CNT 4603: System Administration
Spring 2013

Scripting – Windows PowerShell – Part 6

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 4078-823-2790
http://www.cs.ucf.edu/courses/cnt4603/spr2013

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

The PowerShell Language

- The designers of PowerShell drew extensively on existing scripting and programming languages to ensure that PowerShell incorporated the most useful features of many different languages.
- PowerShell is very similar to many of the programming languages that you are already familiar with in the areas of expressions, operators, control statements, and functional capabilities. We'll look only briefly at these concepts in PowerShell, more to illustrate syntax than any other purpose.
- We'll focus in this section of notes more heavily on the new and unique features of PowerShell that are not found in many other programming or scripting languages.

Expressions And Operators In PowerShell

Expressions

- An expression in PowerShell can be understood as a calculation that evaluates an equation and returns a result.
- An operator in PowerShell is the element of an expression that actually performs the calculation (such as addition or subtraction).
- PowerShell has three general categories of expressions:
  - Arithmetic expressions – expressions that return a numerical result.
  - Assignment expressions – used to set, display, and modify variables.
  - Comparison expressions – use Boolean logic to compare two values and return a true or false result.
- The following page illustrates an example of each type of expression.

C:\Users\Administrator\MyScripts\PS-Part6-p3.ps1 - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins

PS-Part6-p3.ps1

```
1  #---------------------------------------------
2  # PS-Part6-p3
3  # PowerShell Notes - Part 6 Page 3 - Expres
4  #
5  #---------------------------------------------
6
7  #---------------------------------------------
8  # Script Body
9  #---------------------------------------------
10
11 write-host " "write-host "Arithmetic Expression"  -Foregroundcolor Green
12 write-host "1 + 1 = " -Foregroundcolor Red -NoNewLine
13 1 + 1
14 write-host " "
15 write-host "Assignment Expression" -Foregroundcolor Green
16 $myString1 = "This is an example test string"
17 write-host '$myString1 = ' $myString1 -Foregroundcolor Red
18 write-host " "
19 write-host "Comparison Expression" -Foregroundcolor Green
20 $myString2 = "This is a second example test string"
21 write-host "Are the two strings equal? " -Foregroundcolor Red -NoNewLine
22 $myString1 -eq $myString2
23 write-host " "
24 #------------------------------------------------------------------------
25 # END SCRIPT
26 #------------------------------------------------------------------------
```

Select Administrator: Windows PowerShell

```
PS C:\users\Administrator\MyScripts>
PS C:\users\Administrator\MyScripts>
PS C:\users\Administrator\MyScripts>
PS C:\users\Administrator\MyScripts> .\PS-Part6-p3
  write-host Arithmetic Expression
1 + 1 = 2

Assignment Expression
$myString1 =  This is an example test string

Comparison Expression
Are the two strings equal? False

PS C:\users\Administrator\MyScripts> _
```

MyScripts | C:\Users\Administrat... | Select Administrator: Wi...

Expressions And Operators In PowerShell

Operators

- PowerShell contains six basic types of operators:

1. Arithmetic operators: `+, *, -, /, %`
2. Assignment operators: `=, +=, -=, *=, /=, %=`
3. Comparison operators: `-eq, -ne, -gt, -ge, -lt, -le, -contains, -notcontains`
4. Pattern matching operators: `-like, -notlike`
5. Regular expression operators: `-match, -notmatch, -replace`
6. Logical and bitwise operators: `-and, -or, -xor, -not, -band, -bor, -bxor, -bnot`

Expressions And Operators In PowerShell

- Each of the comparison, pattern matching, and regular expression operators also include a "c" and "i" prefixed version. The "c" prefix, as in `–ceq`, indicates case sensitivity, while the "i" prefix, as in `–ieq`, indicates case insensitivity.

- The pattern matching and regular expression operators also have an extensive list of characters that are used with them; we'll see these later when looking at these types of operators more closely.

- The next page illustrates a script that uses several different types of PowerShell operators.

C:\Users\Administrator\MyScripts\PS-Part6-p7.ps1 - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

web.xml | config.php | index.php | PS-Part6-p7.ps1

```
1  #---------------------------------------------------------------------------
2  # PS-Part6-p3
3  # PowerShell Notes - Part 6 Page 7 - Various Operators in PowerShell
4  #
5  #---------------------------------------------------------------------------
6
7  #---------------------------------------------------------------------------
8  # Script Body
9  #---------------------------------------------------------------------------
10 write-host " "
11 write-host "Various PowerShell Operators"  -Foregroundcolor Green
12 write-host " "
13 $myVar = 4
14 $myVar += 2
15 write-host 'If $myVar = 4, then  myVar += 2' "produces $myVar" -Foregroundcolor Red
16 write-host " "
17 $myVar2 = 4
18 $myVar3 = 8
19 write-host "Is $myVar2 less than $myVar3 : " -Foregroundcolor Green -NoNewLine
20 $myVar2 -lt $myVar3
21 write-host " "
22 write-host "Is $myVar2 greater than $myVar3 : " -Foregroundcolor Green -NoNewLine
23 $myVar2 -gt $myVar3
24 write-host " "
25 write-host "Is apple like a?p?e : "  -Foregroundcolor Green
26 "apple" -like "a?p?e"
```

Windows Po | length : 1776 lines : 44 | Ln : 42 Col : 77 Sel : 0 | Dos\Windows | ANSI | INS

Administrator: Windows PowerShell

```
True
PS C:\users\Administrator\MyScripts> .\PS-Part6-p7.ps1

Various PowerShell Operators

If $myVar = 4, then  myVar += 2 produces 6

Is 4 less than 8 : True

Is 4 greater than 8 : False

Is apple like a?p?e :
True
Is apple like [a-c]pple :
True
Is apple like a* :
True
Is apple like [a][p][p][l][e] :
True
Is apple not like [a][p][p][l][e] :
False
Is apple notlike [a][p][l][e] :
True
Is hello -eq to Hello :
True
Is hello -ceq to Hello :
False
Is hello -ieq to Hello :
True
Is Hello -ceq to Hello :
True
Is Hello -gt than hello :
False
Is Hello -igt than hello :
False
Is Hello -cgt than hello :
True
PS C:\users\Administrator\MyScripts> _
```

Pattern Matching Operators In PowerShell

- Pattern matching operators in PowerShell use Boolean logic to compare two strings of text and return a result.
- To provide more granular comparison capabilities, the pattern matching operators work in conjunction with a set of wildcard operators (characters) to create patterns for matching.
- Four different wildcard operators can be used with the pattern matching operators in PowerShell:
  - *, matches any pattern. E.g., "apple" –like "a*" returns true
  - ?, matches any single character. E.g., "apple" –like "a?p?e" returns true
  - [x-y], matches a range of characters. E.g, "apple" –like "[a-c]pple" returns true
  - [xy], matches any one of the specified characters. E.g., "apple" –like [a][p][p][l][e] returns true.

Regular Expression Operators In PowerShell

- Regular expression operators are closely related to pattern matching operators in PowerShell. If you've ever used Perl, Python (you're about to), or PHP, they will be familiar to you, but I'm assuming that you are not very familiar with any of these other languages.
- Regular expression operators are more flexible than the pattern matching operations. Regular expression operations support the same type of syntax and wildcards as pattern matching operations, but the wildcard operators used with regular expressions are different from the pattern matching wildcard operators.
- A listing of some of the wildcard operators for regular expression operators are shown in the tables on the next two pages. These do not comprise a complete listing.

Regular Expression Operators In PowerShell

Character (operator)	Description	Example
None	Matches exact characters or character sequence anywhere in the original value	“apple” –match “pl” → true
.	Matches any single character	“apple” –match “a…e” → true
[value]	Matches at least 1 of the characters between the brackets	“apple” –match “[pear]” → true
[range]	Matches at least 1 of the characters within the specified range	“apple” –match “a[a-z]ple“→ true
[^]	Matches any character except those in the brackets	“apple” –match “[^fig]” → true
^	Performs a match of the specified characters starting at the beginning of the original value	“apple” –match “^app” → true
$	Performs a match of the specified characters starting at the end of the original value.	“apple” –match “ple$” → true

Regular Expression Operators In PowerShell

Character (operator)	Description	Example
*	Matches any pattern	"apple" –match "a*" → true
?	Matches a single character in the string	"apple" –match "a?ple" → true
+	Matches a repeating instance of the preceding character	"apple" –match "ap+le" → true
\	Denotes the character following the backslash as an escaped character	"apple$" –match "apple\$" → true
\w	Matches any word character	"abcd defg" –match "\w+" matches abcd" → true
\s	Matches any white-space character	"abcd defg" –match "\s+" matches abcd" → true
\d	Matches any decimal digit	12345 –match "\d+" → true
{n}	Specifies exactly n matches	"abc" –match "\w{2}" → true "abc" – match "\w{4}" → false

Advanced Operators In PowerShell

- In addition to the basic operators in PowerShell, there are also several different advanced operators.
- These advanced operators include type operators, unary operators, grouping operators, array operators, property and method operators, format operators, and redirection operators.
- We'll look briefly at each of these categories of advanced operators.

Advanced Operators In PowerShell

Type operators

- These operators serve as a method to compare types of two different objects and return a Boolean true or false value.
- There is also an operator to change the type of an object to another type.

Unary operators

- Similar to arithmetic operators, they include: +, -, ++, --, |<type>|, and , (the comma).
- The script on the next page illustrates examples of both of these types of advanced operators.

C:\Users\Administrator\MyScripts\PS-Part6-p15.ps1 - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins

PS-Part6-p15.ps1

```
5   #--------------------------------------------
6
7   #--------------------------------------------
8   # Script Body
9   #--------------------------------------------
10  write-host " "
11  write-host "Advanced PowerShell Operators"  -
12  write-host " "
13  $myString1 = "This is an example string."
14  write-host 'Is $myString1 a string :' -Foregroundcolor Red -NoNewLine
15  $myString1 -is [string]
16  write-host " "
17  write-host 'Is $myString1 not a string :' -Foregroundcolor Red -NoNewLine
18  $myString1 -isnot [string]
19  write-host " "
20  $myInt = "2356"
21  $myInt + 1    # $myInt is a string now, so + is concatenation
22  $myInt -as [int]
23  2356 + 1  # $myInt is an integer now, so + is addition
24  write-host " "
25  $a = 3 * 6
26  -$a
27  --$a
28  ++$a
29  [int]"0xCABCAB"
30  write-host " "
31
```

Windows PowerShell | length : 1197 lines : 35 | Ln : 30 Col : 15 Sel : 0 | Dos\Windows | ANSI | INS

Administrator: Windows PowerShell

```
PS C:\users\Administrator\MyScripts> .\PS-Part6-p15

Advanced PowerShell Operators

Is $myString1 a string :True

Is $myString1 not a string :False

23561
2356
2357

-18
13286571

PS C:\users\Administrator\MyScripts>
```

Advanced Operators In PowerShell

Grouping operators

- These operators are used to bring together a set of terms and perform an operation against these terms.
- The three types available in PowerShell are parentheses (used to group expression operators), the subexpressions grouping operator (the $ used to group together collections of statements, and the array subsexpressions operator (the @ symbol used to group together collections of statements and insert the results into an array).
- Each of these grouping operators is illustrated with examples in the script on the next page.

C:\Users\Administrator\MyScripts\PS-Part6-p17.ps1 - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

PS-Part6-p17.ps1

```
#----------------------------------------------------------------------------
# PS-Part6-p17
# PowerShell Notes - Part 6 Page 17 - Grouping Operators in PowerShell
#
#----------------------------------------------------------------------------

#----------------------------------------------------------------------------
# Script Body
#----------------------------------------------------------------------------
write-host " "
write-host "Grouping PowerShell Operators"  -Foregroundcolor Green
write-host " "
write-host "Parentheses" -Foregroundcolor Red
(4 + 7) * 4
write-host " "
write-host "Subexpression operator - $" -Foregroundcolor Red
$($a = "c*"; get-process $a)
write-host " "
write-host "Array subexpression operator - @" -Foregroundcolor Red
@(get-date; get-executionpolicy)
write-host " "
#----------------------------------------------------------------------------
# END SCRIPT
#----------------------------------------------------------------------------
```

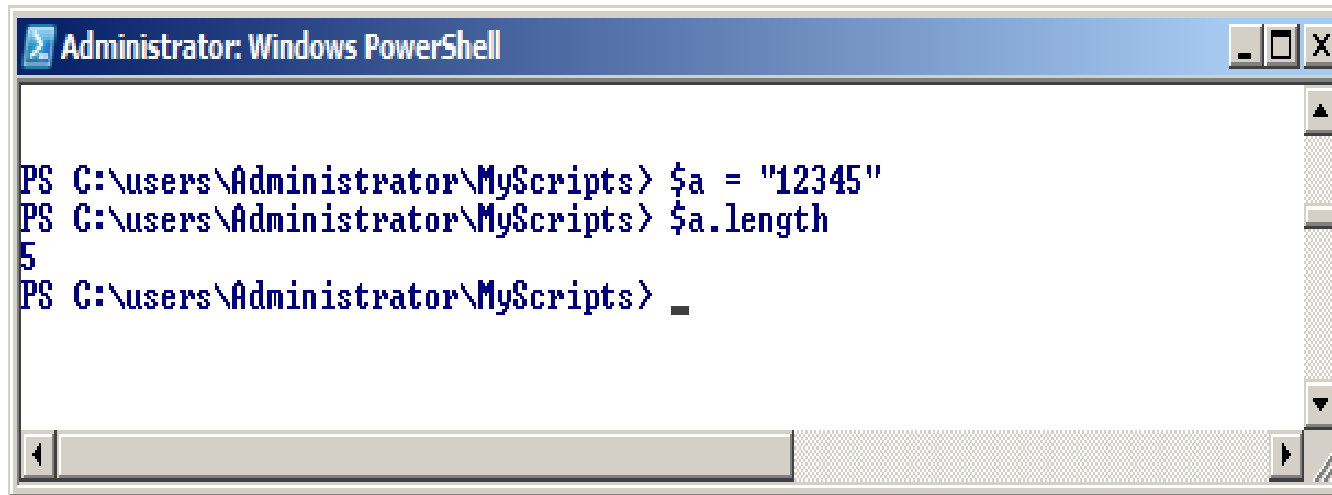
Windows PowerShell | length : 990 lines : 24 | Ln : 24 Col : 77 Sel : 0 | Dos\Windows | ANSI | INS

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\users\Administrator\MyScripts> .\PS-Part6-p17

Grouping PowerShell Operators

Parentheses
44

Subexpression operator - $

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    612       5     1624       5088   104     1.22    492 csrss
    207       7     6652       6716   110     3.75    536 csrss

Array subexpression operator - @

DisplayHint : DateTime
Date        : 11/7/2012 12:00:00 AM
Day         : 7
DayOfWeek   : Wednesday
DayOfYear   : 312
Hour        : 11
Kind        : Local
Millisecond : 737
Minute      : 46
Month       : 11
Second      : 7
Ticks       : 634878855677376000
TimeOfDay   : 11:46:07.7376000
Year        : 2012
DateTime    : Wednesday, November 07, 2012 11:46:07 AM

RemoteSigned


PS C:\users\Administrator\MyScripts>
```

Advanced Operators In PowerShell

Property and method operators

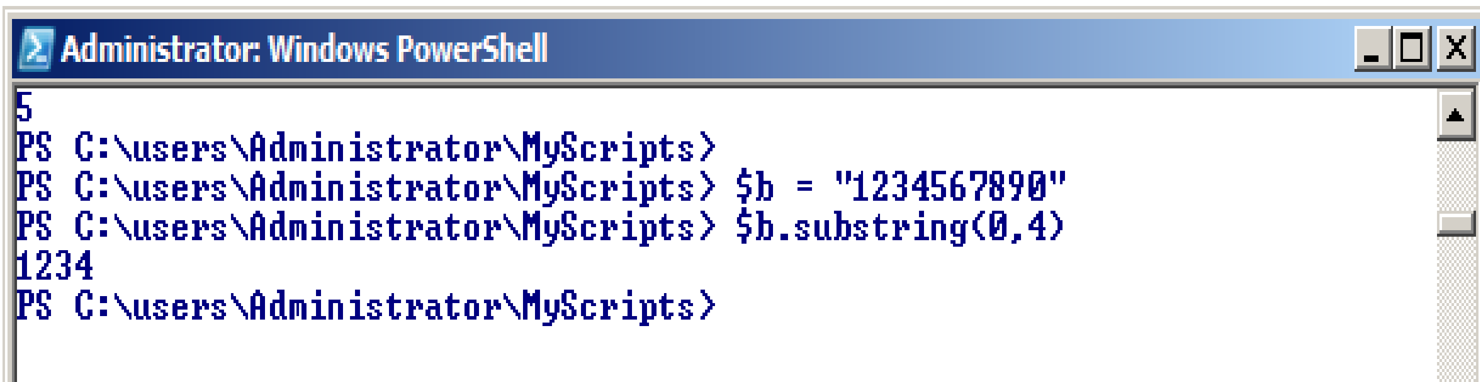
- These operators are very commonly used operators in PowerShell.
- The basic property operator is the period (.), which is used to access properties of a variable.



Advanced Operators In PowerShell

Property and method operators

- The method operators are frequently used to in conjunction with the property operators to provide finer control over the data that is returned from a property query.
- The basic syntax for using method operators with the period property operator is shown below:




```
<variable-name>.<method-name>(method-argument1, method-argument2, … )
```

Advanced Operators In PowerShell

NOTE

When you are working with properties of a variable, one of the helpful features of PowerShell is the **tab completion** feature (also see pages 6 & 7 of PowerShell Part 3 notes). Tab completion enables you to enter the variable name, the period property operator, and then press the **Tab** key to cycle through the available properties for that variable.

Try setting a variable in a PowerShell session, enter just the variable name and a period (as in $a.), and then press the **Tab** key repeatedly to view the property options for that variable. Note that the available properties will be dependent on the variable's type.

Advanced Operators In PowerShell

Property and method operators

- The second most commonly used method operator is the double colon (::). This is used to access members of a specific class of objects, and it is technically referred to as a static member accessor. (Thus, the left argument to a :: must be a valid type name, and the right argument must be a valid member name for the class on the left.)
- An example would be: `[string]::Equals`
- To obtain a list of valid methods for a given type, you can use the command: `[<type-name>] | get-member -static`
- The next page illustrates this command.

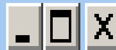
Administrator: Windows PowerShell

```
PS C:\users\Administrator\MyScripts> [char] | get-member -static


   TypeName: System.Char

Name                MemberType Definition
----                ---------- ----------
ConvertFromUtf32    Method     static string ConvertFromUtf32(int utf32)
ConvertToUtf32      Method     static int ConvertToUtf32(char highSurrogate, char lowSurrogate), stat
Equals              Method     static bool Equals(System.Object objA, System.Object objB)
GetNumericValue     Method     static double GetNumericValue(char c), static double GetNumericValue(s
GetUnicodeCategory  Method     static System.Globalization.UnicodeCategory GetUnicodeCategory(char c)
IsControl           Method     static bool IsControl(char c), static bool IsControl(string s, int ind
IsDigit             Method     static bool IsDigit(char c), static bool IsDigit(string s, int index)
IsHighSurrogate     Method     static bool IsHighSurrogate(char c), static bool IsHighSurrogate(strin
IsLetter            Method     static bool IsLetter(char c), static bool IsLetter(string s, int index
IsLetterOrDigit     Method     static bool IsLetterOrDigit(char c), static bool IsLetterOrDigit(strin
IsLower             Method     static bool IsLower(char c), static bool IsLower(string s, int index)
IsLowSurrogate      Method     static bool IsLowSurrogate(char c), static bool IsLowSurrogate(string
IsNumber            Method     static bool IsNumber(char c), static bool IsNumber(string s, int index
IsPunctuation       Method     static bool IsPunctuation(char c), static bool IsPunctuation(string s,
IsSeparator         Method     static bool IsSeparator(char c), static bool IsSeparator(string s, int
IsSurrogate         Method     static bool IsSurrogate(char c), static bool IsSurrogate(string s, int
IsSurrogatePair     Method     static bool IsSurrogatePair(string s, int index), static bool IsSurrog
IsSymbol            Method     static bool IsSymbol(char c), static bool IsSymbol(string s, int index
IsUpper             Method     static bool IsUpper(char c), static bool IsUpper(string s, int index)
IsWhiteSpace        Method     static bool IsWhiteSpace(char c), static bool IsWhiteSpace(string s, i
Parse               Method     static char Parse(string s)
ReferenceEquals     Method     static bool ReferenceEquals(System.Object objA, System.Object objB)
ToLower             Method     static char ToLower(char c, System.Globalization.CultureInfo culture),
ToLowerInvariant    Method     static char ToLowerInvariant(char c)
ToString            Method     static string ToString(char c)
ToUpper             Method     static char ToUpper(char c, System.Globalization.CultureInfo culture),
ToUpperInvariant    Method     static char ToUpperInvariant(char c)
TryParse            Method     static bool TryParse(string s, System.Char&, mscorlib, Version=2.0.0.0
MaxValue            Property   static System.Char MaxValue {get;}
MinValue            Property   static System.Char MinValue {get;}


PS C:\users\Administrator\MyScripts> _
```

Advanced Operators In PowerShell

<u>Property and method operators</u>

- Once you have identified the available methods for a particular type, you can select the particular method that you want to use for that type, and then access it using the double colon operator.
- The examples on the next page illustrates this by defining a variable `$c` set to a capital `A`, and then the `char` type method `ToLower` being called against the `$c` variable to convert the variable from upper case to lower case.
- The second example shows the variable `$d` begin set to lower case `a` and then the `char` method `ToUpper` being called to convert this variable from lower to upper case.
- Some additional method operations are also illustrated.



Administrator: Windows PowerShell

```
PS C:\users\Administrator\MyScripts> $c = "A"
PS C:\users\Administrator\MyScripts> [char]::ToLower($c)
a
PS C:\users\Administrator\MyScripts> $d = "a"
PS C:\users\Administrator\MyScripts> $e = [char]::ToUpper($d)
PS C:\users\Administrator\MyScripts> $e
A
PS C:\users\Administrator\MyScripts> [char]::IsLower($e)
False
PS C:\users\Administrator\MyScripts> [char]::IsUpper($e)
True
PS C:\users\Administrator\MyScripts> [char]::Equals($d,$e)
False
PS C:\users\Administrator\MyScripts> [char]::IsLetterOrDigit($d)
True
PS C:\users\Administrator\MyScripts> [char]::IsNumber($e)
False
PS C:\users\Administrator\MyScripts> [char]::IsPunctuation($d)
False
PS C:\users\Administrator\MyScripts> [char]::IsPunctuation(",")
True
PS C:\users\Administrator\MyScripts>
```

Advanced Operators In PowerShell

Format operator

- The format operator in PowerShell is used to provide more granular control over PowerShell's output.
- The basic structure of a format operator statement includes a format string on the left and an array of values on the right, as shown in the following example:

```
'{0} {1} {2}' – f 1, 2, 3
```

- The values on the right side of a format operator statement are not limited to being numeric (character and string values are also supported). However, most of the advanced format string arguments operate primarily against numerical values.

Advanced Operators In PowerShell

<u>Format operator</u>

- At the basic level, the format operator is used to control the order in which element of an array are displayed, or which elements are displayed at all. See the example on page 29.

- However, the format operator can also provide a tremendous number of other output options, simply by applying different arguments to the format strings.

- The general syntax for format string arguments is: `{0:argument}` where 0 is replaced by the array element that you are interested in, and argument is replaced by the format string argument that you want to apply to the data in the array element. The table on the next page illustrates some of the possible arguments for format strings.

Advanced Operators In PowerShell

Format Element String	Description	Example
`{0}`	General format string argument; displays the specified element exactly as entered.	`'{0}' -f "myText"` returns `MyText`
`{0:x}`	Displays the specified element in hexadecimal format, with the alpha-numeric characters displayed in lower case.	`'{0:x}' -f 12345678` returns `bc614e`
`{0:p}`	Displays the specified element as a percentage	`'{0:p}' -f .372` returns `32.70%`
`{0:C}`	Displays the specified element in currency format	`'{0:C}' -f 15.78` returns `$15.78`
`{0:hh}`, `{0:mm}`	Returns the two-digit hour and two-digit minute value form a `get-date` command	`'{0:hh}:{0:mm}' - f (get-date)` returns something like `08:45`

Some Format String Arguments

C:\Users\Administrator\MyScripts\PS-Part6-p29.ps1 - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

PS-Part6-p29.ps1 | Get-StoppedService.ps1

```
11  write-host "PowerShell Formatting Operators"  -Foregroundcolor Green
12  write-host " "
13  write-host "Basic case - used for ordering/appearance" -Foregroundcolor Black
14  write-host "Commmand is: '{2} {1} {0}' -f "apples","oranges","cherries" " -Foregroundcolor R
15  write-host "Output is: " -Foregroundcolor Black -NoNewLine
16  '{2} {1} {0}' -f "apples","oranges","cherries"
17  write-host " "
18  write-host "Command is: '{0} {2} {1}' -f "123","456","789" " -Foregroundcolor Red
19  write-host "Output is: " -Foregroundcolor Black -NoNewLine
20  '{0} {2} {1}' -f "123","456","789"
21  write-host " "
22  write-host "Command is: '{1}' -f "Item 1", "item 2", "Item 3" " -Foregroundcolor Red
23  write-host "Output is: " -Foregroundcolor Black -NoNewLine
24  '{1}' -f "Item 1", "item 2", "Item 3"
25  write-host " "
26  write-host "Command is: '{0:x}' -f 12345678 " -Foregroundcolor Red
27  write-host "Output is: " -Foregroundcolor Black -NoNewLine
28  '{0:x}' -f 12345678
29  write-host " "
30  write-host "Command is: '{0:p}' -f 0.456 " -Foregroundcolor Red
31  write-host "Output is: " -Foregroundcolor Black -NoNewLine
32  '{0:p}' -f 0.456
33  write-host " "
34  write-host "Command is: '{0:C}' -f 35.94 " -Foregroundcolor Red
35  write-host "Output is: " -Foregroundcolor Black -NoNewLine
36  '{0:C}' -f 35.94
```

Windows PowerShell | length : 2054 lines : 44 | Ln : 43 Col : 13 Sel : 0 | Dos\Windows | ANSI | INS

Administrator: Windows PowerShell

```
PowerShell Formatting Operators

Basic case - used for ordering/appearance
Commmand is: '{2} {1} {0}' -f  apples,oranges,cherries
Output is: cherries oranges apples

Command is: '{0} {2} {1}' -f  123,456,789
Output is: 123 789 456

Command is: '{1}' -f  Item 1, item 2, Item 3
Output is: item 2

Command is: '{0:x}' -f 12345678
Output is: bc614e

Command is: '{0:p}' -f 0.456
Output is: 45.60 %

Command is: '{0:C}' -f 35.94
Output is: $35.94

Command is: '{0:hh}:{0:mm}' -f (get-date)
Output is: 11:35

PS C:\users\Administrator\MyScripts>
```

Advanced Operators In PowerShell

Format operator

- The number of different format operators is extensive and far too lengthy to cover here.
- MSDN provides a comprehensive reference on the syntax for the use of arguments in .NET Framework formatting strings in the .NET Framework Developer's Guide in the section on Formatting Types. You can find this guide at: www.msdn.microsoft.com and searching for "formatting types".

Advanced Operators In PowerShell

<u>Redirection operators</u>

- The final advanced operator type in PowerShell is the redirection operators.

- Redirection operators are used to direct command output to another location, such as a file.

- The example on the next page illustrates redirecting the output of a `get-process s*` command to a textfile named `s-processes.txt`.

NOTE: One major difference in PowerShell's redirection operations versus other shells is that the < (input redirection) operator is currently not implemented. A syntax error is returned if you attempt to use an input redirection operator in a PowerShell command.

C:\Users\Administrator\MyScripts\PS-Part6-p33.ps1 - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

PS-Part6-p33.ps1 | Get-StoppedService.ps1

```
1  #------------------------------------------------------------------------------
2  # PS-Part6-p33
3  # PowerShell Notes - Part 6 Page 33 - Redirection in PowerShell
4  #
5  #------------------------------------------------------------------------------
6
7  #------------------------------------------------------------------------------
8  # Script Body
9  #------------------------------------------------------------------------------
10 write-host " "
11 write-host "PowerShell Redirection Operators"  -Foregroundcolor Green -Backgroundcolor Black
12 write-host " "
13 write-host "Command is: get-process s* > s-processes.txt" -Foregroundcolor Black
14 get-process s* > s-processes.txt
15 write-host " "
16 write-host "Contents of file containing redirected output: " -Foregroundcolor Black
17 get-content s-processes.txt
18 write-host " "
19 #------------------------------------------------------------------------------
20 # END SCRIPT
21 #------------------------------------------------------------------------------
```

Windows PowerShell | length : 969 lines : 21 | Ln : 16 Col : 30 Sel : 0 | Dos\Windows | ANSI | INS

Administrator: Windows PowerShell

PowerShell Redirection Operators

```
Command is: get-process s* > s-processes.txt

Contents of file containing redirected output:

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    240       6     2064       6096    36     2.30    628 services
     95       3     5388       9604    39     1.17   1024 SLsvc
     28       1      256        724     4     0.11    428 smss
    308      10     6676      12168    99     1.97   1520 spoolsv
    297       4     2592       6140    38     3.13    808 svchost
    258       7     2592       5992    35     0.36    868 svchost
    288       9     5512       8408    45     5.16    944 svchost
    148       4     2856       5772    35     0.03    988 svchost
   1019      36    37884      46316   178     4.94   1004 svchost
    577      16     5904      10800    58     0.83   1080 svchost
    248       8     7004       8496    66     0.42   1132 svchost
    409      14    14400      15472    87     1.95   1184 svchost
    267      22     5952       9904    46     0.48   1324 svchost
    125       5     1856       5240    36     0.14   1768 svchost
     73       2      828       2856    23     0.03   1780 svchost
     44       1      540       2264    15     0.03   1928 svchost
    226       7     3176       4904    50     0.11   3380 svchost
    497       0        0       1852     4               4 System

PS C:\users\Administrator\MyScripts> _
```

Advanced Operators In PowerShell

Redirection Operator	Description
>	Redirects the output of a command to the specified file. The specified file is overwritten.
>>	Redirects the output of a command to the specified file. If the specified file exists, the output of the command is appended to the existing file.
2>	Redirects any errors generated by a command to the specified file. The specified file is overwritten if it exists.
2>>	Redirects any errors generated by a command to the specified file. If the specified file exists, any errors generated by the command are appended to the existing file.
2>&1	Redirects any errors generated by a command to the output pipe (displaying the errors at the console) instead of redirecting to a file.

Redirection Operators In PowerShell

Escape Sequences In PowerShell

- The grave-accent or backtick (`) acts as the PowerShell escape character. Depending on when this character is used, PowerShell interprets characters immediately following it in a certain way.
- If the backtick character is used at the end of a line in a script, it acts as a continuation character. In other words, it allows you to break long lines of code into smaller chunks.
- If the backtick character precedes a PowerShell variable, the characters immediately following it should be passed on without substitution or processing.
- If the backtick character is used in a string or interpreted as part of a string, that means the next character is interpreted as a special character. For example, if you want to put a TAB in your string, you use the `` `t `` escape character. See next page for examples of each.

*C:\Users\Administrator\MyScripts\PS-Part6-p37.ps1 - Notepad++

File Edit Search View Encoding Language Settings

PS-Part6-p37.ps1 | Get-StoppedService.ps1

```
1  #-----------------------------
2  # PS-Part6-p37
3  # PowerShell Notes - Part 6 Pa
4  #
5  #-----------------------------
6
7  #-----------------------------
8  # Script Body
9  #-----------------------------
10 write-host " "
11 write-host "PowerShell Excape Sequences"  -Foregroundcolor Green -Backgroundcolor Black
12 write-host " "
13 write-host "Continuing a very long command" -Foregroundcolor Black
14 $reg = get-process a* | `
15 format-table
16 $reg
17 write-host " "
18 write-host "Preceding a variable " -Foregroundcolor Black
19 $string1 = "Is this working?"
20 write-host "The string is: $string1"
21 write-host "The string is: `$string1 the line continues."
22 $string2 = "This line contains a tab: `t [TAB] character."
23 $string2
24 $string3 = "This line contain not 1 `t [TAB], but 2 `t [TAB] characters."
25 $string3
26 write-host " "
```

Windows PowerShell | length : 1196 lines : 29 | Ln : 7 Col : 1 Sel : 0 | Dos\Windows | ANSI | INS

Administrator: Windows PowerShell

```
PowerShell Excape Sequences

Continuing a very long command

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     41       2      952       3084    46     0.08   2860 ApacheMonitor


Preceding a variable
The string is: Is this working?
The string is: $string1 the line continues.
This line contains a tab:         [TAB] character.
This line contain not 1          [TAB], but 2    [TAB] characters.

PS C:\users\Administrator\MyScripts> _
```

Escape Character Sequences Supported By PowerShell

Character Sequence	Meaning
`’	Single quotation mark
`”	Double quotation mark
`0	Null character
`a	Alert (bell or beep signal to computer’s speaker
`b	backspace
`f	Form feed (used only for printer output)
`n	Newline
`r	Carriage return
`t	Horizontal tab (8 spaces)
‘v	Vertical tab (used only for printer output)