Advanced Scripting   
Expressions

Last Updated: 12/23/2022 11:51 AM Version 1  
Document Prepared for: CIT361 Student

# Name Daniel Harris ID 235868292

# Instructions

Save a copy of this document. Answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

Expressions are the backbone of PowerShell. Without the ability to evaluate expressions PowerShell would be little more than a bad way to start programs. You can think of expressions kind of like arithmetic or algebra formulas. They are a series of values manipulated by operators. Expressions contain literals, variables and operators. Variables contain data; the data they contain can be simple like numbers and strings or complex object. Either way, the data they represent is manipulated by operators that are arranged in expressions.

# Requirements

PowerShell

# Task 1—Literals

Literals are numbers or strings that are typed directly into expressions. They do not change. If they are included in scripts, they will have the same value every time the script runs.

## Steps

1. Numeric literals. You will be working a lot with numbers. Sometimes there are exact numbers you want to work with. Let’s say we want to calculate the circumference of a circle. The circumference is calculated as pi times the radius. If we have a circle with the radius of 5 and want to calculate the circumference. You can do that by entering the expression in PowerShell. Enter:  
   **3.1415 \* 5**  
   When you press enter the expression is automatically calculated and the results are sent to the host shell and displayed.
   1. What was the result? 15.7075
   2. In this case you used two numeric literals and one operator. If you hit the up arrow and run the expression again you will get the same answer every time.
   3. Numeric literals can use common computer measurement units: KB, MB, GB, TB, and PB.
   4. Using PowerShell find the value of each of the units. To find the value of KB just enter:  
      1kb

|  |  |
| --- | --- |
| Unit | Value |
| 1kb | 1024 |
| 1mb | 1048576 |
| 1gb | 1073741824 |
| 1tb | 1099511627776 |
| 1pb | 1125899906842624 |



1. Expressions can be simple or complicated. Just like in algebra, parentheses are used to force the order of operations.
2. When you use literals, PowerShell will use the most efficient datatype without losing any precision.
3. String literals. PowerShell supports a few different types of string literals.
   1. Literal strings. Literal strings are created using single quotes. E.g. Enter:  
      Write-Host 'Hello World'
      1. Whatever you put in single quoted strings is stored exactly as typed.
   2. Expansion strings: Expansion strings are created with double quotes. If you don’t put a variable or expression in the string it behaves just like a literal string. Enter:  
      Write-Host "Hello World"
      1. Now add a cmdlet into the expansion string, you can put any expression inside of $() to have the results of the expression inserted into the string at that point of the string. Enter:  
         **Write-Host "Hello World, today is $(Get-Date)"**
         1. What was the result? Hello World, today is 04/19/2021 10:42:44
      2. The “escape” character in PowerShell is the ` (backtick). The following escape sequences can be used in expansion strings to insert special characters:  
         `0 Null  
         `a Alert  
         `b Backspace  
         `f Form feed  
         `n New line  
         `r Carriage return  
         `t Horizontal tab  
         `v Vertical tab  
         `` Backtick  
         `" Double Quote (don’t end string)
      3. Try a couple. Enter the following command all on one line.  
         "`tHere is a`r`nmini paragraph. The `"```" is PowerShell's `"escape`" character."
         1. What is the result? Here is a  
            mini paragraph. The "`" is PowerShell's "escape" character.
   3. here-strings. A **here-string** is a string that preserves whitespace and returns. This makes it easier to write formatted strings. Use the **@** to start and end a **here-string**. Here-strings can be literal with a single quote @' … '@ or expansion with a double quote@" ... "@. Put the delimeters on a line by themselves”. Try this (use the tab key for indents):  
      @'  
      <html>  
       <head>  
       </head>  
       <body>I love PowerShell</body>  
      </html>  
      '@
      1. You could do the same thing with an expansion string and special characters like this: Try it:  
         "<html>`n`t<head>`n`t</head>`n`t<body>I love PowerShell</body>`n</html>"
      2. I think you will agree that the **here-string** is easier to both read and write.

# Task 2—Expressions and Operators

Expressions and Operators are the workhorse of PowerShell. We will spend a significant amount of time going over them in detail throughout the semester. Operators in PowerShell work pretty much the same as in any language. PowerShell has a large list of operators

|  |  |
| --- | --- |
| Type | Operators |
| Arithmetic | +, -, \*, /, % |
| Assignment | =, +=, -=, \*=, /=, %= |
| Comparison | -eq, -ne, -gt, -lt, -le, -ge,  -match, -notmatch,  -like, -notlike,  -in, -notin,  -contains, -notcontains |
| Logical | -and, -or, -xor, -not, ! |
| Redirection | >, >>, 2>, 2>, and 2>&1 |
| Split and Join | -split and -join |
| Type | -is, -isnot, -as |
| Unary | +, - , ++ -- |
| Array subexpression | @( ) |
| Call | & |
| Cast | [ <type>] |
| Array | , |
| Dot Source | . |
| Index | [ ] |
| Pipeline | | |
| Property | <object>.<property|method> |
| Range | .. |
| Static member | :: |
| Subexpression | $() |
| Bitwise | -band, -bor, -bxor, -bnot |
| Order of operations | () |

For this task we will focus on the Comparison and logical operators since they use different symbols than most languages.

The basic comparison operators are:

|  |  |
| --- | --- |
| Operator | Meaning |
| -eq | Equals |
| -ne | not equal |
| -gt | Greater than |
| -lt | Less than |
| -ge | Greater than or equal to |
| -le | Less than or equal to |

The operators are used for both numbers and strings.

## Steps

1. Comparison operators.
   1. For each of the following expressions, Enter your guess then run the expresson in powershell to see if you were right.

|  |  |  |
| --- | --- | --- |
| Expression | Anticipated Result | Actual Result |
| 'red' -eq 'red' | true | true |
| 'Blue' -eq 'blue' | true | true |
| 4 -eq 4 | true | true |
| 4 -eq '4' | true | true |
| '4' -eq 4 | true | true |
| 2 -gt 10 | true | false |
| '2' -gt 10 | false | true |
| 2 -gt 10 | false | false |

* 1. What did you learn from the tests?  
     single quotes don’t have an effect on integers when doing comparisons.

1. Logical operators. Logical operators are used when you need to combine multiple Comparison operators to test for a desired condition. The logical operators always evaluate to true or false. Fill in the truth tables for the following operators. Use PowerShell to test your results. E.g. for the -and operator your tests would look like this   
   $true -and $true  
   $true -and -$false  
   $false -and $true  
   $false -and $false  
   The -not operator is an unary operator so it only has one operand.

|  |  |
| --- | --- |
| Operator |  |
| -and | |  |  |  | | --- | --- | --- | | -and | $true | $false | | $true | true | false | | $false | false | false | |
| -or | |  |  |  | | --- | --- | --- | | -or | $true | $false | | $true | true | true | | $false | true | false | |
| -xor | |  |  |  | | --- | --- | --- | | -xor | $true | $false | | $true | false | true | | $false | true | false | |
| -not or ! | |  |  |  | | --- | --- | --- | | -not | $true | $false | |  | false | false | |

# Task 3—Variables

Variables in PowerShell are strongly typed at the time of assignment. You do not have to declare variables to use them. A variable is created when you assign it a value. All variables start with a **$**, the name can use any character, but for now just stick to letters, numbers and the underscore. The data type stored in a variable can be changed at assignment time. You can use the gettype() method to determine what datatype a variable contains.

## Steps

1. Create a variable named Radius  
   $radius=4
2. Now calculate the circumference and store it in $circumference  
   $circumference=3.1415927 \* $radius
3. Write the circumference to the console in green  
   write-host $circumference -ForegroundColor Green
   1. What is the circumference? 12.5663708
4. Now write a better message using string concatenation. The + operator is used to concatenate string. Create a message and store it in the variable $message (this expression is all on one line)  
   $message='A circle with a radius of ' + $radius + ' has a circumference of ' + $circumference
5. Display the message in green. (Notice you don’t have to write the whole parameter name if you write enough for PowerShell to determine the parameter name.)  
   **Write-Host $message -f Green**
6. Do it again but using an expansion string. Enter the command (all on one line):  
   $message="A circle with a radius of $radius has a circumference of $circumference"
7. Write the message out again but this time in yellow:  
   Write-Host $message -f Yellow
8. What happens if you misspell a variable name? Enter:  
   "The radius is $radus"
   1. Record the result. The radius is
   2. When you access a variable that has not been created PowerShell does not throw an error. Instead it returns a $null.
9. Try this:  
   "The radius is $RADIUS"
   1. Record the result: The radius is 4
   2. What does that tell you about variable names? Not case sensitive.

# Task 4—Objects

PowerShell is an object-based language. All values have a datatype that describes the value. Remember: objects are instances of classes and classes define objects. Classes have properties that describe the thing and methods that act upon the thing.

## Steps

1. Determining an object’s data type. To determine an object’s data type you call the .GetType() method of the object. Since all types in PowerShell are objects they all have a GetType() method that will return the data type of the object. You can also get the type of a literal. We will work with types in greater detail later, this is just an intro to get us going. GetType() returns an object that describes an object’s type.
2. To determining a literal’s data type you just need to surround the literal with parenthesis and then call the GetType() method. Enter:  
   (42).GetType()
   1. What Name is returned? Int32 *That name is the object’s type.*
   2. What base type was this object inherited from? System.ValueType
3. Since GetType() returns an object you can access the properties of that object as well, The FullName property is of particular interest. It is the full .net name specification for the class. You can use that name to get help for the object. If the object is part of a Microsoft library you can get detailed help from docs.microsoft.com, usually you can just search for the fullname on Bing or Google and it will be one of the first entries. Enter:  
   (42).gettype().FullName
   1. What is the FullName? System.Int32
4. Determining the types of the variables we just created. To do that call the **GetType()** method of the variable. E.g. **$radius.gettype()**

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | Name | Base Type | FullName |
| $radius | Int32 | System.ValueType | System.Int32 |
| $circumference | Double | System.ValueType | System.Double |
| $message | String | System.Object | System.String |

1. Find the help on the fullname type for the $message variable. Just do an internet search for the fullname.
   1. From the documentation what does the PadLeft() method do? Returns a new string of a specified length in which the beginning of the current string is padded with spaces or with a specified Unicode character.
2. Accessing properties. To access an object’s properties you just use .propertyname. Access the length method of the object in the $message variable. Enter:  
   $message.Length
   1. What is the length of the message? 49
3. Accessing methods. To access an object’s methods you use .methodname(), if the method takes arguments. They go comma separated in the parenthesis. Try a few
   1. The ToUpper() method returns a new string that is all upper case. Enter  
      $message.toupper()
      1. Record your results? A CIRCLE WITH A RADIUS OF 4 HAS A CIRUMFRENCE OF
   2. The Subtring method has a couple of overloads. One accepts one argument: Substring(StartPosition), try it:  
      $message.Substring(5)
      1. Record your results: cle with a radius of 4 has a cirumfrence of
   3. The other accepts two arguments: Substring(StartPosition,NumberOfCharacters). Try it:  
      $message.Substring(5,12)
      1. Record your result: cle with a r
4. Exploring cmdlet results. Enter the following commands:  
   $p=Get-Process  
   $p.GetType()
   1. What is the name of the type that was returned? System.array
   2. The **[]** after the name indicates that the type is an array. Just for fun, use the count property of the $p variable to find out how many items are in the array.  
      $p.count
      1. How many items are in your array? 247
   3. Examining a specific item in an array. We will cover arrays in detail later, for now you need to know that the first element in an array is at index 0. Examine the first item in the array:  
      $p[0]
      1. What is the process name? AcrobatNotificationClient
      2. What is type of the first element? System.ComponentModel.Component

# Deliverable

Upload this document with completed answers to i-learn.