loop. What we use is a matter of preference. However, there are a couple of differences that may matter when deciding which to use. One difference is that the condition is tested before the code block is executed with the `While` loop, but after the code has been executed with the `Do Until` block. In the example above, although `$answer` is assigned a value of "e" prior to the loop, the loop still executes at least once. If `$answer` is assigned a value of "e" prior to the `While` loop, the code in the loop would never execute. Another difference is that test condition operands are reversed; i.e., the `While` loop is executed while a condition is true, but the `Do Until` loop is executed until the condition is true. These differences can affect the way you need to organize your code.

Script Blocks

An interesting feature of PowerShell is that you can store scripts in variables and then execute the code by referencing the variable. This provides a simple form of reusability. For example, suppose you have a block of code you need to execute several times in a script. You could just copy the code multiple times, but then you would need to modify each copy if any changes are needed. It also would make the code harder to read. This is where reusability comes in. Later, I will cover more sophisticated ways to maximize code reusability, but script blocks offer the simplest way to get some of the benefits.

Imagine you have the following code:

```
$message = "Hi"
$title = "Title"
$type = 1

[System.Windows.Forms.MessageBox]::Show($message , $title,  $type)
```

And you want to display message boxes from different places in your script. After all, it is a useful bit of code. However, the syntax `[System.Windows.Forms.MessageBox]::Show($message, $title,  $type)` is not very easy to remember, and it would be possible to mess up the format. What if we could code this statement in such a way as to make it easy to call wherever we need it? Listing 2-10 shows an example of how to do this.

Listing 2-10. Using a script block with parameters

```
$MsgBox = {param([string] $message, [string] $title, [string] $type)  $OFS=','; [System.
Windows.Forms.MessageBox]::Show($message , $title,  $type) }

Invoke-Command $MsgBox -ArgumentList "First Message", "Title", "1"

Invoke-Command $MsgBox -ArgumentList "Second Message", "Title", "1"

Invoke-Command $MsgBox -ArgumentList "Third Message", "Title", "1"
```

The code in Listing 2-10 creates a variable, `$MsgBox`, and stores the PowerShell script to display a message box in it. Once that is done, the script can be called as many times as we like. Best of all, we can pass parameters to the script so we can change the message, title, and message-box type with each call. Bear in mind, the script can be much longer and more complex, but the concept still applies. If the code for parameters is not clear to you, don't worry. I'll be covering that in detail later.

Arrays

Arrays are one of the most useful constructs in programming, and PowerShell has the best support for arrays of any language I've used. What I mean by this is that in most languages the syntax and amount of code required to effectively use arrays is significant, but in PowerShell arrays are easy to create and use. Actually, PowerShell is designed around arrays and often returns values that are arrays either of simple types or objects. For example, consider the following code:

```
$FileList = Get-ChildItem
$FileList.Count
```

`$FileList` contains an array returned from `Get-ChildIte`, which is confirmed by the `Count` property—i.e., the number of elements in the array. You access an element in the array by specifying the array name followed by the element number in brackets, as shown below. Notice that array numbering starts at zero by default.

```
$FileList[0]
```

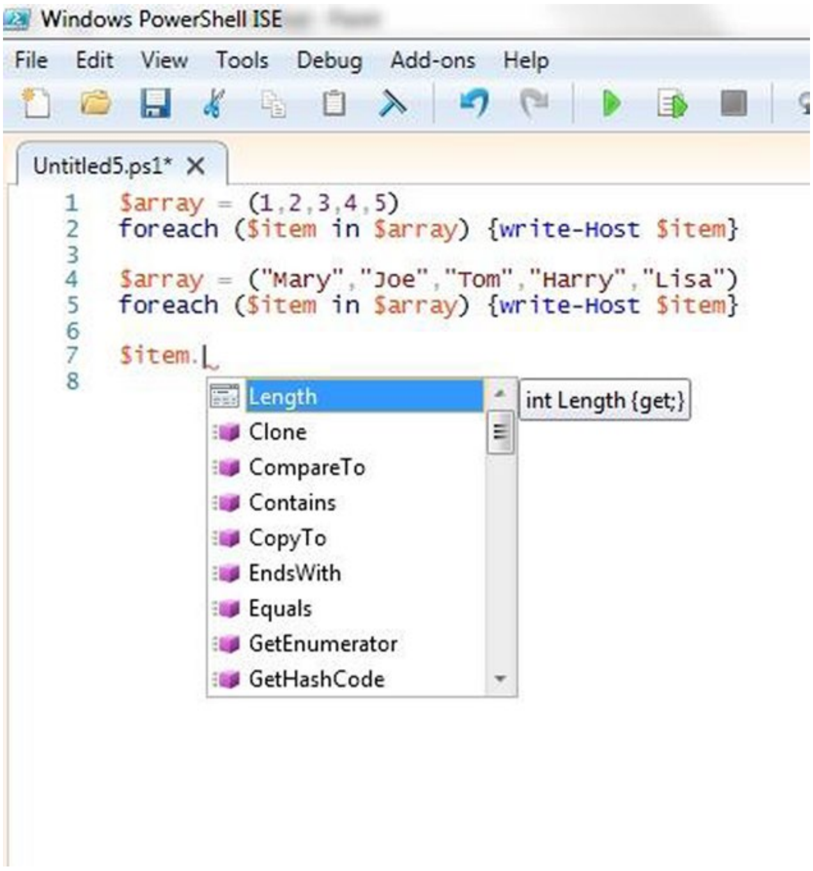
In other languages, we tend to think of arrays in terms of two types, which are simple arrays and object collections. An array of integers is a simple array, whereas an array of Windows controls is a collection. However, this distinction is not relevant in PowerShell, as all variables are objects. Let's look at Listing 2-11, which shows some examples of creating arrays.

Listing 2-11. Using arrays

```
# Create an array of integers...
$array = (1,2,3,4,5)
foreach ($item in $array) {write-Host $item}

# Create an array of strings..
$array = ("Mary","Joe","Tom","Harry","Lisa")
foreach ($item in $array) {write-Host $item}
```

In Listing 2-11, notice that we can easily load the array by listing the elements separated by commas. In fact, because of the format of the value initialization, PowerShell automatically creates an array. Bear in mind that each element in the array has a number of methods and properties that can be leveraged. The ISE's intellisense makes this particularly clear, as shown in Figure 2-2.



```
$array = (1,2,3,4,5)
foreach ($item in $array) {write-Host $item}

$array = ("Mary","Joe","Tom","Harry","Lisa")
foreach ($item in $array) {write-Host $item}

$item.
```

Figure 2-2. *Intellisense in the PowerShell ISE*

Associative Arrays

As a database developer, using lookup tables to get values related to a key is very common. In PowerShell, we can use associative arrays known as hash tables to do something similar. Suppose we want to translate the state abbreviation into the state name. We can do this easily, as this code demonstrates:

```
$States = @{"ME" = "Maine"; "VT" = "Vermont"; "NH" = "New Hampshire"; "MA" =
"Massachusetts"; "RI" = "Rhode Island"; "CT" = "Connecticut" }
```

The `@{` characters designate a hash table is being created. We used just the New England states to keep the example short. The key is the value on the left side of the equal sign and the value returned is on the right side. In a real scenario you might load the list from a SQL Server table or flat file.

To look up the name for a state code, enter:

```
$States["RI"]   # Returns the name of the state, i.e., Rhode Island
```

The hash table object has methods that allow you to add, change, and delete rows similar to a SQL Server table. Examples of these operations are shown below.

```
# Add a new row...
$States["NY"] = "New York"

# See the list...
$States

#Remove a row...
$States.Remove("NY")

# Clear the table...
$States.Clear()

# Test for the existence of a key...
$States.ContainsKey("ME")
```

Hash tables are a feature you will probably use a lot in development work. It can be used any time we need a simple code translation.

Summary

This chapter introduced the PowerShell language. We began by discussing some of the challenges to learning PowerShell. Then we reviewed a script in detail to get a sense of the language. From there we discussed comparison, as well as mathematical operators and logical operands. Then we discussed PowerShell commands, called cmdlets (pronounced *command-lets*). We started with a discussion of Microsoft's cmdlet naming conventions followed by a review of some of the most prominent cmdlets. After this, we explained how to use variables and reviewed the supported data types. PowerShell has advanced string manipulation features, and we reviewed some these, including regular expressions used for string matching, and here strings, which support long string values that include special characters. Then we discussed PowerShell's control flow statements, conditional code execution, and various looping constructs. We closed the chapter by reviewing PowerShell's support for arrays and a special object called a hash table. Hash tables are arrays of key/value pairs ideal for code translations.