Infrastructure as Code-Walkthrough

PowerShell 7.x Variables and Loops

Contents

Last Updated: 24 November 2022

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to give you some more progress with learning PowerShell. You have already installed and basically used PowerShell, we will proceed with the normal programming variables, assignment, and loops.

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. These notes are abbreviated, you should already understand some basic programming terminology. You have already covered an introduction to Python.
2. Ensure you have read the accompanying lecture notes before you begin. You should have already installed PowerShell 7.
3. I will carry out my exercises using a Windows 2019 VM Standard, Desktop Experience, running in Hyper-V.
4. You should have already installed Visual Studio Code as per my notes.
5. You have an existing GitHub account and a basic ability to use it.
6. You have created a directory to keep example files in and you are ready to code.
    a. On my VM, I am using **C:\Powershell**
    b. At the end of each session, I copy the files to **OneDrive\Powershell**

Variables

This section is based on [1].

The default value of any variable is **$null**. Type **Get-Variable** to see what is currently defined by default.

The assignment operator is **=** and I can create a variable and print it as shown below.

```
$Rubbish = 1, 2, "a", "££"
$Rubbish
clear-variable -Name Rubbish
$Rubbish
Remove-Variable -Name Rubbish
```

I can also clear the variable after use or set it = **$null** or I could entirely delete the variable.

You can store any type of object in a variable, such as arrays, integers, and strings. But remember, in PowerShell, processes, services, etc are all also objects.

```
$Rubbish = 1, 2, "a", "££"
$Rubbish.GetType()
```

Usefully, we can cast a variable, so it has a fixed type.

```
[int]$Rubbish = 1
$Rubbish.GetType()
```

If I pass a string to variable, it will automatically convert it.

```
[int]$Rubbish = 1
$Rubbish = "123456789"
$Rubbish
```

However, if it's a string of letters, PowerShell doesn't do miracles!

```
[int]$Rubbish = 1
$Rubbish = "This will give you an error!"
$Rubbish
```

You can translate a date into a datetime object, but the format of the input string is assumed to be US, mmddyyyy. Do some reading and see if you can figure out how to get PowerShell to expect the input string in ddmmyyyy format as we would use in Ireland.

```
[datetime]$OGGI = "11/13/2022"
$OGGI
```

For conversions of variable types, review [2].

Working Example

Below I do a simple tax calculation.

```
$amount = 111
$VAT = 0.23
$result = $amount * $VAT
$result
$text = "Total €$result is the sum of €$amount with $VAT% VAT"
$text
```

I can store the output of a command for later use.

```
$dir_listing = Get-ChildItem c:\
$dir_listing
```

You can check if a variable exists using **test-path variable:\dir_listing**

When programmes get more complex, it can be tough keeping track of variables. I love this feature!

```
New-Variable JORzVariable -value 3.142 -description "PI with write-protection" -option ReadOnly
Get-Variable JORzVariable
```

Notice that this was a constant, so I also write-protected it.

Types

For this section, review [3].

Characters and strings have the usual functions available, review the options at the reference above.

```
$StringValue = "Yoo hoo!"
$StringValue.ToUpper()
$StringValue.ToLower()
```

You should be familiar with the concept of an array; we covered this in the Python section of the course.

```
$MyArray = 1,2,3,4,5
$MyArray[1]
```

Rather than a single Integer type, we have int for 32-bit numbers and long for 64-bit numbers. These are signed, so an int can store +/- 2^{31} values, the first bit demotes positive or negative. A long can store +/- 2^{63} values. There is also a value for byte.

```
$LittleNumber = 12345
$LittleNumber.GetType()
$BigNumber = 123456789123456789
$BigNumber.GetType()
```

In floating point, we have 32- and 64-bit options again, single/float and double precision numbers.

```
[float]$Floaty32 = 12.12
$Floaty32.GetType()
[double]$Floaty64 = 12345.1234
$Floaty64.GetType()
```

Interestingly, there is also a 128-bit decimal type.

Review the reference provided, in particular the math types, although we do not need trigonometry in this module, we use it everywhere in science and engineering.

The usual orders of precedence apply, but I use brackets to keep everything clear for myself.

Tests

As with most language, we have the conditions. There are Equality Operators

- -gt greater than
- -igt greater than, case-insensitive
- -cgt greater than, case-sensitive
- -ge greater than or equal
- -ige greater than or equal, case-insensitive
- -cge greater than or equal, case-sensitive
- -lt less than
- -ilt less than, case-insensitive
- -clt less than, case-sensitive
- -le less than or equal
- -ile less than or equal, case-insensitive
- -cle less than or equal, case-sensitive

If

You can play with the various options to test them.

```
$Variable1 = 12
$Variable2 = 32
if ( $Variable1 -ne $Variable2  )
{
   Write-Output "The condition was true"
}
```

We can also use elseif.

```
$day = 3

if ( $day -eq 0 ) { $result = 'Sunday'       }
elseif ( $day -eq 1 ) { $result = 'Monday'    }
elseif ( $day -eq 2 ) { $result = 'Tuesday'   }
elseif ( $day -eq 3 ) { $result = 'Wednesday' }
elseif ( $day -eq 4 ) { $result = 'Thursday'  }
elseif ( $day -eq 5 ) { $result = 'Friday'    }
elseif ( $day -eq 6 ) { $result = 'Saturday'  }

$result
```

Switch

Any C programmers will start being at home now! In the previous session we used if and elseif to select a day, here I use switch [4].

```
$day = 4

switch ( $day )
{
   0 { $result = 'Sunday'    }
   1 { $result = 'Monday'    }
   2 { $result = 'Tuesday'   }
   3 { $result = 'Wednesday' }
   4 { $result = 'Thursday'  }
   5 { $result = 'Friday'    }
   6 { $result = 'Saturday'  }
}

$result
```

Loops

And we have the usual loops!

I presume you understand break (which exists the loop) and continue (which goes to the next cycle of the loop) and I will not explain them here.

For Loop

The for loop has the syntax:

```
for (<Init>; <Condition>; <Repeat>)
{
   <Statement list>
}
```

Init: a command before the loop begins, for example **$counter=0**
Condition: resolves to true or false and determines whether the loop runs.
Repeat: executes every time the loop runs.

```
for ($counter = 0; $counter -lt 10; $counter++)
{
   $counter
}
```

ForEach

We can iterate through an array using foreach.

```
$MyArray = "J", "o", "h", "n"
foreach ($Letter in $MyArray)
{
   $Letter
}
```

While

While loops can have a simple format.

```
while($val -ne 3)
{
   $val++
   Write-Host $val
}
```

Some while loops are a little more complicated. This is a simple menu example from 4sysops.com. It takes user input and carries out actions depending on the user input.

```
while(($inp = Read-Host -Prompt "Select a command") -ne "Q"){
   switch($inp){
      L {"File will be deleted"}
      A {"File will be displayed"}
      R {"File will be write protected"}
      Q {"End"}
      default {"Invalid entry"}
      }
   }
```

Do/Until

A do until loop is very similar to while.

```
$a = 0
do
{
   “Starting Loop $a”
   $a
   $a++
   “Now `$a is $a”
} until ($a -ge 5)
```

Piping Commands

In Linux and in PowerShell, one of the most powerful things we can do is to pipe the output of one command to be the input of another. The pipe symbol is **|**

```
Dir | Format-Table | Out-Host
```