Getting Started With Microsoft PowerShell

James E. Jarvis

November 24, 2016

Contents

1 About PowerShell

PowerShell is a registered trademark of Microsoft Corporation. PowerShell is provided by Microsoft Corporation and is a well thought out, very usable and powerful scripting language. The influence of other scripting languages, not just Microsoft can be seen

In PowerShell commands (often called *cmdlets*) have the form:

verb-noun

that is a *verb* indicating the action and a noun indicating the *object* for example:

get-childitem

PowerShell itself is provided as a command line environment. In addition to all the PowerShell "cmdlets" one may call a non-PowerShell program directly (e.g. notepad.exe).

2 Getting started

This document assumes you are working on a University of Edinburgh Supported Windows Desktop. The exercises may work in other environments but this has not been tested. If following this document you may need to search the Internet for additional modules and follow the instructions on how to install them. Unless you know what you are doing, it is sensible to only install extra PowerShell functionality from genuine Microsoft sites.

Readers will find the exercises easier if they have previous experience of scripting or programming languages. Additionally, PowerShell uses pipelines extensively so prior attendance at Unix 1 and Unix 2 courses will be and advantage.

1. Click on the Start button
2. Type *PowerShell*
3. You should now see:
    - Windows PowerShell (Figure 1)
    - Windows PowerShell ISE (Figure 2)

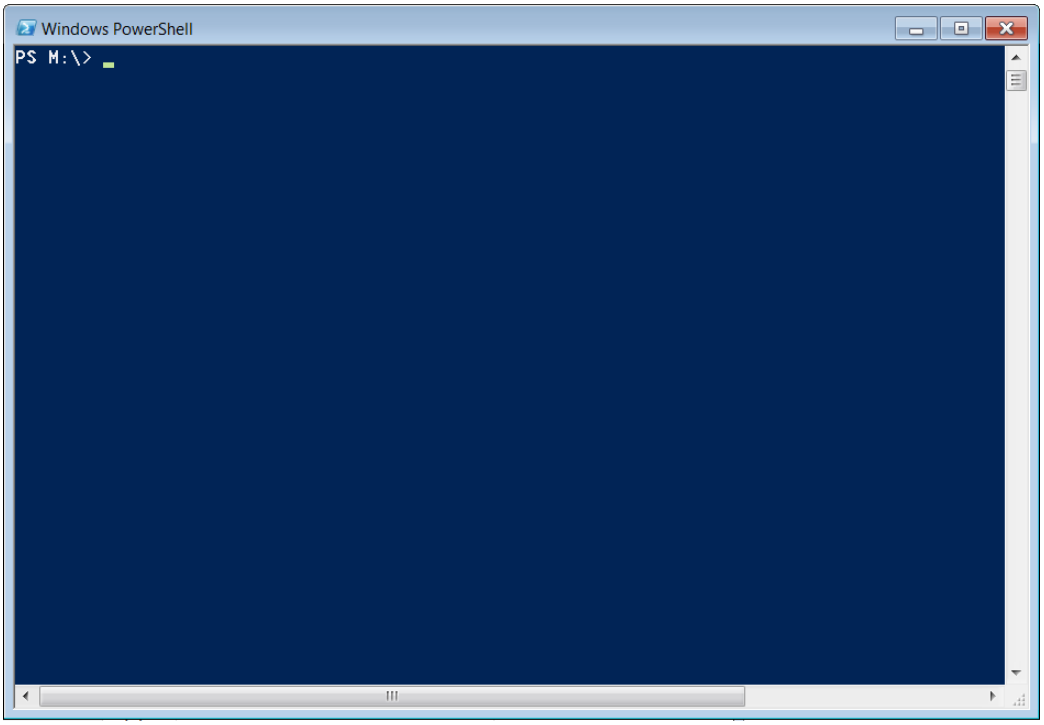

Figure 1: PowerShell - the command line environment.

If running scripts, the first option of using the PowerShell directly is fine. For creating and editing scripts the PowerShell ISE (Integrated Scripting Environment) is very useful.

You may see additional options but these are the two to use for this course.

2.1 Exercise

1. Run PowerShell. Type **exit** and press the **Enter** key. What happens?

2. Run PowerShell ISE. Click on the **Help** menu. Click on **Update Windows PowerShell Help**. You will notice in the window below a command **update-help** runs and then usually produces an error. This is normal on the University of Edinburgh Supported Windows Desktop.

3. Read the red error text. Can you determine why the command failed?

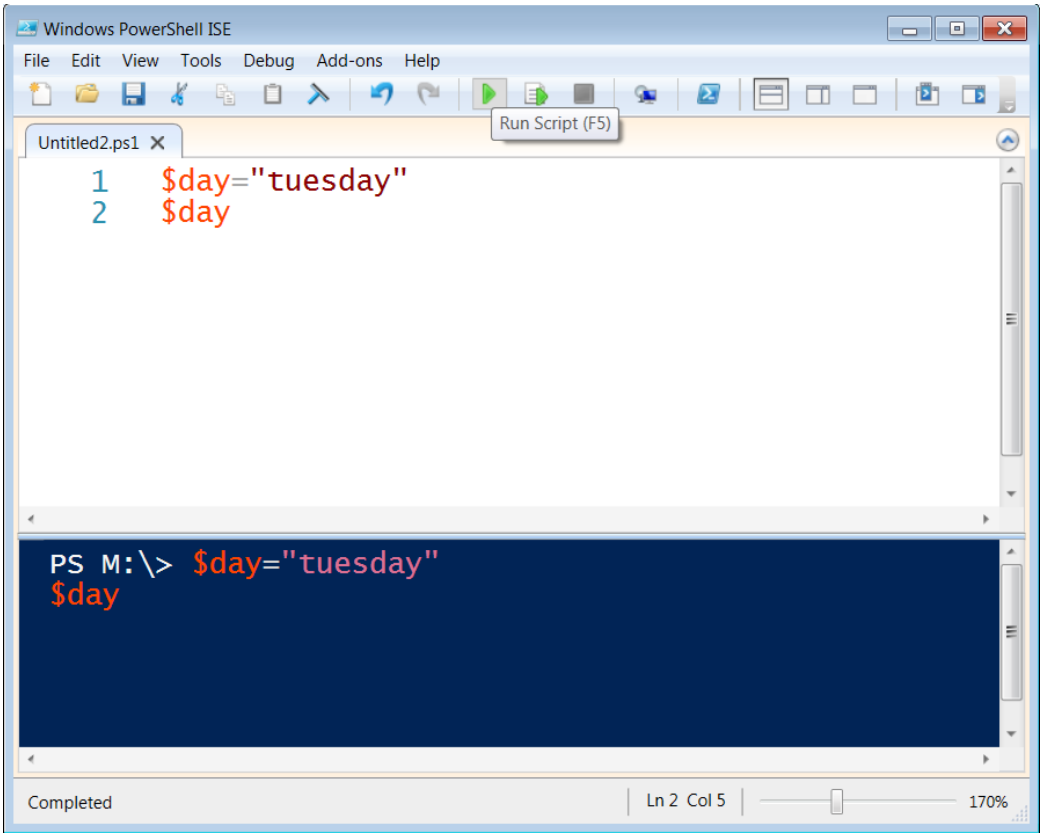

Figure 2: PowerShell ISE - An integrated script editor.

3 Commands

Commands may take none, one or several parameters and one, none or several values (see Listing **??**, page **??**). Users of Unix shell environments will quickly appreciate that the developers of PowerShell have implemented command syntax in a similar way.

The command **write-host** can be used with no parameters, in which case it will create a blank line of output (see Listing 2, page 6).

We can find options for **write-host** by typing **help write-host** (see

Listing 1: Command Syntax

```
<command-name> -<Required Parameter Name> <Required Parameter Value>
                [-<Optional Parameter Name> <Optional Parameter Value>]
                [-<Optional Switch Parameters>]
                [-<Optional Parameter Name>] <Required Parameter Value>
```

Listing 2: Hello World

```
PS M:\> write-host

PS M:\> write-host "Hello World"
Hello World
```

Command	Aliases
clear-host	cls, clear
format-list	fl
get-childitem	gci, ls, dir
get-content	gc, cat, type
get-location	gl, pwd
get-member	gm
remove-item	ri, rm, rmdir, del, erase, rd
write-output	write, echo

Table 1: Some PowerShell Command Aliases

Listing 3, page 7). For extra detail, add the "-full" option: **help -full write-host**

3.1 Exercise

1. Type **write-host "Hello World"** and press the **Enter key**
2. Type **write-host -foregroundcolor yellow "Hello World"**
3. Get PowerShell to print in blue text on a yellow background? Clue, use the **-backgroundcolor** parameter.
4. Type **help clear-host -online** What happens?

4 Aliases

Many commands have aliases (see Table 1) and for those who have used DOS or Unix these can be very familiar. Aliases are short forms of a command. So for someone used to using the command *pwd* typing **help pwd** will indicate that the underlying command is actually **get-location** . However, if coming from a Unix or DOS environment, typing the form you are familiar with makes adopting powershell easier.

Listing 3: Getting help for the write-host command

```
PS M:\> help write-host

NAME
    Write-Host

SYNOPSIS
    Writes customized output to a host.


SYNTAX
    Write-Host [[-Object] <Object>] [-BackgroundColor
    <ConsoleColor>] [-ForegroundColor <ConsoleColor>]
    [-NoNewline] [-Separator <Object>] [<CommonParameters>]


DESCRIPTION
    The Write-Host cmdlet customizes output. You can
    specify the color of text by using the ForegroundColor
    parameter, and you can specify the background color by
    using the BackgroundColor parameter. The Separator
    parameter lets you specify a string to use to separate
    displayed objects. The particular result depends on
    the program that is hosting Windows PowerShell.


RELATED LINKS
    Online Version:
    http://go.microsoft.com/fwlink/?LinkID=113426
    Clear-Host
    Out-Host
    Write-Debug
    Write-Error
    Write-Output
    Write-Progress
    Write-Verbose
    Write-Warning

REMARKS
    To see the examples, type: "get-help␣Write-Host
␣␣␣␣-examples".
    For more information, type: "get-help␣Write-Host
␣␣␣␣-detailed".
    For technical information, type: "get-help␣Write-Host
␣␣␣␣-full".
    For online help, type: "get-help␣Write-Host␣-online"
```

5 Variables in PowerShell

Variables are labels we use to store data that can vary (hence the name "variable").

In PowerShell variables are referenced and dereferenced by appending the variable name with $ symbol.

In PowerShell variables can be of many types:

- integers (whole numbers, positive or negative, e.g. 1, 5 ,-17, 0)
- doubles (numbers with decimal places, positive or negative, e,g. 1.5, 5.0, -17.2, 0.45)
- strings (sequences of characters that make, for example, words or sentences)
- arrays (sequences of variables referenced by an integer index, for example strings can be treated as arrays)
- hash tables (sometimes called dictionary, these are key value pairs)
- objects (an object is may contain a complex set of variable types and associations) - some examples include:
  - processes
  - services
  - event logs
  - computers
  - XML
  - anything you can think might need a variable!

5.1 Integer Variables

Variables can handle numbers and perform arithmetic operations. There are several types of variable that can contain numbers. The simplest variable type is the **[int]** type for integers (see Listing 4, page 9).

PowerShell trys to *do the right thing*. The variable $a in Listing 5 (page 9) is an integer. We the redeclare $a and it becomes a string. The answer provided if we now add the two variables will depend on the type of the first variable listed.

Listing 4: Integer variables and arithmetic

```
PS M:\> [int] $a=5
PS M:\> [int] $b=7
PS M:\> $a
5
PS M:\> $b
7
PS M:\> $a+$b
12
```

Listing 5: Integers and strings

```
PS M:\> [int] $b=7
PS M:\> $a=4
PS M:\> $a.getType().Name
Int32
PS M:\> $a+$b
11
PS M:\> $a="4"
PS M:\> $a.getType().Name
String
PS M:\> $a+$b
47
PS M:\> $b+$a
11
```

5.2 Doubles

Variables of type **[double]** are similar to integers in that they hold numbers however the number can contain fractional parts as decimals (see Listing 6, page 9).

5.3 String Variables

To assign the value "Tuesday" to variable called "day" we type **$day="Tuesday"** as see in Listing 7 (page 10).

We can also ask the user to enter information using the **read-host** command (Listing 8, page 10).

The **read-host** command echoes back what was typed however if we assign that value to a variable, say *$name* then we can capture the user

Listing 6: Arithmetic with numbers with decimal points

```
PS M:\> [double] $a=4.0
PS M:\> [double] $b=3.5
PS M:\> $a-$b
0.5
```

Listing 7: String variables

```
PS M:\> $day="Tuesday"
PS M:\>
PS M:\> $day
Tuesday
PS M:\>
```

Listing 8: Reading user input

```
PS M:\> read-host "What␣is␣your␣name:"
What is your name:: James
James
```

input for use later (see Listing 9, page 10).

String variables can be more than one line long, bounded by the double quotes as in (see Listing 10, page 11).

Things get more interesting when we explicity set $a to be of type *string* inf Listing 11 (page 11). Here, when $a and $b are added they are added as strings. If we change the order and add $b and $a the result is an integer calculation. PowerShell casts the result based on the first variable type.

Generally PowerShell will have a good idea of what you mean when you assign a value to a new variable. It will equally make sensible conversions where it can but type needs care. Listing 12 (page 11 illustrates the variable $myvar starting as an integer, becoming a double and then becoming a string.

Note that the *BaseType* for numeric is *System.ValueType* whereas for strings it is *System.Object*. Generally PowerShell handles mixtures of *System.ValueType* sensibly. To force a variable to be a particular type you specify the type in square brackets (see Listing 13 on page 12).

5.4 Special variables

There are several special variables:

- $true
- $false

Listing 9: Storing a user's response

```
PS M:\> $name=read-host "What␣is␣your␣name:"
What is your name:: Jane
PS M:\> write-host $name
Jane
```

Listing 10: Multiline string variables

```
PS M:\> $workaddress="Main Library,
>> George Square,
>> Edinburgh"
>>
PS M:\> $workaddress
Main Library,
George Square,
Edinburgh
PS M:\> $workaddress
Main Library,
George Square,
Edinburgh
PS M:\>
```

Listing 11: Results can be unexpected if variable types are mixed

```
PS M:\> [string] $a="4"
PS M:\> [int] $b=7
PS M:\> $a+$b
47
PS M:\> $b+$a
11
PS M:\>
```

Listing 12: Determining the type of a variable or object

```
PS M:\> $myvar=5
PS M:\> $myvar.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType


PS M:\> $myvar=$myvar+0.4
PS M:\> $myvar
5.4
PS M:\> $myvar.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Double                                   System.ValueType


PS M:\> $myvar="7"+$myvar
PS M:\> $myvar
75.4
PS M:\> $myvar.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object


PS M:\>
```

Listing 13: Forcing (declaring) type of variable or object)

```
PS M:\> $a.GetType()

IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     True     Int32                    System.ValueType


PS M:\> [double] $q=$a
PS M:\> $q.GetType()

IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     True     Double                   System.ValueType
```

Listing 14: A string can be viewed as an array of characters

```
PS M:\> $i="cheese"
PS M:\> $i[4]
s
PS M:\> $i[0]
c
PS M:\> $i.Length
6
```

- $null

If a command succeeds it will return $true but on failure it will return $false. The variable $null indicates a variable is not set. Setting a variable's value to $null effectively deletes that variable.

5.5 Arrays

Arrays are variables with multiple values with the first value indexed as 0, the second as 1 and so on. We have already used string variables and these can be broken into an array of characters as in Listing 14 (page 12).

Note how we use the **.Length** method to access how long is the string.

We can also have an array of strings as words - Listing 15 (page 13) creates an array to hold the names of Scottish cities:

Note that the length is actually the number of items in the array. Array values are indexed from *0 (zero)* - in the above example *$city[0]* would have the value *Edinburgh*. That is why the fourth element containing "Aberdeen" is dereferenced [1] as **$** city[3]

[1] Dereferencing a variable

Listing 15: An array containing the names of Scottish cities

```
PS M:\> $city=("Edinburgh","Glasgow","Dundee","Aberdeen")
PS M:\> $city[3]
Aberdeen
PS M:\> ($city).Length
4
```

Listing 16: A directory listing is an array of file system objects

```
PS M:\> $myarray=get-childitem
PS M:\> ($myarray).Count
475
PS M:\> $myarray[99]


    Directory: M:\


Mode      LastWriteTime  Length Name
----      -------------  ------ ----
-a---   11/04/2014 17:09 95579 Advanced-PowerShell-To-Office365.docx


PS M:\>
```

5.6 Arrays of Objects

In the Listing 16 (page 13) we use **get-childitem** command to get the file and folder contents of the current directory but assign the result to **$myarray** .

Your directory listing may have a different number of items specified by the **(** $myarray).Count variable.

5.7 Hashes

Hashes or dictionary objects are very useful. They are essentially errays of key-value pairs.

For this example we shall use Scottish cities and their populations.

Listing 17 (page 14) illustrates the use of a hash (see Section 5.7 to store a map of city to population.

In Listing 18 (page 14) we add Falkirk (which is not a city but does have an impressive wheel and some nearby Kelpies).

Essentially what hashes do is allow us to access variable data with a variable and that variable need not be an integer. Note the order hashes are returned is not intended to be predictable. In Listing 19 (page 15) we can access the population by explicit use of the string or a variable containing the

City	Population
Glasgow	976970
Edinburgh	488610
Aberdeen	209460
Dundee	157690

Table 2: Populations of Scottish Cities, Mid-2012 Populations Estimates for Settlements and Localities in Scotland from General Register Office of Scotland website

Listing 17: Using a hash to store Scottish cities and their populations

```
PS M:\> $pops=@{"Glasgow"=976970;"Edinburgh"=488610;
>> "Aberdeen"=209460;"Dundee"=157690}
>>
PS M:\> $pops

Name                           Value
----                           -----
Edinburgh                      488610
Glasgow                        976970
Dundee                         157690
Aberdeen                       209460

PS M:\>
```

Listing 18: Adding to a hash table

```
PS M:\> $pops.Add("Falkirk",100480)
PS M:\> $pops

Name                           Value
----                           -----
Edinburgh                      488610
Glasgow                        976970
Falkirk                        100480
Dundee                         157690
Aberdeen                       209460


PS M:\>
```

Listing 19: Accessing a hash value using its key

```
PS M:\> $pops."Aberdeen"
209460
PS M:\> $city="Aberdeen"
PS M:\> $pops.$city
209460
PS M:\>
```

Listing 20: Accessing a hash value using input from a user

```
PS M:\> $pops=@{"Glasgow"=976970;"Edinburgh"=488610;
>> "Aberdeen"=209460;"Dundee"=157690}
>>
PS M:\> $city=read-host "Enter␣a␣Scottish␣city:"
Enter a Scottish city to be given its population: : Edinburgh
PS M:\> $pops.$city
488610
PS M:\>
```

string. If you have used hashes before you might agree this is rather elegant.

Not convinced? Well it is really useful for lookups based on user input (see Listing **??**, page 15).

5.8 Removing variables

The quickest way to remove a variable is to set it to be null as in Listing 21 (page 15).

5.9 Exercise

1. Create a variable **$a** and assign it a value of 3 then use the **write-host** command to display the value.

2. Type **$a.GetType()** to find the type of variable *$a*

3. Create a variable **$b** and assign it a value of 3.3 then use the **write-host** command to display the value.

Listing 21: Deleting a variable or object

```
PS M:\> $myvar="Something"
PS M:\> $myvar
Something
PS M:\> $myvar=$null
PS M:\> $myvar
PS M:\>
```

Listing 22: The PowerShell environment

```
PS C:\> get-item env:\

Name                           Value
----                           -----
SystemDrive                    C:
ProgramFiles(x86)              C:\Program Files (x86)
USERDNSDOMAIN                  ED.AC.UK
ProgramW6432                   C:\Program Files
.
.
.
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
HOMEDRIVE                      M:
windir                         C:\Windows
NUMBER_OF_PROCESSORS           4
OS                             Windows_NT
ProgramFiles                   C:\Program Files
ComSpec                        C:\Windows\system32\cmd.exe
.
.
.
USERDOMAIN_ROAMINGPROFILE      ED
PUBLIC                         C:\Users\Public
```

4. Type **$b.GetType()** to find the type of variable *$b*
5. Create a variable **$c** and assign it a value of "3.3" (include those quotes) then use the **write-host** command to display the value.
6. Type **$c.GetType()** to find the type of variable *$c*
7. Assign to a variable *$d* the sum of the first two variables by typing **$d=$a + $b**
8. What value is returned when you convert to integer the values 3.4 and 3.6? (This may be a surprise to those with Unix experience.)

6 The PowerShell environment

Within a shell the *environment* is the term to describe the current settings. These settings are exposed in *environment variables*. So for example the variable *username* contains the username of the current logged on user - presumably you! Listing 22 (page 16) shows the environment (with some values omitted).

Listing 23 (page 17) shows how to access and assign a single environment variable. Note the use of the parentheses around the command then

Listing 23: Accessing a single environment value

```
PS C:\> get-item env:\username

Name                           Value
----                           -----
USERNAME                       jjarvis
PS C:\> (get-item env:\username).value
jjarvis
PS C:\> $me=(get-item env:\username).value
PS C:\> $me
jjarvis
PS C:>  $env:username
jjarvis
```

Listing 24: Accessing a single environment value

```
cmdlet1 < c:\fileA.txt | cmdlet2 > c:\fileB.txt
```

prepended with *.value*. However a much easier syntax is **$env:varname** which does away with the parentheses.

7 Redirection and pipes

The redirection operators are the greater than and less than symbols.

The greater than symbol > indicates that output data is redirected to a file (or something behaving as a a file). The less than symbol < indicates that data is read in from a file.

Listing 23 (page 17) shows cmdlet1 will take input from c:
fileA.txt and the output of the command is passed to cmdlet2; cmdlet2 will send any output to c:
fileB.txt

Pipes and redirection are difficult concepts to master. If you have not used them before be assured it is normal to find them difficult to understand.

The summary would be:

- Pipes sit between cmdlets, passing the output of the lefthand cmdlet to the input of the right hand cmdlet.
- Redirection controls flow between a cmdlet on the left of the symbol and a file on the right.
- Redirection < indicates information is read from the file for the cmdlet
- Redirection > indicate information written to the file by the cmdlet

8 Reading and writing to files

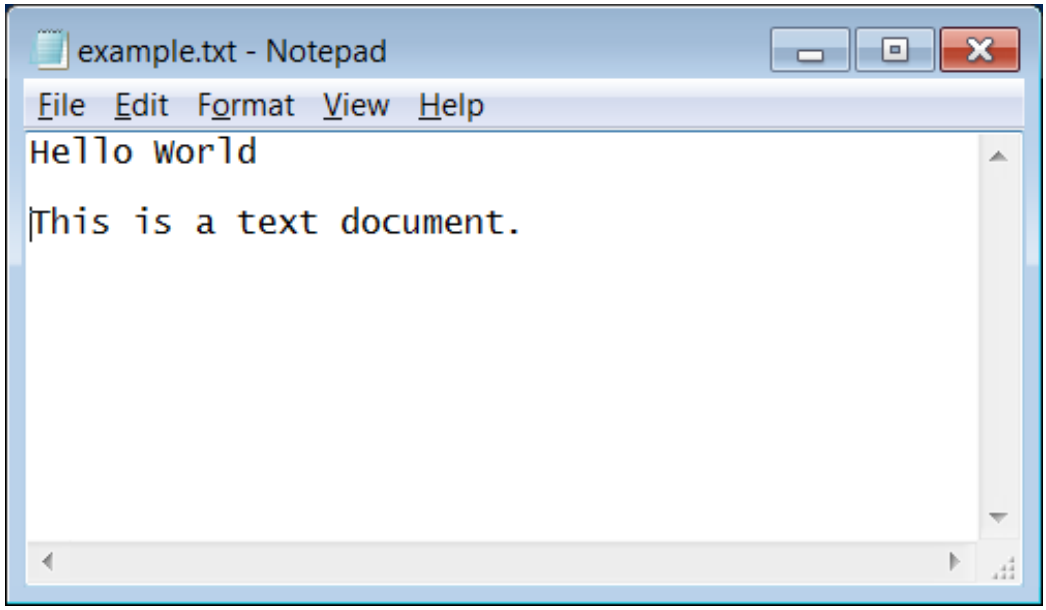

Figure 3: An example text file in Notepad editor.

The information we use on our computers is typically stored in *files*. PowerShell allows us to read and write to files. We will start with a plain text file as seen in Figure 3. This file called *example.txt* has the following text in it:

```
Hello World

This is a text document.
```

Assuming you have created the above (using notepad.exe) adn saved it to *m:\example.txt*, one can use the command **get-content** to inspect the contents. Listing 25 (page 19 shows how to get the content of the file put to screen followed by how to assign it to variable **$filecontent** for later use.

This time we will load the file and then write it our to a new file called *m:\hello.txt* (see Listing 26 on page 19). Before we we write the file we confirm *hello.txt* does not exist, which is the source of those red error messages.

Here we have introduced the *pipe* symbol - that vertical bar **|** to the left of the **out-file** command. The pipe symbol indicates the output of the command on its left is passed as the input to the command on the right. So above, instead of the value in *$filecontent* being sent to the screen (often

Listing 25: Accessing the content of a file

```
PS M:\> get-content M:\example.txt
Hello World

This is a text document.
PS M:\> $filecontent=get-content m:\example.txt
PS M:\> $filecontent
Hello World

This is a text document.
PS M:\>
```

Listing 26: Writing to a file

```
PS M:\> Get-Content .\example.txt
Hello World

This is a text document.
PS M:\> $filecontent=Get-Content .\example.txt
PS M:\> get-content hello.txt
‘        get-content : Cannot find path 'M:\hello.txt'
‘                                    because it does not exist.
‘At line:1 char:1
‘+ get-content hello.txt
‘+ ~~~~~~~~~~~~~~~~~~~~
‘+ CategoryInfo           : ObjectNotFound:
‘                   (M:\hello.txt:String) [Get-Content],
‘                         ItemNotFoundException
‘+ FullyQualifiedErrorId : PathNotFound,
‘    Microsoft.PowerShell.Commands.GetContentCommand

PS M:\> $filecontent
Hello World

This is a text document.
PS M:\> $filecontent | out-file hello.txt
PS M:\> get-content hello.txt
Hello World

This is a text document.
PS M:\>
```

Listing 27: Using get-member (gm) to discover what methods can be applied to variable

```
PS M:\> $filecontent | gm


   TypeName: System.String

Name              MemberType            Definition
----              ----------            ----------
Clone             Method                System.Object Clo...
CompareTo         Method                int CompareTo(Sys...
Contains          Method                bool Contains(str...
CopyTo            Method                void CopyTo(int s....
.
.
```

referred to as *standard output*) it instead is pass as input to the **out-file** command.

The pipe symbol is incredibly useful. It allows us to let information flow through multiple commands without having to save contents on the way. Now we will look at combining the pipe symbol with a really useful command...

Our variable *$filecontent* is of type "object" which means it is not necessarily obvious what you can use it for. This is where the pipe symbol and the **get-member** comes in handy. The **get-member** is so useful that you will find yourself command using the command very frequently so if you prefer less typing you may be pleased to know the alias is *gm*. Here is **gm** in action:

The output has been curtailed here but typically **gm** will show you what you can do or use from an object or variable. Remember earlier the **($city).Length** ? By using **$city | gm** it was clear that Length was a property of the variable *$city* that could be queried.

8.1 Exercise

1. Create and save a text file with notepad.exe containing your address. Call the file *address.txt*. Use **get-content** to display the file *address.txt* contents.

2. Assign the contents of *address.txt* to a variable **$address** and display the contents.

3. Create a multi-line string variable *$workaddress* with your work address

4. Create a variable *$myname* with your firstname and surname in it. Use the pipe symbol and **gm** to work out what methods are available to make it all uppercase or lowercase.

Listing 28: Putting commands into a script that greets with the current year

```
# This is a comment - commenting your scripts will make them
# more understandable for yourself and others.
# Comments begin with the hash symbol #

### Store today's year in a variable called "year"
$year=(get-date -UFormat "%Y")

### Ask the user for their name and store in variable "name"
$name=read-host "What␣is␣you␣name?"

### Write out a reply using the values name and day
write-host "Hello␣$name.␣␣This␣year␣is␣$year"
```

9 Scripts

So far we have been looking at PowerShell as a shell environment but now we will look at using PowerShell for writing scripts. By saving a series of commands to a file we can invoke the series again and again without having to retype. The file we save is a plain text file but with the file extension *.ps1* to indicate it may be invoked (or *executed*).

Listing 28 (page 21) details the sequence of commands which can be typed in PowerShell ISE. By saving the above listing in *m:\report-thisyear.ps1* and then the script may be run (or *executed*) by clicking on the run button in the PowerShell ISE (see Figure 4, page 22).

Alternatively open a PowerShell window, cd into the same folder as the saved file and type: **m** :\report-thisyear.ps1.

9.1 Exercise

1. Type in the previous listing in notepad.exe
2. Save as "report-thisyear.ps1" (you have to change the file extension).
3. Try running "report-thisyear.ps1" - you will need to specify the full path to the script.
4. Open "report-thisyear.ps1" in Notepad.exe again. Copy all the code to the clipboard.
5. Open the PowerShell ISE
6. Paste the contents of the clipboard into a new document in PowerShell ISE.

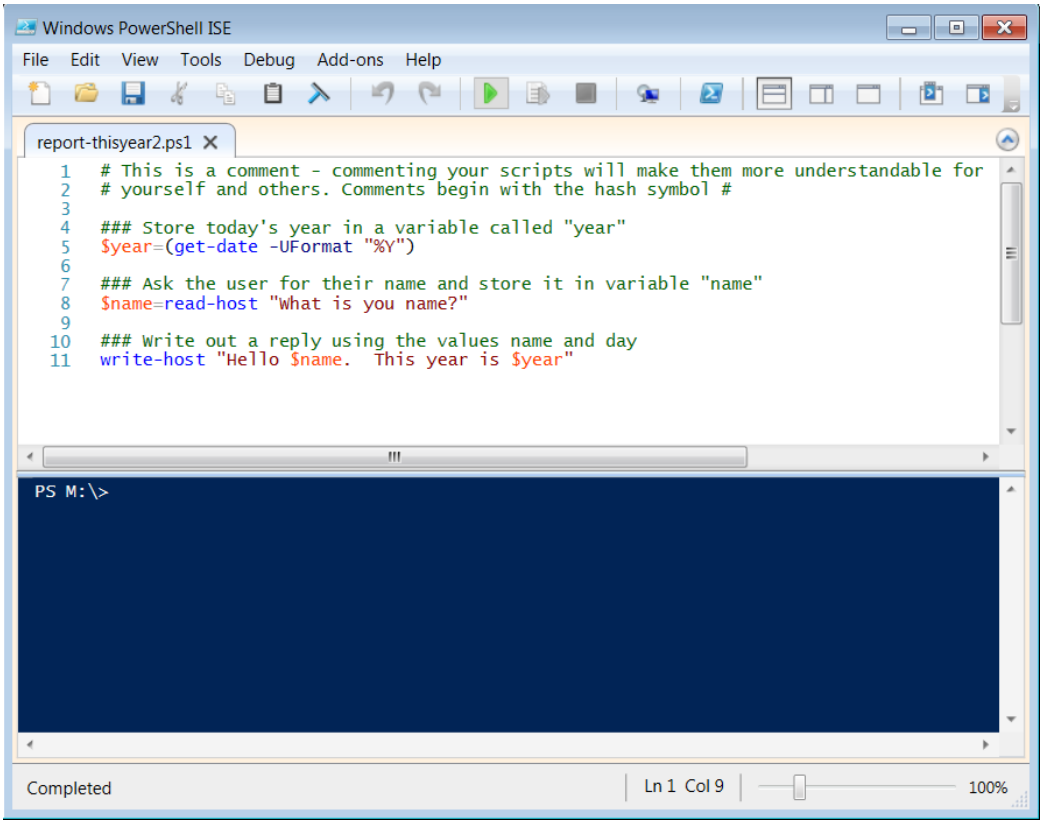

Figure 4: Running a script in the PowerShell ISE.

7. Save as "report-thisyear2.ps1".

8. Use the run button to run the script

Although powershell scripts may be written in Notepad.exe it should be apparent that using PowerShell ISE has several advantages.

10 Logic and loops for flow control

PowerShell like other coding environments allows for the asking of questions and looping a given number of times within scripts. The ability to do something dependent on something else, or to repeat a series of commands is the key to the usefulness of scripts. They allow us to save time doing mundane predictable processes.

Listing 29: Using the if else and elseif statements

```
[string] $name=read-host "What␣is␣your␣name?␣"
if ( $name.length -lt 5 ) {
  write-host "$name␣is␣a␣short␣name"
} elseif ( $name.length -lt 9 ) {
  write-host "$name␣is␣a␣medium␣length␣name"
} else {
  write-host "$name␣is␣long␣␣-␣are␣you␣tired␣typing␣it?"
}
```

Listing 30: Using the for loop

```
PS M:\svn> for ($a = 50; $a -ge 0; $a=$a-10) {
>> $a
>> }
>>
50
40
30
20
10
0
PS M:\svn>
```

10.1 If

Listing 29 (page 23) demonstrates flow control based on if, else and elseif statements. The computers asks for your name and then outputs a different message based on a computer value (length) of the name. If the script is run then *Jim* will get a different response from *Claire* and *Ebeneezer* will have a different response again.

10.2 For

The **for** command enables iteration over a range with a given step sequence. In Listing 30 (page 23) we have a countdown in steps of -10 from 50 to 0. More usually one is doing a count up in steps of one in which case **$a=$a+1** may be abbreviated to **$a++** .

10.3 Foreach

If you remember earlier we had a variable *$city* which had four values. Now four values is not many but it could be thousands. Listing 31 (page 24) totals the populations for the cities and calculates and average. The script does the following on each line:

Listing 31: Using the foreach to loop over an array of objects

```
PS M:\> $pops=@{"Glasgow"=976970;"Edinburgh"=488610;
>> "Aberdeen"=209460;"Dundee"=157690}
>>
PS M:\> [double] $total=0
PS M:\> foreach ($city in $pops.Keys) {
>>   $total=$total+$pops.$city
>> }
>> $average=$total / ($pops).Count
>> write-host "Total␣population:␣$total,␣average:␣$average"
>>
Total population: 1832730, average: 458182.5
```

Listing 32: Using a pipe and the measure-object to avoid the need for a loop

```
$pops=@{"Glasgow"=976970;"Edinburgh"=488610;
>> "Aberdeen"=209460;"Dundee"=157690}
>>
($pops.Values | Measure-Object -average ).Average
```

1. The hash variable **$pops** is initialised with the names and populations of four Scottish cities.

2. We initialise a the variable **$total** with type *double* and a zero value.

3. The **foreach** command indicates that for each value in turn in the **$pops.$keys** we should assign that value to **$city** and apply the code between the opening { brace and the closing } brace.

4. The one line inside the braces adds the population of the current value of the **$city** variable by accessing the hash **$pops.$city** .

5. The **foreach** loop restarts with the next value in **$pops.Keys** when it encounters the closing } brace for as long as there are still values.

6. The value for variable **$average** is calculated using the **$total** divided by the number of cities in **$pops.Count** .

7. The results are printed out.

Of course calculating an average is a fairly common requirement. In PowerShell it is usually possible to do things in more than one way. We could have simplified the task eliminating the need for a loop by instead using a pipe and the measure-object command as in Listing 32 (page 24).

Listing 33: How much time has passed since Nelson Mandela was born?

```
PS M:\> [datetime] $born="18 July 1918"
PS M:\> ((get-date).Year-$born.Year)
96
PS M:\> ((get-date)-$born)


Days              : 35202
Hours             : 10
Minutes           : 25
Seconds           : 37
Milliseconds      : 526
Ticks             : 30414903375268673
TotalDays         : 35202.4344621165
TotalHours        : 844858.427090796
TotalMinutes      : 50691505.6254478
TotalSeconds      : 3041490337.52687
TotalMilliseconds : 3041490337526.87

PS M:\>
```

10.4 Exercise

1. Open the PowerShell ISE
2. Write a script with a hash of people you know and their birthday.
3. Add a question to ask for a person.
4. Return the birthday for that person.

11 Advanced Topics

11.1 More date and time

Those who have scripted in other languages know that date and time can be *interesting* to handle! PowerShell does a good job of handling date and time. Date and time are held in variables of type

datetime

.

Listing 33 (page 25) is an example looking at Nelson Mandela's birth date and the amount of time passed since then.

Note that when two

datetime

Listing 34: Assigning a date

```
PS M:\> [datetime] $b="11:25␣pm␣7␣July␣1998"
PS M:\> $b

07 July 1998 23:25:00
```

values are subtracted the difference is of type

timespan

which is measured in *ticks*. One second lasts for 10000000 *ticks*.

Listing 34 (page 26) illustrates assigning a point in time to a **[datetime]** variable. PowerShell does a reasonable job of working out what is meant but checking input style is recommended as "pm" is accepted bu "p.m." is not!

11.2 Exercise

Using powershell calculate if you have lived for:

1. A million seconds?
2. A million minutes?
3. A million hours?
4. Calculate your age in units of millions of minutes.

11.3 Handling Data - Using CSV files

In Table 5.7 the population of Scottish cities was tabulated. Being able to access and manipulate tables of data is an essential part of the modern knowledge economy. PowerShell has some very useful features for accessing and referencing data read from files.

Being able to read and process this data can be very useful. There are ways to address Excel files using the COM Object Excel.Application however these examples are going to use CSV file formats and the PowerShell command **import-csv** .

Table 11.3 is a representation of the contents of the file "element.csv". Listing 35 (page 27) shows how we can access that data in powershell.

Element	Symbol	Protons	Neutrons
Iron	Fe	26	30
Oxygen	O	8	8
Hydrogen	H	1	0
Carbon	C	6	6

Table 3: Selected chemical elements and their properties

Listing 35: Importing a CSV file into a tabular data object

```
PS C:\Workspace> $elements=Import-Csv .\element.csv
PS C:\Workspace> $elements

Element                         Symbol
Protons                         Neutrons
-------                         ------
-------                         --------
Iron                            Fe
26                              30
Oxygen                          O
8                               8
Hydrogen                        H
1                               0
Carbon                          C
6                               6


PS C:\Workspace> $elements.Protons
26
8
1
6
PS C:\Workspace>  ($elements.Protons | Measure-Object -sum).sum
41
```

Listing 36: Accessing the University hierarchy XML

```
PS C:\Workspace> $webClient = new-object System.Net.WebClient
PS C:\Workspace> $webClient.Headers.Add("user-agent", "PowerShell Script")
PS C:\Workspace> $url="https://www.org.planning.ed.ac.uk/webware/units.xml"
PS C:\Workspace> $webClient.DownloadFile($url,"c:\workspace\units.xml")
PS C:\Workspace> [xml] $hierarchy=Get-Content C:\Workspace\units.xml
```

11.4 Exercise

1. Create a CSV file of elements with values shown in Table 11.3 (page 27).
2. Import the data into a variable $elements
3. Calculate the average number of protons (tip = Measure-Object -average)
4. Print element symbols were the number of protons equals the number of neutrons.

11.5 Advanced: Handling XML data and loading a .Net framework

Many data including metadata are now stored in eXtensible Markup Language (XML). PowerShell can handle objects which are XML by declaring them as **[XML]** .

Many data including metadata are now stored in eXtensible Markup Language (XML). PowerShell can handle objects which are XML by declaring them as

$$XML$$

.

The example in Listing 36 (page 28) loads the *current* University of Edinburgh Organisation Hierarchy. To do this we need to load the *System.NetcWebClient* object. If you have used VBScript before then loading a .Net framework will be familiar.

We now have an XML object $hierarchy. Unfortunately the XML file is really just a flat excel export of units with a field saying which level they reside and which is their parent. We can see this if we select the rows a level 2 of the University hierarchy as in Listing 36 (page 29). Note only the first and last level 2 units are shown for brevity.

Ideally the XML would be structured hierarchically. To transform the XML we need to load another .Net framework. We also need the transform which fortunately is available as a web download. Listing 38 (page 31) details

Listing 37: Accessing the University hierarchy level 2 units (Colleges)

```
PS C:\Workspace> $hierarchy.ROWSET.ROW | where {$_.UNIT_LEVEL -eq 2}
num                     : 2
UNIT_CODE               : HSS
UNIT_DESCRIPTION_LONG   : College of Humanities and Social Science
UNIT_DESCRIPTION_SHORT  : HSS
UNIT_LEVEL              : 2
UNIT_PARENT             : UOE
CATEGORY_AB_ACTIVITY    : A
URL                     : http://www.hss.ed.ac.uk/
UNIT_ADDRESS_LINE_1     : 57 George Square
UNIT_ADDRESS_LINE_2     : Edinburgh
UNIT_POSTCODE           : EH8 9JU
CURRENT_STATUS          : A
ACTIVE_FROM_DATE        : 2002-08-01T00:00:00
HEAD_OF_UNIT            : Professor XXXXXXXXXXX
HOU_EMAIL               : Head.CHSS@ed.ac.uk
HOU_ADDRESS_LINE_1      : 55 George Square
HOU_ADDRESS_LINE_2      : Edinburgh
HOU_POSTCODE            : EH8 9JU
CONTACT                 : XX XXXXXX XXXXXX
CONTACT_EMAIL           : XXXXX.XXXXXX@ed.ac.uk
CONTACT_ADDRESS_LINE_1  : 55 George Square
CONTACT_ADDRESS_LINE_2  : Edinburgh
CONTACT_POSTCODE        : EH8 9JU
LAST_UPDATED            : 2014-02-27T15:31:00
UPDATED_BY              : ORG
SORT_DESCRIPTION        : 1

.
.
.

num                     : 9
UNIT_CODE               : ACS
UNIT_DESCRIPTION_LONG   : Student and Academic Services
UNIT_DESCRIPTION_SHORT  : SASG
UNIT_LEVEL              : 2
UNIT_PARENT             : UOE
URL                     : http://www.ed.ac.uk/misc/acss.html
UNIT_ADDRESS_LINE_1     : Old College
UNIT_ADDRESS_LINE_2     : South Bridge
UNIT_ADDRESS_LINE_3     : Edinburgh
UNIT_POSTCODE           : EH8 9YL
CURRENT_STATUS          : I
ACTIVE_FROM_DATE        : 2002-08-01T00:00:00
INACTIVE_FROM_DATE      : 2006-02-06T00:00:00
HEAD_OF_UNIT            : XX XXXX XXXXXXX
HOU_EMAIL               : University.Secretary@ed.ac.uk
HOU_ADDRESS_LINE_1      : Old College
HOU_ADDRESS_LINE_2      : South Bridge
HOU_ADDRESS_LINE_3      : Edinburgh
HOU_POSTCODE            : EH8 9YL
LAST_UPDATED            : 2006-02-06T16:20:00
UPDATED_BY              : XXXXX
SORT_DESCRIPTION        : 4
```

the how we use another .Net framework to perform a transform on the XML. Note that the URL has been wrapped. The transformed XML object in $tree has an array of colleges under the top node university.

12 Further Help

PowerShell is easy to do quick experiments of how it will behave. Coupled with the *get-method* cmdlet (aliased to *gm*) one can quickly assess what methods are available for objects. Occasionally and object will be suitably opaque - but a web search will usually identify some existing use cases. It hardly needs saying that the best source for definitive answers is Microsoft but for examples the Microsoft Scripting Guys are particularly useful. Choose their examples first.

Other sources should be taken with a degree of caution. In particular do not run code that you have not "eye-balled" to verify you understand what it will do.

Finally, whilst there may be books out there the best source of help is reputable sites on the Internet providing one takes care to realise that technology is a rapidly changing discipline.

Listing 38: Transforming the XML

```
PS C:\> $webClient = new-object System.Net.WebClient
PS C:\> $webClient.Headers.Add("user-agent", "PowerShell Script")
PS C:\> $xslurl="http://edin.ac/1DydneY"
PS C:\> $xslfile="c:\workspace\uoe-create-hierarchical.xsl"
PS C:\> $webClient.DownloadFile($xslurl,$xslfile)
PS C:\> $xslt = New-Object System.Xml.Xsl.XslCompiledTransform
PS C:\> $xslt.Load($xslfile)
PS C:\> $xslt.Transform("C:\Workspace\units.xml","C:\Workspace\tree.xml")
PS C:\> [xml]$tree=Get-Content C:\Workspace\tree.xml
PS C:\> $tree

xml                                                          university
---                                                          ----------
version="1.0" encoding="utf-8"                               university


PS C:\Workspace> $tree.university


unit                  : UOE
UNIT_CODE             : UOE
UNIT_PARENT           :
UNIT_DESCRIPTION_LONG : University of Edinburgh
UNIT_LEVEL            : 1
LAST_UPDATED          : 2002-10-25T00:00:00
UPDATED_BY            : ORG
college               : {college, college, college, college...}



PS C:\Workspace> $tree.university.college[0]


unit                  : HSS
UNIT_CODE             : HSS
UNIT_PARENT           : UOE
UNIT_DESCRIPTION_LONG : College of Humanities and Social Science
UNIT_LEVEL            : 2
LAST_UPDATED          : 2014-02-27T15:31:00
UPDATED_BY            : ORG
node                  : node


PS C:\Workspace
```

13 Listings

Listings