**Session - 01**

* PowerShell Data Types
* Naming
* Variables
* Read and Write (Input/Output)
* Operator
* Conditional Statement

**Extension of PowerShell file:** .ps1

**To run the script in Editor/IDE (integrated development environment)**: F5, F8 (run selected code)

**Comment line:** #

**Variable:**

$a =100

Here $a is variable which is temp. memory where data can be stored

**PowerShell Data Types**

The most common DataTypes used in PowerShell are listed below.

[string] Fixed-length string of Unicode characters

[char] A Unicode 16-bit character

[byte] An 8-bit unsigned character

[int] 32-bit signed integer

[long] 64-bit signed integer

[bool] Boolean True/False value

[decimal] A 128-bit decimal value

[single] Single-precision 32-bit floating point number

[double] Double-precision 64-bit floating point number

[DateTime] Date and Time

[xml] Xml object

[array] An array of values

[hashtable] Hashtable object

Naming

Let’s look at the basics first. An important property of a PowerShell variable is its name, which is always preceded by the dollar sign “$” and can only contain letters, numbers, and the underscore. If you feel a strong urge to use other characters, you have to enclose the name in curly braces. You should not use the name of variables that have been pre-defined (more about that later).

Here are examples of valid variable names:

*$myVariable, $MyVariable\_1, ${my-variable}*

And these are invalid names:

*myVariable, $my-variable, $my variable*

Strictly speaking, the variable name is the part after the dollar sign. This is important to know because when you have to specify the variable name as a parameter in a cmdlet, you have to enter it without the dollar sign. The dollar sign tells the shell to read the variable's value.

Values

You can assign values to a PowerShell variable by combining a variable name, an assignment operator, and an expression. Here is a simple example:

|  |  |
| --- | --- |
| 1 | $a = 1 + 1 |

The “=” sign is one of eight assignment operators. An expression is everything for which PowerShell can determine a value. If you enter an expression at a PowerShell prompt, PowerShell returns its value. “Hello world” and 1 are also expressions; determining their values just doesn’t need so many calculations.

If you want to populate multiple variables with the same value, you can save some typing as in the example below:

|  |  |
| --- | --- |
| 1 | $a = $b = $c = 1 |

You can also define multiple variables with different values on one line:

|  |  |
| --- | --- |
| 1 | $a, $b, $c = 1, 2, 3 |

This reduces the number of lines in your script but also makes it harder to read.

Other ways exist to store a value in a variable. We will see examples later.

To display the value of a variable, you don’t need a special command as in many other programming languages; entering the variable name is enough. This works in a script and on a command prompt.

|  |  |
| --- | --- |
| 1 | $c |

Thanks to the variable interpolation, you can also expand a variable in a string if the string is enclosed in double quotation marks.

|  |  |
| --- | --- |
| 1 | "These are the values of the variables: $a, $b, $c." |

[Interpolation example](https://4sysops.com/wp-content/uploads/2015/03/Interpolation-example.png)

If you want to display the variable names as text instead of displaying their values, you have to enclose the string in single quotation marks.

|  |  |
| --- | --- |
| 1 | 'These are the names of our variables $a, $b, $c.' |

[No interpolation with single qutation marks](https://4sysops.com/wp-content/uploads/2015/03/No-interpolation-with-single-qutation-marks.png)

Example:

$Number = Read-Host "Please enter a number"

$Square=$Number\*$Number

Write-Host "The square of the number $Number is $Square."

[Int]$Number = Read-Host "Please enter a number"

$Square=$Number\*$Number

Write-Host "The square of the number $Number is $Square."

[DateTime]$Date = "February 28, 2015"

$Today = Get-Date

$Days = ($Today - $Date).Days

Write-Host "The hacker encrypted all your servers $Days day(s) ago."

### The Arithmetic Operators

The arithmetic operators consist of the following characters: + (add), - (subtract), \* (multiply), / (divide), and % (modulus, or division remainder). These operators behave exactly as expected when using numbers. This means you can use PowerShell as a simple calculator by typing expressions at the PowerShell prompt. Figure 1 shows some simple examples.

### The Assignment Operators

The assignment operators assign values to variables or object properties. The most common assignment operator is =, which sets a variable or object property to a value. There are some other assignment operators as well: +=, -=, \*=, /=, %=, ++, and --. These other assignment operators modify the value before assigning it. The commands in Figure 2 demonstrate some of the assignment operators.

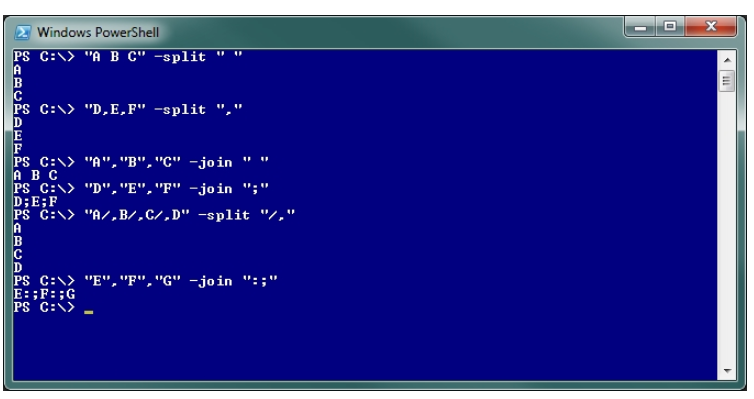
### The Comparison Operators

The comparison operators compare values in PowerShell. There are four kinds of comparison operators: equality, match, containment, and replace. Let's take a look at each of these.

**Equality operators.** The equality operators are equal (-eq), not equal (-ne), greater than (-gt), greater than or equal (-ge), less than (-lt), and less than or equal (-le). PowerShell doesn't use an equals sign (=) to test equality because it's used for the assignment operator. Similarly, PowerShell doesn't use the greater than (>) or less than (<) characters because they're used for output and input redirection, respectively.

**The Split and Join Operators**

The -split operator splits a string into an array of substrings. The -join operator is its inverse—it joins an array of strings into a single string. The -split and -join operators make converting a string into an array (or vice versa) a snap. Both operators let you specify one or more delimiter characters that define how PowerShell should split or join the string. Figure 4 shows some examples of how you can use these operators.



**IF, ELSE, SWITCH: Conditional statements in PowerShell**

if statement in PowerShell

The following rules apply for *if* statements in PowerShell:

* PowerShell commands are case-insensitive, so you can use *If*, *if*, or *IF*. The same applies to *else*.
* The boolean condition has to be in parentheses.
* The depending statement must be in braces.

In the simplest situation, the condition only contains one variable that has to be tested whether it is defined or unequal to zero. If this is the case, the test of the boolean condition evaluates in TRUE; otherwise, it is FALSE.

|  |  |
| --- | --- |
| 1 | if($test) {echo "Value of test: " $test} |

Conditional statements and comparison operators

Usually, you will require more complex conditions for which you need [comparison operators](https://4sysops.com/archives/powershell-comparison-operators-eq-lt-gt-contains-like-match/). However, if you just want to negate one boolean variable, you can use the unary *–NOT* operator:

if(-NOT $test) {echo "Value of test is zero or undefined."}

Rules for conditional statements [^](https://4sysops.com/archives/if-else-switch-conditional-statements-in-powershell/#Content-bal-title)

Common syntactical rules apply for the statements that depend on the condition. In contrast to JavaScript or PHP, if each statement is on a separate line, you don’t have to end the line with a semicolon. However, in an interactive shell, if you want to pack multiple statements on a single line, you have to separate them with semicolons:

|  |  |
| --- | --- |
| 1 | if(Test-Path \*.gif){gci \*.gif|foreach{$len += $\_.length}; Write-Host $len " Bytes"} |

In distinction from VBScript, you don’t need an *End If* in PowerShell. This is also true if an *else* branch follows the *if* statement. The *else* branch has the same syntax except that it doesn’t require a condition. After *else* follows the block that will be executed if the *if* condition doesn’t apply:

|  |  |
| --- | --- |
| 1  2 | if($test) {echo "Value of test: " $test}  else {echo "Value of test is zero or undefined"} |

Multiple conditions with elseif [^](https://4sysops.com/archives/if-else-switch-conditional-statements-in-powershell/" \l "Content-bal-title" \o "Back to table of contents)

In those cases where an *if* statement plus *else* branch is insufficient, you can combine multiple conditions. PowerShell offers the *elseif* keyword for this purpose.

If the evaluation of the *if* condition results in FALSE, all *elseif* conditions will be tested afterwards. If one of the *elseif* conditions evaluates in TRUE, the depending block will be executed, and PowerShell will exit the entire *if-else-elseif* block. If neither the *if* condition nor one of the *elseif* condition is true, the optional closing *else* condition will be tested.

|  |  |
| --- | --- |
| 1  2  3  4 | if (condition 1) {command}  elseif (condition 2) {command}  elseif (condition 3) {command}  else {command} |

switch as alternative to long elseif lists [^](https://4sysops.com/archives/if-else-switch-conditional-statements-in-powershell/#Content-bal-title)

Instead of using long *if-elseif* chains, it is more elegant to switch to *switch* (sorry for the pun). *switch* follows the same pattern as all languages that were syntactically derived from C, such as PHP or JavaScript. However, in PowerShell, *switch* is used for testing equality only. Thus, you can’t use comparison operators in *switch* statements.

The value to be tested, the so-called control expression or control variable, must be enclosed in parentheses right after the *switch* keyword, but the value (also called case) will only be compared with the control variable in the following block:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | switch(Read-Host "Select a menu item"){      1 {"File will be deleted"}      2 {"File will be displayed"}      3 {"File is write protected"}      default {"Invalid entry"}  } |

If you enter 1, 2, or 3, the corresponding message will be displayed; otherwise, you’ll see “Invalid entry.”

PowerShell tests all switch cases [^](https://4sysops.com/archives/if-else-switch-conditional-statements-in-powershell/#Content-bal-title)

If a match occurs, the corresponding instruction will be executed; however, PowerShell will still run through the remaining cases. If another value matches, the corresponding instruction will be executed as well.

If you want to avoid this behavior, you have to end the instruction with *break*. PowerShell will then exit the block whenever a value matches.

In the above example, this would look like this:

|  |  |
| --- | --- |
| 1 | 1 {"File will be deleted"; break} |

Capitalization, wildcards, regex [^](https://4sysops.com/archives/if-else-switch-conditional-statements-in-powershell/#Content-bal-title)

By default, the comparison of strings is case-insensitive. If you don’t want this, you can add the operator -*CaseSensitive* to *switch*.

A specific feature of PowerShell is that you can use wildcards and regular expressions if you compare strings. For this purpose, you have to add the operators *-wildcard* and *-regex*.

|  |  |
| --- | --- |
| 1  2  3  4  5 | switch -wildcard("PowerShell"){      "Power\*" {echo "'\*' stands for 'shell'"}      \*ersh\*" {echo "'\*' replaces 'Pow' and 'ell'"}      "PowerShe??" {echo "Pattern matches because ?? replaces two 'l' "}  } |

The wildcard “\*” stands for multiple arbitrary characters, whereas “?” is just for one character. You can work with regular expressions in a similar way as with wildcards; you just have to add the operator -*regex*.