Translating VBScript to Windows PowerShell

One of the cornerstones of education is the concept of scaffolding: the idea that it is easier to learn something new if you relate your new knowledge to, and build upon, existing knowledge. For example, suppose you've spent your entire driving life behind the wheel of a car with an automatic transmission. Now, you'd like to learn how to drive a car that has a manual transmission. Should you forget everything you've ever learned about driving and start all over again from scratch?

Well, OK, maybe some people *would* be better off doing that. For most people, however, starting over from scratch makes no sense. Instead, by building on what you already know you'll find it much easier to learn to drive a stick shift. For one thing, that enables you to focus solely on learning how to shift. There’s no need to learn how to turn on the windshield wipers or signal for a left-hand turn; after all, you already know how to do those things.

The same is true when it comes to learning Windows PowerShell. Yes, Windows PowerShell is different than VBScript, sometimes dramatically different. However, the underlying concepts between the two are often exactly the same: an If statement allows you to take alternate routes depending on a specified condition; an uppercase a is still an *A*; a constant remains a value that cannot be changed during the running of a script. If you're familiar with VBScript you actually have a head-start over people who know nothing about scripting or programming. After all, you already understand many of the basic concepts. You don't have to learn the function of a Do While loop; you just have to learn the Windows PowerShell syntax for a Do While loop.

Hence this guide to translating VBScript to Windows PowerShell. We should point out, right from the start, that this is not intended to be a “word-for-word” translation guide; for better or worse, things don’t quite work that way. Instead, it’s designed to act as a reference guide as you begin writing your first Windows PowerShell scripts. After all, sooner or later you’re bound to run into a situation where you find yourself thinking, "OK, I need to use a Select Case statement here. I know how to do that in VBScript, but how do I do that in Windows PowerShell?” That’s where this guide can come in handy; it takes all the VBScript functions, statements, and operators and shows you, as much as possible, a Windows PowerShell equivalent.

Incidentally, we tried to find an equivalent for most of the commands in the VBScript Language Reference; however, we would never claim that we found the *best* equivalent for these commands. If you know of a better/faster/easier way to, say, format a number as currency in Windows PowerShell, please let us know. We’ll publish these alternate approaches as an addendum to this guide.

|  |  |
| --- | --- |
| **VBScript Function** | **Windows PowerShell Equivalent** |
| Abs | **Definition:** Returns the absolute value of a number.  For system administrators, VBScript’s Abs function, which returns the absolute value of a number, is like having a spare house key hidden somewhere in the backyard: most likely you’ll never use it, but if you ever *do* need a spare key (or an absolute value function), well …. Although Windows PowerShell doesn’t have a built-in method for returning the absolute value of a number you can achieve that result by using the .NET Framework’s System.Math class and the **Abs** method. This command assigns the absolute value of -15 to the variable $a:    $a = [math]::abs(-15)  When you run this command and then echo back the value of $a (something you can do simply by typing $a) you should get the following:  15 |
| Array | **Definition:** Returns a **Variant** containing an array.  In VBScript it's pretty easy to create an array: all you have to do is dimension a variable as an array, or use the **Array** function to add multiple values to a variable. Believe it or not, in Windows PowerShell it’s even *easier* to create an array; all you have to do is assign multiple values to a variable, no special functions or methods are required. For example, this command assigns all the colors of the rainbow to a variable named $a:  $a = "red","orange","yellow","green","blue","indigo","violet"  When you run this command and then echo back the value of $a you should get the following:  red  orange  yellow  green  blue  indigo  violet |
| Asc | **Definition:** Returns the ANSI character code corresponding to the first letter in a string.  It’s surprising how often scripters need to convert a character value (such as the letter *A*) to its ASCII equivalent (65). In Windows PowerShell you can determine the ASCII value of a character by using this crazy-looking line of code (which takes the character value of the letter *A* and then converts the result to a byte value):  $a = [byte][char] "A"  When you run this command and then echo back the value of $a you should get the following:  65 |
| Atn | **Definition:** Returns the arctangent of a number.  Granted, this one doesn't come up too terribly often in system administration scripts. Nevertheless, if you ever *do* need to calculate the arctangent of an angle you can do so using the System.Math class and the **Atan** method. This command assigns the arctangent of a 90 degree angle to the variable $a:    $a = [math]::atan(90)  When you run this command and then echo back the value of $a you should get the following:  1.55968567289729 |
| CBool | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Boolean**.  Sure, you can convert a string or numeric value to a Boolean value (True or False): all you have to do is convert the data type of the variable to a Boolean type. As far as we know, 0 will always be converted to False and anything that isn't zero (even something as nonsensical as the string value *cat*) will be converted to True. In the following example we assign the value 0 to the variable $a, then convert $a to a Boolean value:  $a = 0  $a = [bool] $a  When you run this command and then echo back the value of $a you should get the following:  False |
| CByte | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Byte**.  What do you mean, “Windows PowerShell is great and all, but there’s probably no way to convert a variable to the byte data type?" As it turns out, that's about as easy a conversion as you could hope to make. For example, the first of these two commands assigns a string value to the variable $a; the second command then converts $a to the byte data type (which, incidentally, consists of integers from 0 to 255):  $a = "11.45"  $a = [byte] $a  When you run these two commands and then use the GetType() method to determine the data type of $a you should get the following:  IsPublic IsSerial Name  -------- -------- ----  True True Byte  Incidentally, the value of $a will now be 11. |
| CCur | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Currency**.  Windows PowerShell uses the same data types as the .NET Framework; because the .NET Framework does not support the Currency data type that means Windows PowerShell doesn’t support this data type, either. However, while you cannot convert a variable to the Currency data type you can at least format the value so that it looks like a currency value when displayed onscreen. The following command assigns the value 13 to the variable $a and formats the value so it displays as currency:  $a = "{0:C}" -f 13  This command uses the .NET Framework currency formatting string **“{0:C}”** to format the value as currency (note that the value to be formatted is included as part of the **–f** parameter).  When you run the command and then echo back the value of $a you should get the following:  $13.00 |
| CDate | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Date**.  Need to make sure that a value is treated as a date? Then do this. First, assign the value to a variable; here we’re assigning the string 11/1/2006 to the variable $a:  $a = "11/1/2006"  In fact, if we now call the **GetType()** method we’ll see that $a is a string variable:  IsPublic IsSerial Name  -------- -------- ----  True True String  To convert $a to a date-time value we can use this command:  $a = [datetime] $a  When you run this command and check the data type for $a you should get the following:  IsPublic IsSerial Name  -------- -------- ----  True True DateTime |
| CDbl | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Double**.  The double data type “contains a double-precision, floating-point number in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.”  Whew.  If it turns out you need a double-precision, floating-point number like that you can use Windows PowerShell to convert a variable to the double data type. The following two commands assign a string value to the variable $a, then convert $a to the double data type:  $a = "11.45"  $a = [double] $a  If you run these two commands and then use the GetType() method against $a you should get back the following information:  IsPublic IsSerial Name  -------- -------- ----  True True Double |
| Chr | **Definition:** Returns the character associated with the specified ANSI character code.  Here’s a question for you: Suppose you have an ASCII value and you need to convert that value to a real character; how are you supposed to do *that*?  Well, one way is to simply take the integer value (ASCII values are always integer values) and convert its data type to character. This command converts ASCII 34 to a character value and then stores that value in the variable $a:  $a = [char]34  When you run this command and then echo back the value of $a you should get the following:  " |
| CInt | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Integer**.  Need to convert a value to the integer data type? No problem: not only will Windows PowerShell make the conversion for you, but it will also round the value off to the nearest whole number. (Some languages convert a value to an integer simply by stripping off the decimal point.) The following two commands assign a string value to the variable $a, then convert the data type of $a to integer:  $a = "11.57"  $a = [int] $a  When you run these two commands and then echo back the value of $a you should get the following:  12  And if you run the GetType() method against $a you’ll see it now has the following data type:  IsPublic IsSerial Name  -------- -------- ----  True True Int32 |
| CLng | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Long**.  “Long” values are integers in the range -2,147,483,648 to 2,147,483,647. The following two commands show how you can convert a value to the long data type. In command no. 1 we assign a value to the variable $a; in command no. 2 we then convert $a to a long value:  $a = "123456789.45"  $a = [long] $a  When you run this command and then echo back the value of $a you should get the following:  123456789  **Note**. What happened to the .45 on the end of our original value? Well, remember, long values must be *integers*. Therefore Windows PowerShell rounds a value to the nearest whole number when converting that value to the long data type. |
| Cos | **Definition:** Returns the cosine of an angle.  Yes, we know: in a million years you'll never *once* have to calculate the cosine of an angle in one of your scripts. Still, better safe than sorry, right? In Windows PowerShell you can determine the cosine of an angle using the System.Math class and the **Cos** method. This sample command returns the cosine of a 45-degree angle and then stores the value in a variable named $a:  $a = [math]::cos(45)  When you run this command and then echo back the value of $a you should get the following:  0.52532198881773 |
| CreateObject | **Definition:** Creates and returns a reference to an Automation object.  In Windows PowerShell you create new COM objects by using the **New-Object** Cmdlet. To do so call New-Object, passing the Cmdlet the **–comobject** parameter, which tells Windows PowerShell to create a new COM object rather than, say, a new .NET Framework object. The –comobject parameter is then followed by the ProgID of the COM object.  For example, the following two commands create an instance of Microsoft Excel and then, just to prove that the first command *did* create an instance of Excel, the second command makes the application visible onscreen:  $a = new-object -comobject Excel.Application  $a.visible = $True  Incidentally, when using the New-Object Cmdlet you might want to add the **–strict** parameter, like so:  $a = new-object -comobject Excel.Application -strict  That helps ensure that you are working with a true COM object and not a COM wrapper around a .NET object. |
| CSng | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **Single**.  If you’ve read this conversion guide from the very beginning then you already know way more about data types than you probably wanted to know. Nevertheless, here’s yet another fact about data types: a single value is “a single-precision, floating-point number in the range -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.” Can you convert a variable to the single data type? The following two commands argue that you can. In the first command we assign a value to the variable $a; in the second command we convert $a to the single data type:  $a = "11.45"  $a = [single] $a  When you run these commands and then use the GetType() method to return the data type for $a you should get the following:  IsPublic IsSerial Name  -------- -------- ----  True True Single |
| CStr | **Definition:** Returns an expression that has been converted to a **Variant** of subtype **String**.  There will be times (especially when you work with databases) when you’ll have a numeric value that needs to be treated as a string. Can you convert a numeric value into a string value? Of course you can. To begin with, assign the numeric value to a variable (in this example, we assign 17 to $a):  $a = 17  If we want to verify that $a is really a numeric value we can do this using the **GetType()** method:  IsPublic IsSerial Name  -------- -------- ----  True True Int32  Now, what about converting this to a string value? Here you go:  $a = [string] $a  When you run this command and then check the data type of $a you should get the following:  IsPublic IsSerial Name  -------- -------- ----  True True String |
| Date | **Definition:** Returns the current system date.  In VBScript you can use the Date function to assign the current date (and only the current date, not the current time as well) to a variable. In Windows PowerShell you can do the same thing by calling the **Get-Date** Cmdlet and using the **–format** **d** (for date) parameter. For example, this command assigns the current date to the variable $a:  $a = get-date –format d  When you run this command and then echo back the value of $a you should back something similar to this:  9/1/2006 |
| DateAdd | **Definition:** Returns a date to which a specified time interval has been added.  No scripting language (or at least none that we know of) can predict the future. However, many of them at least enable you to predict when the future will occur. For example, suppose today is 9/21/2006. What will the date be 37 days from now? In Windows PowerShell you can determine that by using the **Get-Date** Cmdlet along with the appropriate method. For example, this command calculates the date 37 days from the current date (using the **AddDays()** method) and stores that value in the variable $a  $a = (get-date).AddDays(37)  If this command is run on 9/21/2006 you should get the following when you echo back the value of $a:  Saturday, October 28, 2006 11:33:27 AM  Of course, you aren’t limited to working only with the AddDays() method. Here are other date arithmetic methods available to you:  (get-date).AddHours(37)  (get-date).AddMilliseconds(37)  (get-date).AddMinutes(37)  (get-date).AddMonths(37)  (get-date).AddSeconds(37)  (get-date).AddTicks(37)  (get-date).AddYears(37)  You can also combine these methods in a single command. This command calculates the time 2 hours and 34 minutes from now. To do that, it first uses **AddHours()** to determine the time 2 hours from now, then takes the resulting value and uses the **AddMinutes()** method to determine the time 34 minutes from *then*. (Note the use of parentheses, which ensures that AddHours() is run first and AddMinutes() is run only when AddHours() is done.) Here’s the command:  $a = ((get-date).AddHours(2)).AddMinutes(34) |
| DateDiff | **Definition:** Returns the number of intervals between two dates.  The VBScript DateDiff function is used to determine the amount of time (months, weeks, days, hours, etc.) between two date-time values. In Windows PowerShell you can perform this same kind of date arithmetic using the **New-TimeSpan** Cmdlet.  For example, the following command calculates the amount of time between the current date and time and 11:30 PM on December 31, 2006. Note that the **Get-Date** Cmdlet is used to create the two date-time values. Also note that a dollar sign ($) is used to preface each of the Get-Date commands, and each command is enclosed in parentheses. That ensures that Windows PowerShell first calculates the two date-time values, and only *then* uses New-TimeSpan to determine the time interval.  The command itself looks like this:  $a = New-TimeSpan $(Get-Date) $(Get-Date –month 12 -day 31 -year 2006 -hour 23 -minute 30)  When you run this command and then echo back the value of $a you should get the following (depending, of course, on the current date and time):  Days : 109  Hours : 3  Minutes : 55  Seconds : 0  Milliseconds : 0  Ticks : 94317000000000  TotalDays : 109.163194444444  TotalHours : 2619.91666666667  TotalMinutes : 157195  TotalSeconds : 9431700  TotalMilliseconds : 9431700000  Suppose all you really care about is the number of days between the two dates. In that case, just echo back the value of the **Days** property:  $a.Days |
| DatePart | **Definition:** Returns the specified part of a given date.    Given a date-time value the DatePart function can tease out just a portion of that value, such as the hour, minute, or second of the day. In Windows PowerShell the **Get-Date** Cmdlet provides the same capability: just call Get-Date and take a peek at the desired property value. Here are some examples:  $a = (get-date).day  $a = (get-date).dayofweek  $a = (get-date).dayofyear  $a = (get-date).hour  $a = (get-date).millisecond  $a = (get-date).minute  $a = (get-date).month  $a = (get-date).second  $a = (get-date).timeofday  $a = (get-date).year  Suppose you *do* need to know just the hour of the day. No problem; first, use this command to grab the value of the **Hour** property and store it in the variable $a:  $a = (get-date).hour  And then simply echo back the value of $a. Depending on the time of day when you ran the command, you should get back something like this (based on a 24-hour clock):  15 |
| DateSerial | **Definition:** Returns a **Variant** of subtype **Date** for a specified year, month, and day.  OK, we’re not sure how often (if ever) this one comes up, but the DateSerial function in VBScript enables you to create a date just by passing the appropriate values for the year, month, and day:  MyDate1 = DateSerial(2006, 12, 31)  (Hey, we said we didn’t know how often this comes up.)  You can achieve roughly the same effect in Windows PowerShell by calling the **Get-Date** Cmdlet and passing the appropriate values:  $a = get-date -y 2006 -mo 12 -day 31  Notice that, in Windows PowerShell, we need to include parameters for the year (**-y**), the month (**-mo**), and the day (**-day**). We can, however, abbreviate the year and the month to make the finished product more closely mimic the DateSerial function.  When you run this command and then echo back the value of $a you should get something similar to the following:  Sunday, December 31, 2006 1:27:42 PM |
| DateValue | **Definition:** Returns a **Variant** of subtype **Date**.  How can you make sure Windows PowerShell treats a value as a date and not as, say, a string? One way is to explicitly “cast” that value as a date-time value. For example, this command assigns the string 12/1/2006 to the variable $a, at the same time making $a a date-time value:  $a = [datetime] "12/1/2006"  When you run this command and then use the GetType() method to retrieve the data type for $a you should get the following:  IsPublic IsSerial Name  -------- -------- ----  True True DateTime |
| Day | **Definition:** Returns a whole number between 1 and 31, inclusive, representing the day of the month.    So, in conclusion – yes, did you have a question? You say you have an automated procedure that’s supposed to run only on the 10th and 25th of each month? You say you don't know how to use Windows PowerShell to determine the day portion of the month? Hey, no problem; this is an easy one to answer: all you have to do is call the **Get-Date** Cmdlet, then grab the value of the **Day** property. For example, this command retrieves the value of the Day property for the current date and stores that value in the variable $a:  $a = (get-date).day  When you run this command and then echo back the value of $a you should get the following, assuming you’re running the command on, say, 9/17/2006:  17 |
| Escape | **Definition:** Encodes a string so it contains only ASCII characters.  Admittedly, this is another one of those functions you probably don’t use on a daily basis. To quote from the VBScript Language reference:  “The **Escape** function returns a string (in Unicode format) that contains the contents of charString. All spaces, punctuation, accented characters, and other non-ASCII characters are replaced with **%***xx* encoding, where xx is equivalent to the hexadecimal number representing the character. Unicode characters that have a value greater than 255 are stored using the **%u***xxxx* format.”  Hey, why not?  Shockingly, Windows PowerShell doesn’t include a built-in method for encoding a string in this fashion. However, you can easily do this by loading the .NET Framework **System.Web** class and then using the **Web.Utility** class’ **URLEncode** method. In other words, by executing a pair of commands similar to this:  [Reflection.Assembly]::LoadWithPartialName("System.Web")  $a = [web.httputility]::urlencode("http://www.microsoft.com/technet/scriptcenter/default.mspx")  When you run this command and then echo back the value of $a you should get the following:  http%3a%2f%2fwww.microsoft.com%2ftechnet%2fscriptcenter%2fdefault.mspx  And yes, that’s what it’s *supposed* to look like. |
| Eval | **Definition:** Evaluates an expression and returns the result.  OK, we’re not sure how often you need to evaluate an expression (such as a mathematical equation) and simply know whether that expression is true or false. If you *do* need to do this, however, you can do so simply by using the **–eq** comparison operator. For example, this command evaluates the expression  *2 + 2 = 45* and then stores the evaluation (true or false) in the variable $a:  $a = 2 + 2 -eq 45  When you run this command and then echo back the value of $a you should get the following:  False |
| Exp | **Definition:** Returns *e* (the base of natural logarithms) raised to a power.  You say your company will go broke unless you can find a way to raise e to a specified power? Relax; in Windows PowerShell you can do this by using the System.Math class and the **Exp** method. Here’s an example that raises e to the second power and assigns the resulting value to the variable $a:    $a = [math]::exp(2)  When you run this command and then echo back the value of $a you should get the following:  7.38905609893065 |
| Filter | **Definition:** Returns a zero-based array containing a subset of a string array based on specified filter criteria.  To tell you the truth, we’ve never actually seen anyone use the Filter function; however, we can see where it *might* be useful. Given an array, Filter enables you to do a wildcard search of items within that array. For example, suppose you have an array with the following items:   * Monday * Month * Merry * Mansion * Modest   Using the Filter function and the wildcard value *mon\** would return a sub-array containing all the values that start with the letters m-o-n:   * Monday * Month   In Windows PowerShell you create a similar substring by using the **Where-Object** Cmdlet and weeding out only those values that start with *mon* (note that you use the **–like** operator to do a wildcard search like that). The first of the following two commands creates an array named $a and assigns five string values to that array. The second command takes $a and pipes the values to Where-Object, which selects all the values that start with *mon* and assigns them to the variable $b:  $a = "Monday","Month","Merry","Mansion","Modest"  $b = ($a | where-object {$\_ -like "Mon\*"})  When you run this command and then echo back the value of $b you should get the following:  Monday  Month |
| FormatCurrency | **Definition:** Returns an expression formatted as a currency value using the currency symbol defined in the system Control Panel.  To format a Windows PowerShell value as currency you simply use the .NET Framework formatting commands. The following two commands assign the value 1000 to the variable $a, then use the currency formatting string **“{0:C}”** to format the value as currency (note that the value to be formatted is included as part of the **–f** parameter):  $a = 1000  $a = "{0:C}" -f $a  When you run this command and then echo back the value of $a you should get the following:  $1,000.00 |
| FormatDateTime | **Definition:** Returns an expression formatted as a date or time.  The .NET Framework enables you to create a vast array of custom date formats, far too many to cover in this introductory guide. Fortunately, you can create the four basic date-time formats without having to use any fancy (and sometimes confusing) custom formatting strings. Instead, just employ the four methods shown below:  $a = (get-date).tolongdatestring()  $a = (get-date).toshortdatestring()  $a = (get-date).tolongtimestring()  $a = (get-date).toshorttimestring()  When you run these commands and then echo back the values of $a you should get something similar to the following, in order:  Wednesday, September 13, 2006  9/13/2006  7:57:25 PM  7:57 PM |
| FormatNumber | **Definition:** Returns an expression formatted as a number.  For most people, formatting a number simply means specifying the number of decimal points to be displayed. In Windows PowerShell you can use .NET formatting strings to specify the number of decimal places. The following example assigns the value 11 to the variable $a, then uses the .NET formatting string **"{0:N6}"** to format $a so that it displays 6 digits beyond the decimal point (that's what the *6* in the formatting string is for):  $a = 11  $a = "{0:N6}" -f $a  When you run this command and then echo back the value of $a you should get the following:  11.000000  **Note**. Windows PowerShell will round numbers up or down as needed to fit the new format. For example, suppose $a is equal to 11.54543 and you decide to format it with just a single decimal point. Windows PowerShell will assign the value 11.5 to $a (because the discarded decimal digits – 4543 – are rounded down to 5).  Incidentally, this formatting command will also insert the appropriate list separator. For example, try running these two commands and see what you get back  $a = 11000000  $a = "{0:N6}" -f $a  When you echo back the value of $a (at least on a machine using the default setting for US English) you’ll get back the following value:  11,000,000.000000 |
| FormatPercent | **Definition:** Returns an expression formatted as a percentage (multiplied by 100) with a trailing % character.  In Windows PowerShell you can format a value as a percent by using the .NET formatting methods. For example, these two commands set the value of $a to .113, then use .NET formatting to format $a as a percentage:  $a = .113  $a = "{0:P1}" -f $a  The formatting command is interpreted like this: you specify the format type in brackets – {} – with the entire method (brackets and all) enclosed in double quote marks. The **0** is the index number of the item to be formatted (in this case it’s a 0 because we're dealing with a single string value). The **P** indicates that we want to format the value as a percentage, and the **1** represents the number of digits to display following the decimal point.  Got all that? Excellent.  The format method is then followed by the **–f** parameter and the value to be formatted ($a).  When you run this command and then echo back the value of $a you should get the following:  11.3 % |
| GetLocale | **Definition:** Returns the current locale ID value.  InVBScript you can identify the user's locale (which determines such things as keyboard layout and alphabetic sort order, as well as date, time, number, and currency formats) by calling the GetLocale function and then translating the returned ID number. You can get this same locale ID in Windows PowerShell by using the **Get-Culture** Cmdlet and looking at the value of the **LCID** property:  $a = (get-culture).lcid  When you run this command and then echo back the value of $a you should get the following, provided your computer has US English configured for its language and regional settings:  1033  If you’d prefer to see the locale name (as opposed to the ID number) use this command instead:  $a = (get-culture).displayname  That will return information similar to this:  English (United States) |
| GetObject | **Definition:** Returns a reference to an Automation object from a file.  OK, we’ll level with you: we’re taking the easy way out on this one. Although Windows PowerShell does not include a GetObject command, it *is* possible to load the Microsoft.VisualBasic assembly and then use the **Interaction** class and *its* **GetObject** method:  [reflection.assembly]::LoadWithPartialName("'Microsoft.VisualBasic")  $a= [Microsoft.VisualBasic.Interaction]::GetObject("WinNT://atl-ws-01/Administrator")  That part’s easy. To actually *use* this object reference is far more complicated, requiring the use of binding flags and the InvokeMember method, and, well, it goes way beyond what we can cover in this introductory manual. The best thing to do? Assume that there is no GetObject equivalent and forget about it. The second best thing to do: Check [**this Web site**](http://mow001.blogspot.com/2006/04/access-adsi-winnt-provider-from-monad.html) for an example of using the Visual Basic .NET GetObject method from within Windows PowerShell. |
| GetRef | **Definition:** Returns a reference to a procedure that can be bound to an event.  Not applicable. As far as we know, this is a Web page-only command. |
| Hex | **Definition:** Returns a string representing the hexadecimal value of a number.  How do you convert a decimal number to a hexadecimal value? Why, you use the .NET formatting methods, of course. (How else would you do it?) For example, these two commands assign the value 4517 to the variable $a, then use .NET formatting to return the hexadecimal equivalent of 4517:  $a = 4517  $a = "{0:X}" -f $a  The formatting command is interpreted like this: you specify the format type in brackets – {} – with the entire method (brackets and all) enclosed in double quote marks. The **0** is the index number of the item to be formatted (in this case 0, because we're dealing with a single string value). The **X** indicates that we want to format the value as a hexadecimal number.  The format method is then followed by the **–f** parameter and the value to be formatted ($a).  When you run this command and then echo back the value of $a you should get the following:  11A5 |
| Hour | **Definition:** Returns a whole number between 0 and 23, inclusive, representing the hour of the day.  Not that days, months, and years aren't important, mind you, but sometimes all you really need to know is the hour. To return an integer value indicating the hour (based on a 24-hour clock) just use the **Get-Date** Cmdlet and take a peek at the value of the **Hour** property:    $a = (get-date).hour  When you run this command at 4:00 PM and then echo back the value of $a you should get the following:  16 |
| InputBox | **Definition:** Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns the contents of the text box.  OK, this was a tough one. At first we figured that the easiest way to create an input box would be to rely on the .NET Framework and Visual Basic .NET. That worked, but with one major problem: the input box always appeared behind all the other windows on the desktop. That meant that the input box would appear, but no one would ever see it. Not exactly what we were hoping for.  At our wit’s end, we did what anyone else would do in our situation: we started searching the Internet looking for sample code we could steal. And, sure enough, we found some, courtesy of /\/\o\/\/, a Windows PowerShell MVP. (Check out his [**Web site and blog**](http://mow001.blogspot.com/2006/03/vbscript-hosting-in-msh-inputbox-in.html); when it comes to Windows PowerShell, this guy knows what he’s talking about.) Here’s the solution /\/\o\/\/ came up with:  $a = new-object -comobject MSScriptControl.ScriptControl  $a.language = "vbscript"  $a.addcode("function getInput() getInput = inputbox(`"Message box prompt`",`"Message Box Title`") end function" )  $b = $a.eval("getInput")  We won’t bother trying to explain how this works; we’ll simply note that it *does* work. We’ll also point out that you have to type this exactly as shown; in particular, that means including all the little backticks (the ` character) when assigning a message box prompt and title.  By the way: thanks, /\/\o\/\/! |
| InStr | **Definition:** Returns the position of the first occurrence of one string within another.  The InStr function is primarily used to determine whether a specific character or set of characters can be found in a string. In Windows PowerShell you can test for the existence of a substring by using the **Contains()** method. For example, here we assign the word *wombat* to a variable named $a, then use Contains() to determine whether or not the letter *m* appears anywhere in that string. In turn, our True/False return value is stored in the variable $b:  $a = "wombat"  $b = $a.contains("m")  When you run this command and then echo back the value of $b you should get the following:  True  What if we used *this* command:  $b = $a.contains("x")  That’s right: Windows PowerShell would report back **False**, because there is no *x* in wombat.  And no *i* in team.  Confession time: The InStr function is typically used to determine whether a specified substring exists in a parent string; most of the time script writers only care whether or not the substring can be found. Technically, though, InStr doesn’t return a Boolean value (that is, True/False); instead, it returns the character position of the first instance of the substring. If that’s what you’d *really* like to do you can use the **IndexOf()** method instead of the Contains()method. For example, consider these two lines of code:  $a = "wombat"  $b = $a.indexof("m")  If you run these two lines of code and then echo back the value of $b, you’ll get back this:  2  Why? Because the letter *m* can first be found in the third character position in *wombat*. (Yes, we know. But the first character occupies position 0, meaning the second character occupies position 1 and the third character occupies position 2). If the letter *m* could *not* be found in the string then a -1 would be returned. |
| InStrRev | **Definition:** Returns the position of an occurrence of one string within another, from the end of string.  To the best of our knowledge, no Scripting Guy has ever used the InStrRev function. (Or, if they have, they won’t admit it.) What InStrRev does is examine a string value and then determine the *last* occurrence of a specified substring.  We don’t blame you. But here’s an example. Suppose you run InStrRev against the string value *1234x6789x1234*, specifying the letter *x* as the substring you’re searching for. In that case InStrRev will return a 10, because the last *x* happens to be the 10th character in the string.  OK. Now, what if you want to perform this same trick in Windows PowerShell? No problem: just use the **LastIndexOfAny** method, like so:  $a = "1234x6789x1234"  $b = $a.lastindexofany("x")  When you run those two commands and then echo back the value of $b you should get the following:  9  And, no, that’s not a mistake: in .NET the first character position in a string is actually position 0; that makes the 10th character (the character we're interested in) position 9. (A good thing to keep in mind if and when you start doing string manipulation in Windows PowerShell.) |
| Int/Fix | **Definition:** Returns the integer portion of a number.  Friends, are you plagued by decimal points? Would you just as soon strip off those pesky little decimal values and leave yourself with a clean, pure integer value? Then have we got a deal for you. In Windows PowerShell you can remove decimal places by using the System.Math class and the **Truncate** method. For example, here we assign the value 11.98 to $a, then use Truncate to remove the .98:  $a = 11.98  $a = [math]::truncate($a)  When you run this command and then echo back the value of $a you should get the following:  11  **Note**. Keep in mind that the truncate method simply removes all numbers following the decimal point; it does not round values to the nearest whole number. If you want to do that, use the **Round** method instead. |
| IsArray | **Definition:** Returns a Boolean value indicating whether a variable is an array.  When it comes to working with arrays, Windows PowerShell is a bit more forgiving than VBScript. For example, VBScript will blow up with a “Type mismatch” error should you try to directly echo the value of an array; instead, you need to set up a For Each loop to get at those values (or specify individual values one at a time). With Windows PowerShell you don’t have to worry about that; a simple command such as this will echo back all the values in the array $a, no For Each loop required:  $a  That said, there will still be times when you’ll find it useful to know, in advance, whether or not you are dealing with an array. In Windows PowerShell that can be done by using the **–is** operator and then checking to see if the item has an array data type. In these two lines of code, we create an array named $a, then check to see whether or not we really *do* have an array. The return value (True or False) is then stored in the variable $b:  $a = 22,5,10,8,12,9,80  $b = $a -is [array]  When you run this command and then echo back the value of $b you should get the following:  True |
| IsDate | **Definition:** Returns a Boolean value indicating whether an expression can be converted to a date.  Windows PowerShell makes it easy to format and manipulate date and time values … provided, of course, that you actually *have* date and time values. To verify that a value truly *is* a date-time value all you have to do is use the **–is** operator and check to see if the data type is datetime. For example, these two lines of code assign a value to the variable $a and then determine whether or not $a is a date-time value:  $a = 11/2/2006  $a -is [datetime]  When you run this command you should get the following:  False  **Note**. Why is this False; why isn’t 11/2/2006 a valid date? That’s easy: to assign a date to a variable you need to enclose that date in double quote marks and specify the [datetime] variable type:  $a = [datetime] "11/2/2006"  Without the double quote marks Windows PowerShell believes that this is a mathematical expression: 11 divided by 2 divided by 2006. In fact, if you check the value of $a you’ll get back this:  0.00274177467597208  With the quotes but without the [datetime] specified, Windows PowerShell thinks this is a string and still returns False. |
| IsEmpty | **Definition:** Returns a Boolean value indicating whether a variable has been initialized.  Rather than get bogged down in a long discussion on metaphysics, let’s go with a rudimentary explanation of the difference between an empty variable and a null variable. A null variable is a value we know nothing about; we can't even say for certain whether the variable *has* a value. By contrast, we know the value of an empty variable: it's nothing. An empty variable is a variable that has no value (for example, it's been assigned an empty string as its value). A null variable – well, like we said, we don't know anything at all about a null variable.  So how do we know whether or not a variable is empty? Well, for now about the only approach we’ve come up with is to check and see whether or not the variable has a **Length** equal to 0. An empty variable has a length equal to 0; a null variable does not. (Because we don't know anything about a null variable, we have no idea what its length might be.)  Confused? We don’t blame you. But maybe this example will help. In our first command, we assign an empty string ("") to the variable $a; in the second command, we use the **–eq** operator to determine whether or not the Length of $a is equal to 0. That value gets stored in the variable $b:  $a = ""  $b = $a.length -eq 0  When you run this command and then echo back the value of $b you should get the following:  True  We can’t guarantee that this will always work, but it’s a good place to start. |
| IsNull | **Definition:** Returns a Boolean value that indicates whether an expression contains no valid data (Null).  It’s often useful to know whether or not a value is Null; scripters who have worked with Active Directory have often run into problems when trying to do something as seemingly-trivial as echo back the value of a property, at least if that property turns out not to *have* a value. In Windows PowerShell you can check for a Null value simply by using the **-eq** comparison operator to compare a variable with the system variable $Null. For example, this command compares $b to $Null, and stores the results of that comparison in $a:  $a = $z -eq $null  When you run this command and then echo back the value of $a you should get the following, assuming, of course, that $z really *is* Null:  True  If you then use a similar command to determine whether or not $a is Null (i.e., $a –eq $null) you should get back **False**; that’s because $a *isn’t* Null. |
| IsNumeric | **Definition:** Returns a Boolean value indicating whether an expression can be evaluated as a number.  OK, so this one turned out to be a bit more complicated than we anticipated; that’s because neither Windows PowerShell nor the .NET Framework include a “generic” method for testing whether or not a given value is a number. After toying with the notion of doing this using regular expressions we finally decided to use Visual Basic .NET instead. In the following set of commands we assign the value 44.5 to the variable $a, then load the **Microsoft.VisualBasic** assembly. With the assembly loaded we can then use the Visual Basic **IsNumeric** function to determine whether or not 44.5 is a number::  $a = 44.5  [reflection.assembly]::LoadWithPartialName("'Microsoft.VisualBasic")  $b = [Microsoft.VisualBasic.Information]::isnumeric($a)  As you might expect, when we run these commands and then echo back the value of $b we get the following:  True  If we change the value of $a to “44.5a” and re-run the commands we’ll get back this:  False |
| IsObject | **Definition:** Returns a Boolean value indicating whether an expression references a valid Automation object.  That’s a good question: how *can* you tell whether or not a variable is an object reference; that is, how can you tell whether a variable is a pointer to a COM object or a .NET object? Well, one way is to invoke the **–is** parameter and see if the variable really *is* an object. For example, in the following two commands we create an object reference named $a. We then check to see whether or not $a is an object reference, with the resulting value (True or False) stored in the variable $b:  $a = new-object -comobject scripting.filesystemobject  $b = $a -is [object]  When you run this command and then echo back the value of $b you should get the following:  True  As we intimated, .NET objects will also be identified as objects using this command. |
| Join | **Definition:** Returns a string created by joining a number of substrings contained in an array.  There will be times (trust us, there will be) when you want to take all the values in an array and transform them into a single string. In Windows PowerShell you can do that using the System.String class and the **Join** method. For example, in the first of the following two commands we assign the letters h-e-l-l-o to an array variable named $a; in the second command we use the Join method to combine those values into a string variable named $b. Notice that we pass Join two parameters: the separator (that is, the character we want inserted between each array item) and the array to be joined. In this example we don’t want *any* character inserted between items, so we simply pass an empty string:  $a = "h","e","l","l","o"  $b = [string]::join("", $a)  When you run this command and then echo back the value of $b you should get the following:  hello  What if we wanted to place a \ between each item? Then we’d use this command:  $b = [string]::join("\", $a)  In turn, the value of $b would be equal to this:  h\e\l\l\o  Cool, huh? |
| LBound | **Definition:** Returns the smallest available subscript for the indicated dimension of an array.  Typically the lower bound of an array (that is, the index number assigned to the first element in the array) is 0. In Windows PowerShell you can verify that by using the **GetLowerBound()** method. For example, suppose we use the following command to create an array named $a:  $a = 1,2,3,4,5,6,7,8,9  We can then use this command to determine the lower bound of the array and store that value in the variable $b:  $b = $a.getlowerbound(0)  **Note**. The parameter **0** passed to GetLowerBound() simply means that we want to look at the first dimension in the array. If we were working with a multi-dimensional array we could supply other values to GetLowerBound().  When you run this command and then echo back the value of $b you should get the following:  0 |
| LCase | **Definition:** Returns a string that has been converted to lowercase.  Using Windows PowerShell, can you convert all the characters in a string to their lowercase equivalent? Of course you can; all you have to do is call the **ToLower()** method. For example, here we assign 26 uppercase letters to the variable $a; we then use the ToLower() method to convert all the characters in the string to lowercase:  $a = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  $a = $a.ToLower()  When you run this command and then echo back the value of $a you should get the following:  abcdefghijklmnopqrstuvwxyz |
| Left | **Definition:** Returns a specified number of characters from the left side of a string.  What’s that? You want to know if it’s possible to use Windows PowerShell to return the first *x* characters in a string? Of course it is: all you have to do is call the **Substring** method, passing two parameters: the starting character (with 0 being the first character position) and the number of characters to return. For example, this command assigns the letters of the alphabet to a variable named $a, then uses Substring to reassign $a just the first three letters in the string:  $a="ABCDEFGHIJKLMNOPQRSTUVWXYZ"  $a = $a.substring(0,3)  When you run this command and then echo back the value of $a you should get the following:  ABC |
| Len | **Definition:** Returns the number of characters in a string or the number of bytes required to store a variable.  The “length” of a string is simply the number of characters contained in that string. In Windows PowerShell you can determine the length of a string by retrieving the value of the aptly-named **Length** property. In this example we stored the 26 letters of the English alphabet in a variable named $a, then assign the length of that string to the variable $b:  $a = "abcdefghijklmnopqrstuvwxyz"  $b = $a.length  When you run this command and then echo back the value of $b you should get the following:  26 |
| LoadPicture | **Definition:** Returns a picture object.  Not applicable. This command is designed for use with Web pages. And, to the best of our understanding, it doesn’t work the way it’s supposed to anyway. |
| Log | **Definition:** Returns the natural logarithm of a number.  Most people probably have the natural logarithms for all known numbers committed to memory. If you don't, and you need to know the log for a particular number, then use the System.Math class and the **Log** method. This command returns the log of 100 and assigns the value to the variable $a:  $a = [math]::log(100)  When you run this command and then echo back the value of $a you should get the following:  4.60517018598809 |
| LTrim | **Definition:** Returns a copy of a string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).  In VBScript the LTrim function is used to remove any blank spaces that appear at the *beginning* of a string; in Windows PowerShell, you can carry out this same task by using the **TrimStart()** method. For example, these two commands assign a string value to the variable $a, then use the TrimStart() method to remove the blank spaces from the beginning of the string (note that, for illustration purposes, we’ve used dots to represent blank spaces):  $a = "..........123456789.........."  $a = $a.TrimStart()  When you run this command and then echo back the value of $a you should get the following:  123456789.......... |
| RTrim | **Definition:** Returns a copy of a string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).  In VBScript the RTrim function is used to remove any blank spaces that appear at the *end* of a string; in Windows PowerShell, you can carry out this same task by using the **TrimEnd()** method. For example, these two commands assign a string value to the variable $a, then use the TrimEnd() method to remove the blank spaces from the end of the string (note that, for illustration purposes, we’ve used dots to represent blank spaces):  $a = "..........123456789.........."  $a = $a.TrimEnd()  When you run this command and then echo back the value of $a you should get the following:  ..........123456789 |
| Trim | **Definition:** Returns a copy of a string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).  You had to know that this one was coming: In VBScript the Trim function is used to remove any blank spaces that appear at the beginning or the end of a string; in Windows PowerShell, you can carry out this same task by using the **Trim()** method. For example, these two commands assign a string value to the variable $a, then use the Trim() method to remove the blank spaces from both the beginning and the end of the string (note that, for illustration purposes, we’ve used dots to represent blank spaces):  $a = "..........123456789.........."  $a = $a.Trim()  When you run this command and then echo back the value of $a you should get the following:  123456789 |
| Mid | **Definition:** Returns a specified number of characters from a string.  Sometimes the good stuff – at least when it comes to Oreo cookies and string values – is found in the middle. With Oreos you can get to the stuff in the middle by unscrewing the cookie; with string values you can get to the stuff in the middle by calling the **Substring()** method. In the following example we assign a string value to the variable $a and then use the SubString() method to retrieve a portion of that string. And what portion are we retrieving? To answer that, let’s first look at the two commands:  $a="ABCDEFG"  $a = $a.substring(2,3)  Notice that we’re passing Substring() two parameters: a 2 and a 3. The 2 represents the character position in the main string where we want to start grabbing letters. We want to start with the third character, so we pass a 2. No, that’s not a typo: we pass a 2 because the first character in the string is in position 0. That means the second character is found in position 1, and the third character – the one we’re interested in – resides in position 2.  The 3, meanwhile, indicates the number of characters we want to extract. In other words, we want to start at position 2 and grab the next three characters.  Pretty simple when you think about it.  When you run this command and then echo back the value of $a you should get the following:  CDE |
| Minute | **Definition:** Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.  Wow: you need to know the exact *minute* for a specified date-time value? No problem: just use the **Get-Date** Cmdlet and grab the value of the **Minute** property:  $a =(get-date).minute  When you run this command and then echo back the value of $a you should get the following (depending on the actual time, of course):  24 |
| Month | **Definition:** Returns a whole number between 1 and 12, inclusive, representing the month of the year.  Need to get the number representing the month for a specified date (e.g., 1 for January, 2 for February, etc.)? Then just use the **Get-Date** Cmdlet, the **–f** parameter and the **MM** .NET formatting command. This command returns a numeric value representing the month of the year for the current date:  $a = get-date -f "MM"  When you run this command (at least during the month of September) and then echo back the value of $a you should get the following:  09  If you don’t want the leading zero, then retrieve the month value and change it to an integer data type. How do you do that? Like this:  $a = [int] (get-date -f "MM") |
| MonthName | **Definition:** Returns a string indicating the specified month.  Sometimes all you want to know is that we’re in month number 9; at other times you’d prefer to know the *name* of the month (e.g., September). In Windows PowerShell you can determine the name of the month by using the **Get-Date** Cmdlet and then using the **–f** formatting parameter and the value MMMM. In other words, this command will retrieve the name of the month for the current month and store that value in the variable $a:  $a = get-date -f "MMMM"  When you run this command and then echo back the value of $a you should get the following (depending on the actual month):  September |
| MsgBox | **Definition:** Displays a message in a dialog box, waits for the user to click a button, and returns a value indicating which button the user clicked.  Windows PowerShell does not posses built-in methods for displaying message boxes. However, it’s very easy to use the **New-Object** Cmdlet to create an instance of the Windows Script Host (WSH) Shell object; from there you can use the WSH Popup method to display a message box. The following example first uses New-Object (and the **–comobject** parameter) to create an instance of the Wscript.Shell object. In the second command, the WSH Popup method is used to display a message box, with the resulting action (that is, the value of the button the user clicked to dismiss the message box) stored in the variable $b:  $a = new-object -comobject wscript.shell  $b = $a.popup("This is a test",0,"Test Message Box",1)  For more information on the parameters used when displaying the message box see the [**Windows Script Host Language Reference**](http://msdn2.microsoft.com/en-us/library/x83z1d9f.aspx). |
| Now | **Definition:** Returns the current date and time according to the setting of your computer's system date and time.  If you’re looking for easy one then you came to the right place: assigning the current date and time to a variable is no more difficult than using the **Get-Date** Cmdlet. For example:  $a = get-date  When you run this command and then echo back the value of $a you should get something similar to the following:  Tuesday, September 05, 2006 10:24:04 AM |
| Oct | **Definition:** Returns a string representing the octal value of a number.  As script writers, a question we always ask ourselves is this: where would we be without octal numbers?  Well, OK, good point. Still, you never know: maybe someday you *will* have to use Windows PowerShell to convert a decimal value to its octal equivalent. If and when that time comes you can do this using the .NET Framework **Convert** class and the **ToString()** method. Just pass ToString() two parameters:   * The decimal value to be converted (in our sample command, 999). * The value **8** (the second parameter tells .NET to convert the value to its octal equivalent).   In other words, do something like this:  $a = [Convert]::ToString(999,8)  When you run this command and then echo back the value of $a you should get the following:  1747 |
| Replace | **Definition:** Returns a string in which a specified substring has been replaced with another substring a specified number of times.  The world as we know it would come to a crashing halt if we didn’t have search-and-replace commands. In Windows PowerShell you can do a simple search-and-replace on any string value simply by using the **–replace** parameter and specifying two things:   * The string value to search for. * The replacement text.   For example, the following two command assign a value to the variable $a, then use –replace to replace all instances of the letter *x* with the letter *a*:  $a = "bxnxnx"  $a = $a -replace("x","a")  When you run this command and then echo back the value of $a you should get the following:  banana |
| RGB | **Definition:** Returns a whole number representing an RGB color value.  If you are a Web designer you might very well find yourself needing to calculate RGB color values. Although Windows PowerShell doesn't include a built-in method or function for calculating RGB values you can write a simple equation that will do the math for you. In the following example, we assign values to three different variables, representing the three color components: $blue, $green, $red. We then calculate the RGB value and store the result in the variable $a:  $blue = 10  $green= 10  $red = 10  $a = [long] ($blue + ($green \* 256) + ($red \* 65536))  When you run these commands and then echo back the value of $a you should get the following:  657930 |
| Right | **Definition:** Returns a specified number of characters from the right side of a string.  You can use the **Substring()** method to retrieve a specified number of characters from the beginning of a string (see Left). That’s great, but how do you return a specified number of characters from the *end* of a string?  Here’s one sneaky way to do that. Use the Substring() method, specifying the following two parameters:   * The starting position, which should be the length of the string minus the number of characters you’re after. If you want the last 4 characters in the string then use the length minus 4. * The number of characters you want returned.   For example, these two commands assign a value to the variable $a, then use the Substring() method to retrieve the last 9 characters in the string:  $a = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  $a = $a.substring($a.length - 9, 9)  When you run this command and then echo back the value of $a you should get the following:  RSTUVWXYZ |
| Rnd | **Definition:** Returns a random number.  Although it doesn’t require a lot of code to generate a random number in VBScript, the code that *is* required is a bit cryptic, to say the least. In Windows PowerShell the process is much easier: all you need to do is create an instance of the System.Random class and then call the **Next** method, passing this method the first and last numbers in the series. For example, these two commands create an instance of System.Random and then store a random number between 1 and 100 in the variable $b  $a = new-object random  $b = $a.next(1,100)  When you run this command and then echo back the value of $b you should get something like the following:  72  And, of course, the next time you run the command you should get something different.  To generate a random integer (not necessarily an integer falling between a specified starting and ending number) just call the Next method without supplying any additional parameters:  $b = $a.next() |
| Round | **Definition:** Returns a number rounded to a specified number of decimal places.  When working with system administration scripts (and performing chores like returning free disk space on a drive) you’ll often get back numbers similar to this: 45.987654321. More often than not you don’t really need all those decimal places; instead, you’d just as soon round the value up (or down).  OK; in Windows PowerShell all you have to do is use the System.Math class and the **Round** method. This command rounds the number 45.987654321 to two decimal places (notice the value **2** passed as the second parameter to the method). The resulting value is then stored in the variable $a:  $a = [math]::round(45.987654321, 2)  When you run this command and then echo back the value of $a you should get the following:  45.99 |
| ScriptEngine | **Definition:** Returns a string representing the scripting language in use.  Not really applicable; after all, if you’re running Windows PowerShell you’re using the Windows PowerShell script engine. Still, you can use the **Get-Host** Cmdlet and the **Version** property to return information about *which* version of Windows PowerShell you’re running:  $a = (get-host).version  When you run this command and then echo back the value of $a you should get the following, depending on which build you are using:  Major Minor Build Revision  ----- ----- ----- --------  1 0 0 0 |
| ScriptEngineBuildVersion | **Definition:** Returns the build version number of the scripting engine in use.  You’re right about that: it *can* be useful to know which build of Windows PowerShell you happen to be running. Fortunately you can determine that simply by calling the **Get-Host** Cmdlet and then taking a peek at the value of the **Version.Build** property:  $a = (get-host).version.build  When you run this command and then echo back the value of $a you should get the following, depending on which build you are using:  0 |
| ScriptEngineMajorVersion | **Definition:** Returns the major version number of the scripting engine in use.  Determining the major version of Windows PowerShell itself is a snap: all you have to do is call the **Get-Host** Cmdlet and then grab the value of the **Version.Major** property. The following command determines the major version and then stores that value in the variable $a:  $a = (get-host).version.major  When you run this command and then echo back the value of $a you should get the following, depending on which version of PowerShell you are running:  1 |
| ScriptEngineMinorVersion | **Definition:** Returns the minor version number of the scripting engine in use.  As you might expect, if you can use Windows PowerShell to determine the major version of the product you can also use it to determine the minor version. In fact, all you have to do is call the **Get-Host** Cmdlet and take a look at the value of the **Version.Minor** property. This command returns the minor version and stores it in the variable $a:  $a = (get-host).version.minor  When you run this command and then echo back the value of $a you should get the following, depending on which version of PowerShell you are running:  0 |
| Second | **Definition:** Returns a whole number between 0 and 59, inclusive, representing the second of the minute.  Is it useful for you to be able to take a specified date and time and throw out everything but the seconds? In other words, if today is 12:29:16 on 9/30/2006, is it useful for you to be able to determine that the seconds are equal to 16?  To tell you the truth, we don’t know whether that’s useful or not. If it *is* useful, however, you can do that in Windows PowerShell by using the **Get-Date** Cmdlet and then examining the value of the **Second** property:  $a = (get-date).second  When you run this command (assuming it really *is* 12:29:16 on 9/30/2006) and then echo back the value of $a you should get the following:  16 |
| SetLocale | **Definition:** Sets the global locale and returns the previous locale.  To be honest, we didn’t put much effort into this one; after all, SetLocale works only in Web pages and Windows PowerShell is not designed for use in Web pages. So we let this one slide. |
| Sgn | **Definition:** Returns an integer indicating the sign of a number.  Quick: is that a negative number or a positive number? An easy way to verify that in Windows PowerShell is to use the System.Math class and the **Sign** method. For example, this command uses the Sign method to determine the sign of -453:  $a = [math]::sign(-453)  When you run this command and then echo back the value of $a you should get the following:  -1  If you check the sign of a positive number (e.g., 453) you should get back something like this (actually, you should get back something *exactly* like this):  1 |
| Sin | **Definition:** Returns the sine of an angle.  In mathematics the sine of an angle is defined as … well, it doesn’t matter what the definition is. Instead, all that matters is that, in Windows PowerShell, you can use the System.Math class and the **Sin** method. For example, this command calculates the sine of a 45-degree angle and then stores the result in the variable $a:  $a = [math]::sin(45)  When you run this command and then echo back the value of $a you should get the following:  0.850903524534118 |
| Space | **Definition:** Returns a string consisting of the specified number of spaces.  Sometimes it’s just enough that you can do something; it doesn’t matter if you have to use some sort of crazy approach to do it. For example, in VBScript you can use the Space function to create a string consisting of *x* number of consecutive blank spaces. Can you pull off this same feat in Windows PowerShell? You sure can: all you have to do is, well, take a blank space and multiply it by 25:  $a = " " \* 25  Sure it’s crazy. But, like we said, it works. Run the command, then use this command to add the letter *x* to the end of the string:  $a = $a + "x"  Guess what you’ll get if you now echo back the value of $a? That’s right, 25 blank spaces followed by the letter *x*:  x |
| Split | **Definition:** Returns a zero-based, one-dimensional array containing a specified number of substrings.  “Returns a zero-based, one-dimensional array containing a specified number of substrings.” Granted, that doesn’t sound too terribly interesting. Nevertheless, the VBScript Split function (or the Windows PowerShell **Split()** method) can be incredibly useful. For example, suppose you have a comma-delimited list of computer names. As a single string value that’s of minimal use; as an *array*, however, that opens up a whole world of possibilities, including the opportunity to loop through each item in the array and perform a task against each of those items (i.e., each of those computers).  So how do you turn a delimited string into an array? Here’s how:  $a = "atl-ws-01,atl-ws-02,atl-ws-03,atl-ws-04"  $b = $a.split(",")  In the first command, we simply assign a series of computer names (separated by commas) to the variable $a. In the second command, we use the Split method to separate the list of names and store them in an array named $b. Note the sole parameter passed to the Split() method: ",", which simply indicates that the comma is used as the delimiter (or, if you prefer, separator).  When you run this command and then echo back the value of $b you should get the following:  atl-ws-01  atl-ws-02  atl-ws-03  atl-ws-04 |
| Sqr | **Definition:** Returns the square root of a number.    Finally, a mathematical construct we all recognize! Granted, you probably need to calculate square roots about as often as you need to calculate arctangents. (On the other hand, and unlike arctangents, at least most of us know what square roots are.) Best of all, even if you *don't* know what square roots are you can still calculate them by using the System.Math class and the **Sqrt** method. For example, this command determines the square root of 144 and stores the result in the variable $a:  $a = [math]::sqrt(144)  When you run this command and then echo back the value of $a you should get the following:  12 |
| StrComp | **Definition:** Returns a value indicating the result of a string comparison.  Can we tell you whether two strings or identical? We can, provided you tell us one thing: do you want a case-sensitive comparison or a case-insensitive comparison?  Let’s try a case-insensitive comparison for starters. In the following set of commands we assign values to variables $a and $b. We then use the System.String class and the **Compare()** method to compare the two strings. Note that we pass the Compare() method three parameters: the two strings to be compared, and the value $True, which indicates that we want to do a case-insensitive comparison:  $a = "dog"  $b = "DOG"  $c = [String]::Compare($a,$b,$True)  When you run this command and then echo back the value of $c you should get the following:  0  **Note**. A 0 means that the strings are identical. Any other value means that the strings are *not* identical.  Of course, these strings are identical because we did a case-insensitive comparison. Suppose we use *this* command, which does a case –sensitive comparison:  $c = [String]::Compare($a,$b,$False)  This time you get back -1, meaning that the strings are *not* identical (because the letter cases are not the same). |
| String | **Definition:** Returns a repeating character string of the length specified.  The String function provides a quick, reasonably fail-safe way to create a string consisting of a specified number of repeating characters. In other words, a string similar to this:  "===================="  (Yeah, we know. But this is sometimes useful when formatting data for display in a command window.)  If you ever *do* need to string together 20 equal signs or what-have-you in Windows PowerShell, you can accomplish this task by multiplying (yes, we said multiplying) an equals sign by 20 (strange, but true):  $a = "=" \* 20  When you run this command and then echo back the value of $a you should get the following:  ==================== |
| StrReverse | **Definition:** Returns a string in which the character order of a specified string is reversed.  How many times have you said to yourself, “Gosh, if only there was a way for me to reverse the character order of a specified string”? That’s what we thought.  Shockingly, Windows PowerShell has no equivalent to VBScript’s StrReverse function; fortunately, though, we found at least one way to achieve the same effect. The following set of commands (which we won’t bother explaining in any detail) assigns the value *Scripting Guys* to the variable $a, then uses a For Next loop to reverse the order of the characters in the string:  $a = "Scripting Guys"  for ($i = $a.length - 1; $i -ge 0; $i--) {$b = $b + ($a.substring($i,1))}  When you run this command and then echo back the value of $b you should get the following:  syuG gnitpircS  **Note**. Believe it or not, that’s what you’re *supposed* to get back! |
| Tan | **Definition:** Returns the tangent of an angle.  Good point: Windows PowerShell wouldn’t be much of a scripting language if it allowed you to calculate arctangents but *didn't* allow you to calculate tangents. Fortunately, that's not the case: you can determine the tangent of an angle by using the System.Math class and calling the **Tan** method. Here’s a command that calculates the tangent of a 45-degree angle and then stores that value in the variable $a:  $a = [math]::tan(45)  When you run this command and then echo back the value of $a you should get the following:  1.61977519054386 |
| Time | **Definition:** Returns a **Variant** of subtype **Date** indicating the current system time.  OK, give us a hard one. What’s that? You’d like to store just the current time – and only the current time – in a variable? Fortunately, that’s not very hard; for example, one way we can do that is to call the **Get-Date** Cmdlet, then use the **–displayhint** parameter to assign just the current time to a variable. Sort of like this command, in which we store the current time to the variable $a:  $a = get-date -displayhint time  When you run this command and then echo back the value of $a you should get the following (assuming you run it at 11:22:37 AM):  11:22:37 AM |
| Timer | **Definition:** Returns the number of seconds that have elapsed since 12:00 AM (midnight).  In VBScript the Timer function is typically used to determine how long it takes a script, function, or subroutine to execute. If that's what you want to do then you can use the Windows PowerShell **Measure-Command** Cmdlet to calculate elapsed time. The following command creates a For Next loop that runs from 1 to 100,000, writing each value of $a to the screen as it goes. Along the way, the Measure-Command Cmdlet is used to keep track of how long it takes for the command to finish:  measure-command {  for ($a = 1; $a -le 100000; $a++)  {write-host $a}  }  When you run this command you should get back information similar to the following:  Days : 0  Hours : 0  Minutes : 1  Seconds : 9  Milliseconds : 365  Ticks : 693655925  TotalDays : 0.000802842505787037  TotalHours : 0.0192682201388889  TotalMinutes : 1.15609320833333  TotalSeconds : 69.3655925  TotalMilliseconds : 69365.5925 |
| TimeSerial | **Definition:** Returns a **Variant** of subtype **Date** containing the time for a specific hour, minute, and second.  Like DateSerial, this is a function that is in no danger of being overused. However, if you want to pass in values for the hours, minutes, and seconds and get back a *time* value (as opposed to a date-time value) the following command will do the trick:  $a = get-date -h 17 -mi 10 -s 45 -displayhint time  **Note**. As you probably figured out for yourself, the **–h** parameter represents the hours, the **–mi** parameter represents the minutes, and the **–s** parameter represents the seconds. Values for the hours are based on a 24-hour clock.  When you run this command and then echo back the value of $a you should get the following:  5:10:45 PM |
| TimeValue | **Definition:** Returns a **Variant** of subtype **Date** containing the time.  OK, so you have a string value (say, 1:45 AM) and you’d like to convert that to a date-time value, is that right? No problem; Windows PowerShell can do that for you. In this command we not only assign the string *1:45 AM* to the variable $a, but we also explicitly give $a a datetime data type:  $a = [datetime] "1:45 AM"  When you run this command (assuming it was run on 9/21/2006) and then echo back the value of $a you should get the following:  Thursday, September 21, 2006 1:45:00 AM |
| TypeName | **Definition:** Returns a string that provides **Variant** subtype information about a variable.    In system administration scripting it's often useful to know what type of a variable you're working with; for example, databases don't like it if you try to save string data into a field reserved for numeric data. In Windows PowerShell you can use the **GetType()** method to determine the variable type of any variable. For example, the following set of commands assigns the value 55.86768 to the variable $a, then uses GetType() to retrieve the **Name** of $a's variable type. The resulting type name is assigned to the variable $b:  $a = 55.86768  $b = $a.gettype().name  When you run this command and then echo back the value of $b you should get the following:  Double |
| UBound | **Definition:** Returns the largest available subscript for the indicated dimension of an array.  There are at least two ways to determine the index number of the last item in a Windows PowerShell array. For example, suppose we have the following array:  $a = "a","b","c","d","e"  With a 5-item array the last item has an index number of 4 (because the first item in the array is actually item 0). But how do we *know* that? One way is to use the **GetUpperBound()** method. This command returns the upper bound for the first dimension (i.e., dimension 0) in the array $a:  $a.getupperbound(0)  You can also get the same results by subtracting one from the array’s **Length** property:  $a.length-1  When you run either command you should get the following:  4 |
| UCase | **Definition:** Returns a string that has been converted to uppercase.  Sometimes bigger truly *is* better, which is one reason you might want to convert all the characters in a string to their uppercase equivalent. How do you pull off such a feat in Windows PowerShell? That’s easy: you just call the **ToUpper()** method. For example, these two commands assign the letters of the alphabet to the variable $a, then use the ToUpper() method to convert each of those letters to uppercase:  $a = "abcdefghijklmnopqrstuvwxyz"  $a = $a.ToUpper()  When you run this command and then echo back the value of $a you should get the following:  ABCDEFGHIJKLMNOPQRSTUVWXYZ |
| Unescape | **Definition:** Decodes a string encoded with the **Escape** function.  Suppose someone hands you a string value like this one, a value that has been encoded using the VBScript Escape function (or its equivalent):  http%3a%2f%2fwww.microsoft.com%2ftechnet%2fscriptcenter%2fdefault.mspx  Can you use Windows PowerShell to decode this value (that is, turn it back into a regular string value)? You can if you use the .Net Framework’s **System.Web.HTTPUtility** class and the **URLDecode()** method. Assuming that the string value is stored in the variable $a these two commands should do the trick:  [Reflection.Assembly]::LoadWithPartialName("System.Web")  $a = [web.httputility]::urldecode($a)  When you run the commands and then echo back the value of $a you should get the following:  http://www.microsoft.com/technet/scriptcenter/default.mspx |
| VarType | **Definition:** Returns a value indicating the subtype of a variable.  In VBScript you can use the VarType function to return an integer value that represents a particular data type (for example, a 9 represents an object). To be honest, we don’t know how to do this in Windows PowerShell (that is, return an integer value as opposed to the actual data type name). And, to be equally honest, we don’t think it matters much.  **Technical Note**. VBScript uses a data type known as a Variant. That means that its types are comprised of 2 things: a chunk of memory where the type value is stored, and a description of the type. If you have a set of bits stored in memory somewhere, that’s not enough to tell where it’s a number 32 or the space character or a certain object with certain values. Because of that VBScript’s VarType function returns an integer that tells you the type bit for the Variant.  PowerShell variables are strongly typed. That means there is no exact equivalent for the VarType function. To determine the data type in a PowerShell variable you pipe the contents of the variable to Get-Member and rely on the Cmdlet to tell you the type. There’s no way to have an enumeration that maps to each possible data type; because you can construct new data types on the fly, that enumeration would have to account for an infinite number of data types. Needless to say, that would be a pretty tough job. |
| Weekday | **Definition:** Returns a whole number representing the day of the week.  You know, you’re right: if you want to return an integer value representing a day of the week (say, getting back a *6* as opposed to getting back *Friday*), well, that’s really none of our business, is it? To tell you the truth, we don’t know of any way built into Windows PowerShell to do this; however, you *can* do this by using the .NET Framework’s **Microsoft.VisualBasic.DateAndTime** class and the **DatePart** function. The following three commands assign the current date and time to the variable $a, load the Microsoft.VisualBasic assembly, then use the DatePart function to return an integer representing the day of the week. Note that DatePart requires two parameters:   * **w**, which tells the function we want the day of the week. * **$a**, the date we want to check.   Here are the commands:  $a = get-date  [reflection.assembly]::LoadWithPartialName("'Microsoft.VisualBasic")  [Microsoft.VisualBasic.DateAndTime]::DatePart("w",$a)  Or, if you’re really gung-ho on this, here’s another approach:  The Get-Date Cmdlet returns a System.DateTime object that has a DayOfWeek property. This property is an enumeration (a sequence of numbers that each represent something) with 0 representing Sunday, 1 representing Monday, and so on. If you type **(Get-Date).DayOfWeek**, you will see the text representation of the day. To get at the underlying numeric value you need to type this:  (Get-Date).DayOfWeek.value\_\_  **Note**: The value is followed by 2 underscores.  Of Course, because the DayOfWeek enumeration starts by numbering Sunday at 0 rather than 1, you need to add 1 at the end of this last command to get the same result as the VBScript function.  (Get-Date).DayOfWeek.value\_\_ + 1    Assuming you run these commands on a Friday you should get back the following:  6 |
| WeekdayName | **Definition:** Returns a string indicating the specified day of the week.  Determining the day of the week (Monday, Tuesday, Wednesday, etc.) is a tad bit convoluted in VBScript; you have to first call the Weekday function (which returns an integer value representing the day of the week) and then call the WeekdayName function to translate that integer into a string value. In Windows PowerShell this is actually much easier; you simply get the desired date and then get the value of the **DayOfWeek** property. For example, this command assigns the DayOfWeek for the current date to the variable $a:  $a = (get-date).dayofweek  When you run this command and then echo back the value of $a you should get something similar the following:  Friday  What if you wanted to determine the day of the week for a date other than the current one? No problem; this command returns the day of the week for 12/25/2007:  $a = (get-date "12/25/2007").dayofweek  And here’s what you get back after running that command:  Tuesday |
| Year | **Definition:** Returns a whole number representing the year.  Don’t feel bad: sometimes the Scripting Guys forget what year it is, too. Or at least we used to, until we realized we could use Windows PowerShell to determine the year for us. All we have to is call the **Get-Date** Cmdlet and then grab the value of the **Year** property. For example, this command retrieves the current year and stores that value in the variable $a:  $a = (get-date).year  When you run this command and then echo back the value of $a you should get the following (assuming the command was run during the year 2006):  2006  Of course, you can determine the year for *any* date; all you have to do is ask Get-Date to work with that date instead. For example, this command returns the year for the date 9/15/2005:  $a = (get-date "9/15/2005").year  When you run this command and then echo back the value of $a you should get the following:  2005 |

|  |  |
| --- | --- |
| **VBScript Statement** | **Windows PowerShell Equivalent** |
| Call Statement | **Definition:** Transfers control to a **Sub** or **Function** procedure.  The Call statement is optional in VBScript: you can call a subroutine or function either by using the Call statement followed by the procedure name or by simply specifying the procedure name. The Call statement is not used at all in Windows PowerShell; instead, you call a function simply by specifying the function name followed by any function parameters. For example, this command calls the function MultiplyNumbers, supplying a pair of parameters (25 and 67):  multiplynumbers 25 67 |
| Class Statement | **Definition:** Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class.  Although it *is* possible to create a class in Windows PowerShell, it’s not a very straightforward process and definitely goes beyond the scope of this introductory manual. So, for the time being, forget we even mentioned it. |
| Const Statement | **Definition:** Declares constants for use in place of literal values.  A constant is nothing more than a variable whose value cannot be changed while a script is running. In Windows PowerShell you can define a constant by using the **Set-Variable** Cmdlet and specifying three parameters:   * **-name**, the name of the constant (don’t include the $). * **-value**, the value of the constant. * **-option** **constant**, to create a constant as opposed to a standard variable.   For example, this command creates a constant named $ForReading, with a value of 1:  set-variable -name ForReading -value 1 -option constant  If you try to assign a new value to this constant you’ll get an error message similar to this:  Cannot overwrite variable ForReading because it is read-only or constant. |
| Dim Statement | **Definition:** Declares variables and allocates storage space.  In VBScript the Dim statement is used to declare a variable, typically without assigning that variable a value of any kind. To declare a variable in Windows PowerShell, but without assigning a value to the variable, use a command similar to this:  $a = [string]  This command creates a new, empty string variable named $a. In order to declare a variable without assigning it a value you must specify a data type (although this data type can later be changed). |
| Do...Loop Statement | **Definition:** Repeats a block of statements while a condition is **True** or until a condition becomes **True**.  The preceding definition pretty much says it all: a Do While loop enables you to repeat a block of code over and over again, at least as long as a specified condition is true. By comparison, a Do Until loop enables you to repeat a block of code over and over again, at least *until* a specified condition becomes true. As you might have guessed, you can create either type of loop using Windows PowerShell.  To begin with, let’s take a look at the Do While loop. To create a Do While loop you need the following items:   * The **Do** keyword. * The action (or actions) you want to perform. These actions must be contained within curly braces. * The **While** keyword. * The condition that will spell the end of the loop. (In other words, when this condition is met then the loop should end.)   The following example assigns the value 1 to the variable $a, then uses that variable as part of a Do While loop. The action to be performed is fairly simple: we just echo back the value of $a, then use the **++** operator to increment $a by 1. That’s what this portion of the code does:  {$a; $a++}  Meanwhile, we’ll keep the loop running as long as $a is less than (**-lt**) 10:  ($a -lt 10)  Here’s the complete example:  $a = 1  do {$a; $a++} while ($a -lt 10)  And here’s what you’ll get back if you execute those two lines of code:  1  2  3  4  5  6  7  8  9  The Do Until loop is constructed the same way, except that we use the **Until** keyword instead of the While keyword. In this example, the loop is designed to run until $a is equal to (**-eq**) 10:  $a = 1  do {$a; $a++} until ($a -eq 10)  And here’s what you’ll get back if you execute these two lines of code:  1  2  3  4  5  6  7  8  9 |
| Erase Statement | **Definition:** Reinitializes the elements of fixed-size arrays and deallocates dynamic-array storage space.  The Erase statement is similar to a check for a million dollars: although we have no doubt that these items exist, the Scripting Guys have little experience with either one. Suppose you have an array with 9 items. When you run the Erase statement against that array you’ll still have an array with 9 items; however, none of those items will have a value (string values will now be empty, while numeric values will be set to 0).  Got that? Like we said, we run across the Erase statement about as often as we run across a million dollars. However, if you need to set all the values in an array to 0, well, here’s a command that can do that for you:  for ($i = 0; $i -lt $a.length; $i++) {$a[$i] = 0}  Let’s assume that $a started off life as an array with the following values:  1  2  3  4  5  6  Here’s what $a looks like after running the preceding command:  0  0  0  0  0  0 |
| Execute Statement | **Definition:** Executes one or more specified statements.  The VBScript Execute statement enables you to assign an expression (typically lines of VBScript code) to a variable; you can then “execute” that variable as though it was code hard-coded into the script itself (as opposed to code existing only in memory).  Don’t worry about it; we’ll just show you what we mean. In the following example, the string value “get-date” is assigned to the variable $a. The **Invoke-Expression** Cmdlet is then used to execute $a as though the value had been typed from the command prompt:  $a = "get-date"  invoke-expression $a  When you run these two commands you’ll get back the current date and time (because the string value "get-date" will be executed as a command). |
| ExecuteGlobal Statement | **Definition:** Executes one or more specified statements in the global namespace of a script.  Is there a difference between Execute and ExecuteGlobal? Yes, but we decided it wasn’t all that important, at least not for now. In turn, that meant we could simply use the same example for both statements. In the following example, the string value “get-date” is assigned to the variable $a. The **Invoke-Expression** Cmdlet is then used to execute $a as though the value had been typed from the command prompt:  $a = "get-date"  invoke-expression $a  When you run these two commands you’ll get back the current date and time (because the string value "get-date" will be executed as a command). |
| Exit Statement | **Definition:** Exits a block of **Do...Loop**, **For...Next**, **Function**, or **Sub** code.  In Windows PowerShell you can “prematurely” break out of a loop by using the **break** command; this is equivalent to using the Exit Do or Exit For statement in VBScript. For example, the following Windows PowerShell script creates an array ($a) containing 9 values. The script then uses a For Each loop to enumerate all the items in that array; however, if any of those items is equal to 3 – **if ($i –eq 3)** – then the break command is used to exit the loop right then and there. The script itself looks like this:  $a = 1,2,3,4,5,6,7,8,9  foreach ($i in $a)  {  if ($i -eq 3)  {  break  }  else  {  $i  }  }  And here’s what you get back when you run the script:  1  2 |
| For Each...Next Statement | **Definition:** Repeats a group of statements for each element in an array or collection.  A For Each loop is designed to automatically run through all the items in a collection or array. In Windows PowerShell you can iterate all the items in a collection by using the **ForEach-Object** collection (or its more-commonly-used alias, **foreach**). To set up a For Each loop in Windows PowerShell, call foreach followed by two parameters:   * The collection you want to iterate. This value must be enclosed in parentheses. * The action you want to perform on each item in that collection. This value must be enclosed in curly braces: {}.   For example, the following command sets up a For Each loop that uses the **Get-ChildItem** Cmdlet to return a collection of all the files found in the folder C:\Scripts. Note the syntax that must be used: **($i in get-childitem c:\scripts)**. Obviously this is different than what you are used to. After all, in VBScript you “kick off” a For Each loop using code similar to this:  For Each i in colItems  However, in Windows PowerShell you call foreach and then kick things off using code like this:  ($i in $colItems)  After specifying the loop condition the command then indicates the behavior to be performed:  {$i.extension}  This simply says, “For each item ($i) in the collection, echo back the value of the **Extension** property.”  Here’s the actual command:  foreach ($i in get-childitem c:\scripts) {$i.extension}  You should get back something similar to this, depending on the files found in C:\Scripts:  .gif  .xls  .xls  .txt  .txt  .xls  .ps1  .ps1  .zip  .vbs  .txt  .ppt |
| For...Next Statement | **Definition:** Repeats a group of statements a specified number of times.  A For Each loop is designed to run a block of code against each and every item in a collection; by contrast, a For Next loop is typically designed to run a specific number of times. For example, to run a code block 10 times you might set up a loop that begins with a counter variable set to 1. Each time through the loop the counter variable is incremented by1, and the loop ends only after that counter variable is equal to 10.  In other words, something similar to this:  for ($a = 1; $a -le 10; $a++) {$a}  In this command we use the **For** keyword followed by two items: the loop condition (enclosed in parentheses) and the code we want to execute each time we run through the loop (enclosed in curly braces). The loop condition requires three values:   * The starting value. We want our loop to run from 1 to 10, so we set a counter variable named $a equal to 1. * The ending value. We want the loop to run 10 times, so we specify that the loop should continue as long as $a is less than or equal to 10. If and when the value of $a becomes greater than 10 the loop automatically ends. * The “incrementer.” Each time through the loop we want the value of $a to increase by 1; therefore we use the **++** operator. Suppose we wanted to increment $a by 2 each time through the loop (equivalent to **Step 2** in a VBScript loop). In that case we would use this syntax: **$a+=2**, which adds 2 to the current value of $a.   As to the code we want to run each time through the loop, well, here we’re keeping it pretty simple; we’re just echoing back the current value of $a:  {$a}  Here’s what we’ll get back when we execute this command:  1  2  3  4  5  6  7  8  9  10 |
| Function Statement | **Definition:** Declares the name, arguments, and code that form the body of a **Function** procedure.  You can define a function in Windows PowerShell by using the **function** statement followed by:   * The name of the function. * The task to be performed by the function (enclosed in curly braces).   For example, the following command defines a function named MultiplyNumbers. This function will take two arguments ($args[0] and $args[1]) and multiply those two values. Here’s the command itself:  function multiplynumbers { $args[0] \* $args[1] }  After a function has been defined you can call that function simply by specifying its name and (in this case) the two values you want multiplied:  multiplynumbers 38 99  When you run the preceding command you should get back the following:  3762 |
| If...Then...Else Statement | **Definition:** Conditionally executes a group of statements, depending on the value of an expression.  If you know how to use If Then statements in VBScript then you have a nice head start on using If Then statements in Windows PowerShell. After all, the underlying philosophy is exactly the same; all you have to do is learn the Windows PowerShell syntax.  With that in mind, let’s take a look at a simple If Then statement in Windows PowerShell. Before we go much further, we should note that we could have typed the entire statement on a single line. We *could* have, but we didn’t, mainly because we wanted to keep things as simple and easy-to-follow as possible.  But we could have.  Here’s our example which, after assigning the value 5 to the variable $a, uses an If Then statement to determine whether or not $a is less then 10, and then echoes back the appropriate message:  $a = 5  if ($a -gt 10)  {"The value is greater than 10."}  else  {"The value is less than 10."}  Right off the bat you should notice two things. First, although we refer to this as an If Then statement, we never actually use the word *Then*. Instead, all we need is the keyword **If** followed by the condition we are checking for, enclosed in parentheses:  if ($a -gt 10)  The second thing to notice? There’s no EndIf keyword (or its equivalent). In Windows PowerShell you typically don’t need to using “ending” statement like EndIf, Loop, Next, etc.  Immediately following the condition we have the action we want to take if the condition is true, that is, if $a really *is* greater than 10. These actions must be enclosed in curly braces, like so:  {"The value is greater than 10."}  So far what we’ve looked at is equivalent to the following VBScript code:  If a > 10 Then  Wscript.Echo "The value is greater than 10."  All we have to do now is add the Else statement. We do that simply by typing in the keyword **Else** followed by the action to be taken should the condition *not* be true. In other words:  else  {"The value is less than 10."}  And, yes, we could include an ElseIf (or two or three or …) in here as well. In fact, here’s a revised If Then statement that includes an ElseIf. If the value of $a is *red* a message to that effect will be echoed to the screen; if the value of $a is *white* – **elseif ($a –eq "white")** – a message to *that* effect is echoed to the screen. (Note that the ElseIf condition must be enclosed in parentheses, and the ElseIf action must be enclosed in curly braces.) Otherwise (**else**) a message saying that the color is blue is echoed back to the screen.  Here’s what our new example looks like:  $a = "white"  if ($a -eq "red")  {"The color is red."}  elseif ($a -eq "white")  {"The color is white."}  else  {"The color is blue."} |
| On Error Statement | **Definition:** Enables or disables error-handling.  By default Windows PowerShell issues an error message the moment an error occurs. If you prefer that processing continue without displaying an error message then set the value of the Windows PowerShell automatic variable $ErrorActionPreference to **SilentlyContinue**. You know, like this:  $erroractionpreference = "SilentlyContinue"  Incidentally, your choices for this variable include:   * SilentlyContinue * Continue (the default value) * Inquire * Stop   To illustrate the differences between these error-handling states, suppose we have the following script, a script that features a syntax error on the very first line:  bob  $a = 2  $b = 2  $c = 2 + 2  $c  If you run this script and error-handling is set to **Continue**, you’ll get back the following:  The term 'bob' is not recognized as a Cmdlet, function, operable program, or script file. Verify the term and try again  .  At C:\scripts\test.ps1:1 char:4  + bob <<<<  4  As you can see, the script ended up running and the value 4 is echoed back to the screen, but only after a detailed error message has been displayed.  By contrast, here’s what you get when error-handling has been set to SilentlyContinue:  4  The script continued to process, and no error was reported.  If error-handling is set to Inquire Windows PowerShell will suspend operations and ask you how you wish to proceed:  Action to take for this exception:  The term 'bob' is not recognized as a Cmdlet, function, operable program, or script file. Verify the term and try again.  [C] Continue [I] Silent Continue [B] Break [S] Suspend [?] Help (default is "C"):  Finally, here’s what happens if error-handling is set to Stop:  The term 'bob' is not recognized as a Cmdlet, function, operable program, or script file. Verify the term and try again  .  At C:\scripts\test.ps1:1 char:4  + bob <<<<  In this case the error message is displayed and the script terminates; the value 4 is not displayed because once the error occurred no additional lines of code were executed. |
| Option Explicit Statement | **Definition:** Forces explicit declaration of all variables in a script.  In VBScript, the Option Explicit statement forces you to declare all variables before using them in a script. To do the same thing in Windows PowerShell use the **Set-PSDebug** Cmdlet and the **–strict** parameter, like so:  set-psdebug –strict  What will that do for you? Well, suppose you now try to execute a command similar to this, using the uninitialized variable $z:  $a = 2 + $z  You won’t get an answer; instead you’ll get the following error message:  The variable $z cannot be retrieved because it has not been set yet.  To turn off this behavior, use this command:  set-psdebug -off |
| Private Statement | **Definition:** Declares private variables and allocates storage space. Declares, in a **Class** block, a private variable.  The Private statement in VBScript is concerned with classes; we’re going to take the liberty of extending the definition of Private here, primarily because we aren’t going to talk about classes. Instead of talking about private variables in terms of classes, we’re going to talk about private variables in terms of scope.  In Windows PowerShell all you have to do is set the scope to Private. For example, this command creates a variable named $a with the scope set to private. Note that, when specifying a scope, you do not preface the variable name with a $; however, in your commands you will still refer to this variable as $a.  Here’s the command:  $Private:a = 5  Because this is a private variable it will be available only in the scope in which it was created. If you type $a from the command prompt you’ll get back the value of $a. However, if you run a script that simply echoes back the value of $a you won’t get back anything; that’s because, as a private variable, $a is not available outside the command shell scope. |
| Property Get Statement | **Definition:** Declares, in a **Class** block, the name, arguments, and code that form the body of a **Property** procedure that gets (returns) the value of a property.  The VBScript Get statement is used when working with VBScript classes. Custom classes are theoretically possible in Windows PowerShell, but lie far beyond the scope of this introductory document. Therefore, we’ll take the easy way out and simply say that there is no Windows PowerShell equivalent to the Get statement. |
| Property Let Statement | **Definition:** Declares, in a **Class** block, the name, arguments, and code that form the body of a **Property** procedure that assigns (sets) the value of a property.  The VBScript Let statement is used when working with VBScript classes. Custom classes are theoretically possible in Windows PowerShell, but lie far beyond the scope of this introductory document. Therefore, we’ll take the easy way out and simply say that there is no Windows PowerShell equivalent to the Let statement. |
| Property Set Statement | **Definition:** Declares, in a **Class** block, the name, arguments, and code that form the body of a **Property** procedure that sets a reference to an object.  The VBScript Set statement is used when working with VBScript classes. Custom classes are theoretically possible in Windows PowerShell, but lie far beyond the scope of this introductory document. Therefore, we’ll take the easy way out and simply say that there is no Windows PowerShell equivalent to the Set statement. |
| Public Statement | **Definition:** Declares public variables and allocates storage space. Declares, in a **Class** block, a private variable.  Windows PowerShell enables you to create a global (public) variable simply by specifying the scope when you create the variable. For example, this command creates a variable named $a with the scope set to Global. Note that, when specifying a scope, you do not preface the variable name with a $; however, the variable will, in fact, be named $a.  $Global:a = 199  Here’s an interesting use for global variables. Start up Windows PowerShell, but do *not* declare the variable $a. Instead, put the preceding line of code in a script (e.g., test.ps1), run the script, and *then* type $a from the command prompt. You should get back the following:  199  Because this is a global variable, it’s available in scripts, in the command shell, and in any other scope. Cool, huh?  **Note**. Admittedly, this isn’t quite the same as what the Public statement does in VBScript; that’s because the VBScript statement is concerned with classes, and we aren’t discussing Windows PowerShell classes in this introductory series. Therefore we’ve extended the definition of “public” to include Windows PowerShell scopes. That gives us a chance to talk about the Windows PowerShell global variables, which we find very cool. |
| Randomize Statement | **Definition:** Initializes the random-number generator.  The Randomize statement (or its equivalent) is not required in Windows PowerShell. Instead, you can generate a random number using code similar to this:  $a = new-object random  $b = $a.next()  When you run this command and then echo back the value of $b you should back something similar to the following:  1558696419 |
| ReDim Statement | **Definition:** Declares dynamic-array variables, and allocates or reallocates storage space at procedure level.  In general there’s no need for a ReDim statement in Windows PowerShell; arrays can be resized without having to explicitly call ReDim (and without having to explicitly preserve the existing contents of the array). For example, suppose the following 5 items are assigned to the array $a:  $a = 1,2,3,4,5  Want to add a new value – 100 – to the array? This is all you have to do:  $a = $a + 100  If you echo back the value of $a you’ll get the following:  1  2  3  4  5  100  Likewise you can truncate an array without explicitly redimensioning anything. Suppose you want only the first three values in $a. This code assigns just those first three values (using the syntax **0..2**, which means start with item 0 in the array, end with item 2, and take all the values in between) and assigns them to $a:  $a = $a[0..2]  What do you get back when you execute this command? This is what you get back:  1  2  3 |
| Rem Statement | **Definition:** Includes explanatory remarks in a program.  To add a comment to a Windows PowerShell script (or a Windows PowerShell command, for that matter) simply preface that comment with the pound sign (#); anything following the pound sign will be treated as a comment and ignored by the script processor. For example, this command calls the **Get-Date** Cmdlet and then tacks on a comment explaining what the Cmdlet does:  get-date # This Cmdlet retrieves the current date and time  When you execute this command you’ll simply get back the current date and time; the comment will be ignored. That’s also true with this command:  get-date # get-help  The Get-Date Cmdlet will be executed; the **Get-Help** Cmdlet will not. That’s because, in this example, Get-Help is a comment, not a command. |
| Select Case Statement | **Definition:** Executes one of several groups of statements, depending on the value of an expression.  In VBScript you can test for multiple possibilities by using a long and sometimes-complicated series of If Then ElseIf statements. Alternatively, you can achieve the same effect – with far less typing – by using a Select Case statement.  In Windows PowerShell you can emulate (and, in some ways, improve upon) the Select Case statement by using the **Switch** statement. To explain how the Switch statement works, let’s take a look at some sample code:  $a = 5  switch ($a)  {  1 {"The color is red."}  2 {"The color is blue."}  3 {"The color is green."}  4 {"The color is yellow."}  5 {"The color is orange."}  6 {"The color is purple."}  7 {"The color is pink."}  8 {"The color is brown."}  default {"The color could not be determined."}  }  In the preceding code we assign the value 5 to the variable $a. We then create a Switch block that assesses the value of $a and takes the appropriate action based upon that value:  switch ($a)  As you can see, all we have to do is insert the Switch keyword followed by the value to be tested (which must be enclosed in parentheses).  Next we list the possible values for $a along with the corresponding action (this entire block, incidentally, must be enclosed in curly braces). For example, if $a is equal to 1 we want to echo back a message saying that the color is red. Therefore, we use this line of code, with the action to be taken enclosed in curly braces:  1 {"The color is red."}  See how that works? (It’s actually pretty easy.) We can also specify an action to be taken if none of the preceding Case statements are true (the same thing you do with Case Else in VBScript). To do that we simply use the **default** condition followed by the desired action:  default {"The color could not be determined."} |
| Set Statement | **Definition:** Assigns an object reference to a variable or property, or associates a procedure reference with an event.  You do not need to use the Set statement (or any sort of equivalent keyword/command) in order to create an object reference in Windows PowerShell. Instead, you simply use the **New-Object** Cmdlet, followed by (in the case of COM objects) the **–comobject** parameter and the ProgID of the object to be created.  For example, the following two commands create an instance of Microsoft Excel and then, just to prove that the instance of Excel *was* created, make the application visible onscreen:  $a = new-object -comobject Excel.Application  $a.visible = $True |
| Stop Statement | **Definition:** Suspends execution.  If you have the Microsoft Scripting Debugger installed you can insert Stop statements that temporarily suspend the script each and every time they are encountered. Windows PowerShell does not have a similar feature (in part because it does not have a script debugger). However, you *can* run Windows PowerShell in step-by-step mode, which is roughly akin to putting a Stop statement after each and every line of code. That can be done by using the **Set-PSDebug** Cmdlet and using the **–step** parameter:  set-psdebug –step  To exit the step-by-step mode simply use this command:  set-psdebug -off |
| Sub Statement | **Definition:** Declares the name, arguments, and code that form the body of a **Sub** procedure.  Windows PowerShell supports only functions, as opposed to functions and subroutines. You can define a function in Windows PowerShell by using the **function** statement followed by:   * The name of the function. * The task to be performed by the function (enclosed in curly braces).   For example, the following command defines a function named MultiplyNumbers. This function will take two arguments ($args[0] and $args[1]) and then multiply those two values. Here’s the command:  function multiplynumbers { $args[0] \* $args[1] }  From then on you can call the function simply by specifying the function name and the two values you want multiplied:  multiplynumbers 38 99  When you run the preceding command you should get back the following:  3762 |
| While...Wend Statement | **Definition:** Executes a series of statements as long as a given condition is **True**.  The VBScript While Wend loop is just another way of creating a Do While loop. (You can never have too many ways to create loops, at least not in VBScript.) You can create a While loop in Windows PowerShell by using the – surprise – **while** statement. While takes two parameters:   * The loop conditions (in other words, how long do you intend to keep looping). * The action to be performed in each iteration of the loop.   In the following example, the value 1 is assigned to the variable $a. A While loop is then established that does two things:   * Continues to loop as long as the value of $a is less than (**-lt**) 10. * On each iteration of the loop displays the current value of $a, and then increments that value by 1. In Windows PowerShell, the **++** operator increments a value by 1; the syntax **$a++** is equivalent to VBScript’s a = a + 1.   Here’s the sample code:  $a = 1  while ($a -lt 10) {$a; $a++}  And here’s what you should get back after running the preceding commands:  1  2  3  4  5  6  7  8  9 |
| With Statement | **Definition:** Executes a series of statements on a single object.  As far as we know there is no Windows PowerShell equivalent to the With statement. |

|  |  |
| --- | --- |
| **VBScript Err Object** | **Windows PowerShell Equivalent** |
| Description Property | **Definition:** Returns or sets a descriptive string associated with an error.  So what *was* the last error that occurred? If you need to know that, just get the value of error number 0 and then call the **ToString()** method. The following command retrieves information from the most recent error and uses the ToString() method to assign the value to the variable $a:  $a = $error[0].ToString()  When you run this command and then echo back the value of $a you should back something similar to the following:  The term 'error[0]' is not recognized as a Cmdlet, function, operable program, or script file. Verify the term and try  again. |
| HelpContext Property | **Definition:** Sets or returns a context ID for a topic in a Help File.  If an error is associated with a topic in a help file you can retrieve that information using the **HelpLink** property. This command assigns the HelpLink property for the most recent error in the error collection to the variable $a:  $a = $error[0].helplink |
| HelpFile Property | **Definition:** Sets or returns a fully qualified path to a Help File.  If an error is associated with a topic in a help file you can retrieve that information using the **HelpLink** property; the first part of the link will be the fully-qualified path to the help file itself. This command assigns the HelpLink property for the most recent error in the error collection to the variable $a:  $a = $error[0].helplink |
| Number Property | **Definition:** Returns or sets a numeric value specifying an error. **Number** is the **Err** object's default property.  We’re just guessing here, but we assume that the **ErrorRecord** property is designed to return the error number. However, we’ve never actually encountered a numbered error in Windows PowerShell; in our experience, ErrorRecord usually returns a string value such as this:  ScriptHalted  Still, you can always give this command a try and see what happens:  $error[0].errorrecord |
| Source Property | **Definition:** Returns or sets the name of the object or application that originally generated the error.  To determine which object or component was responsible for an error you can check the value of the **Source** property. This command assigns the Source property for the most recent error in the error collection to the variable $a:  $a = $error[0].source  When you run this command and then echo back the value of $a you should back something similar to the following:  System.Management.Automation |
| Clear Method | **Definition:** Clears all property settings of the **Err** object.  The VBScript Err object keeps track of only the last error that occurred; in contrast, and by default, Windows PowerShell keeps track of the last 256 errors that occurred. Among other things, that changes the definition of “clearing” an error in Windows PowerShell. If you want to mimic VBScript, you can clear the last error – and only the last error – by setting the last error to an empty string (the zero in square brackets represents the last error in the collection):  $error[0] = ""  Alternatively, you can clear out the entire error collection by using the **Clear()** method:  $error.clear() |
| Raise Method | **Definition:** Generates a run-time error.  When testing scripts script writers need to ensure that their scripts can respond appropriately to specific errors. One way to do that is to simulate those errors programmatically. In VBScript this can be done using the Raise function; in Windows PowerShell this same thing can be done using the **Throw** method. In the following command an error message is assigned to the variable $b; the Throw method is then used to “throw” the error associated with $b:  $b = "The file could not be found."; throw $b  If you now check the value of the most recent error (that is, $error[0]) you’ll get back the following:  The file could not be found. |

|  |  |
| --- | --- |
| **VBScript Operator** | **Windows PowerShell Equivalent** |
| Addition operator (+) | **Definition:** Sums two numbers.  Here’s a shocker: in Windows PowerShell you add two numbers by using the plus sign. In other words:  $a = 2 + 2  We’ll let you figure out for yourself what the value of $a will be equal to. |
| And operator (AND) | **Definition:** Performs a logical conjunction on two expressions.  The AND operator evaluates two arguments and returns True only if both arguments are actually true. For example, suppose we want to evaluate the following two statements, then store the verdict in the variable $a:   * 23 is greater than 12 * 12 is greater than 40   Here’s the command we’d use:  $a = 23 -gt 12 -and 12 -gt 40  As you might expect, if you run the command and then echo back the value of $a you’ll get the following:  False |
| Assignment operator (=) | **Definition:** Assigns a value to a variable or property.  Assigning a value to a variable is no big deal in Windows PowerShell: you put the variable on the left side of the equation, the value to be assigned on the right side, and an equals sign (=) smack dab in the middle. For example, this command assigns the string value *test* to the variable $a:  $a = "test"  And this command assigns the sum of the numbers 1, 2, and 3 to $a:  $a = 1 + 2 + 3 |
| Concatenation operator (&) | **Definition:** Forces string concatenation of two expressions.  In VBScript you can join two or more strings by using either the plus sign (+) or the ampersand (&).  **Note**. Or, if you want to sound fancy: In VBScript you can *concatenate* two or more strings by using either the plus sign (+) or the ampersand (&).  In Windows PowerShell you can concatenate two or more strings by using the plus sign (don’t even bother trying to use the ampersand). For example, this command combines the word *test*, a blank space, and the word *value* and stores the resulting string in a variable named $a:  $a = "test" + " " + "value"  When you run this command and then echo back the value of $a you should get the following:  test value |
| Division operator (/) | **Definition:** Divides two numbers and returns a floating-point result.  Divide and conquer. We can’t make any promises regarding the conquer part of that statement, but in Windows PowerShell you can divide a pair of numbers by using / as the division operator. For example, this commands divides 761 by 11 and stores the value in the variable $a:  $a = 761 / 11  When you run this command and then echo back the value of $a you should get the following:  69.1818181818182 |
| Eqv operator (Eqv) | **Definition:** Performs a logical equivalence on two expressions.  No doubt Eqv has its uses; we’re just not sure if it has any *practical* uses. Although there might be a Windows PowerShell equivalent we have to be honest: we didn’t look real hard for one. |
| Exponentiation operator (^) | **Definition:** Raises a number to the power of an exponent.  Interestingly enough, Windows PowerShell doesn’t have a built-in exponentiation operator. So does that mean you’re out of luck if you need to raise 8 to the third power? Of course not; all you need to do is use the **System.Math** class and the **Pow** method:  $a = [math]::pow(8,3)  When you run this command and then echo back the value of $a you should get the following:  512 |
| Imp operator (Imp) | **Definition:** Performs a logical implication on two expressions.  What do we know about the Imp operator? Two things: 1) it’s not supported in Visual Basic .NET, and 2) we have never heard of anyone actually using it. Therefore, we’re going to say that it doesn’t really matter whether or not there’s a Windows PowerShell equivalent. |
| Integer division operator (\) | **Definition:** Divides two numbers and returns an integer result.  In VBScript the integer division operator returns only the integer portion of an answer, stripping off decimal points without rounding the number either up or down. For example, 100 / 26 is equal to 3.84615384615385; however 100 **\** 26 is equal to 3 (just the integer portion of the answer).  **Note**. OK, so that’s not 100% true; that’s only true for whole numbers. Given a fractional value, that value will be rounded up or down before division takes place. But that’s all right; we’ve accounted for that by converting the values in the equation to integers before we divide them.  You can perform integer division in Windows PowerShell by using the .NET Framework class **System.Math** and the **Floor** method. The following command divides 100 by 26 and then stores the integer portion of the answer in the variable $a:  $a = [math]::floor([int] 100 / [int] 26)  When you run this command and then echo back the value of $a you should get the following:  3 |
| Is operator (Is) | **Definition:** Compares two object reference variables.  In VBScript the IS operator is used to determine whether two object references refer to the same object. This differs from Windows PowerShell, in which the IS operator is used to determine whether a variable has a specified data type. (In other words, it all depends on what the definition of “is” is.)  That doesn’t mean that you can’t compare objects in Windows PowerShell; you can. It just means is that you need to use the **Compare-Object** Cmdlet to carry out this task. In the following example, we create two instances of the Scripting.Dictionary object. We then use Compare-Object to compare the two object references, tacking on the **–includeequal** parameter to ensure that we’ll get back an answer even if the two objects *are* identical:  $a = new-object -comobject scripting.dictionary  $b = new-object -comobject scripting.dictionary  $c = compare-object $a $b –includeequal  When you run this command and then echo back the value of $c you should get the following:  InputObject SideIndicator  ----------- -------------  {} ==  The == **SideIndicator** means that the two objects are equal.  Here’s a modified example in which the two objects are *not* equal: one references the Scripting.Dictionary object, while the other references the Scripting.FileSystem object. Here are the commands:  $a = new-object -comobject scripting.dictionary  $b = new-object -comobject scripting.filesystemobject  $c = compare-object $a $b –includeequal  And here’s what we get back when we run these commands and then echo the value of $c:  InputObject SideIndicator  ----------- -------------  System.\_\_ComObject =>  {} <=  Although it is somewhat meaningless when working with COM objects, the => and <= symbols indicate which of the two objects contain a given item (such as an individual line in a text file). |
| Mod operator (Mod) | **Definition:** Divides two numbers and returns only the remainder.  The Mod operator is a bit of an odd duck (although it *does* have its uses): it divides two numbers and then returns the remainder. In Windows PowerShell you can achieve the same effect by using the % operator. For example, this command divides 28 by 5, then stores the remainder in the variable $a:  $a = 28 % 5  When you run this command and then echo back the value of $a you should get the following:  3 |
| Multiplication operator (\*) | **Definition:** Multiplies two numbers.  If you need to multiply two numbers in Windows PowerShell you can do what pretty much everyone who needs to multiply two numbers does: just use the asterisk (\*) as the multiplication symbol. For example, this command multiplies 45 by 334 and stores the result in the variable $a:  $a = 45 \* 334  For those few of you who didn’t bother to do this equation in your head, $a should be equal to this:  15030 |
| Not operator (Not) | **Definition:** Performs logical negation on an expression.  Before you go any further remember, the Scripting Guys don’t make up these operators; we just report back what they do. The NOT operator is designed to test whether an expression is false. If an expression *is* false then the NOT operator reports back True. If an expression is true, then the NOT operator reports back False.  Like we said, we don’t make these things up, although, now that we think about it, this one *does* sound like something the Scripting Guys would come up with, doesn’t it?  For example, this command uses the NOT operator to evaluate the following expression **10 \* 7.7. = 77**:  $a = -not (10 \* 7.7 -eq 77)  Because this expression is true, if you run this command and echo back the value of $a you’ll get the following:  False |
| Or operator (OR) | **Definition:** Performs a logical disjunction on two expressions.  The OR operator evaluates two expressions and returns True if either of the expressions are true; you will get back the value False only if both parts are False. For example, suppose we want to evaluate the following two statements, then store the verdict in the variable $a:   * 23 is less than 12 * 12 is greater than 40   Here’s the command we’d use:  $a = 23 -lt 12 -or 12 -gt 40  As you might expect, if you run the command and then echo back the value of $a you’ll get the following, seeing as how both expressions are False:  False |
| Subtraction operator (-) | **Definition:** Finds the difference between two numbers or indicates the negative value of a numeric expression.  Actually, no, we *don’t* know what 22,018 minus 1,915 is. But Windows PowerShell can determine that difference for us; just use the minus sign (-) as the subtraction operator.  $a = 22018 - 1915  Well, what do you know? We were right: the answer is *20103*. |
| Xor operator (XOR) | **Definition:** Performs a logical exclusion on two expressions.  The XOR operator is a crazy one. XOR evaluates two expressions and returns True if the two are different; that is, if one expression is True and the other expression is False. If both expression are true (or false) then XOR returns False.  We told you it was crazy.  Here’s an example. This command evaluates the following two statements, then stores its verdict in the variable $a:   * 23 is less than 12 * 12 is greater than 40   Here’s the command  $a = 23 -gt 12 -xor 12 -lt 40  Because both expressions are true, if you run the command and then echo back the value of $a you’ll get the following:  False |