Coding and Logic

Understand the features and applications of a range of coding and logic used in support roles.

Command Line Scripting

- Describe Scripting at the command line when supporting server administration
  - Unix Shell
  - Power Shell
  - Batch Scripting

Coding and Language

- Recognise the features and benefits of the following language types:
  - Low Level
    - Assembler
    - Machine Code
  - High Level
    - Procedural
    - Object Orientated
    - Event Driven

Application and Lifecycle management (ALM)

- Describe the functions of each stage of ALM
- Application Development Phases
  - Requirements
  - Design
  - Build
- Service Management Phases
  - Optimise
  - Operate
  - Deploy
- Application Management

Algorithms and Data Structures

- Recognise the practical Applications of:
- Algorithms
- Flow of Control
  - Branching
  - Looping
  - Iteration
- Data Structures
  - High Level
  - Floating Point
  - Strings
  - Integers

Webpage Development

- Recognise fundamental elements of website development
- Environment
  - Web Server
  - Database
  - Web Browser
- Development tools/options
  - Coding web pages with text files
  - Content Management Systems (CMS)
- Web page elements
  - CSS
  - HTML
  - XML
- Security
  - Secure Data Transit
  - Authentication and authorisation
  - Certificates

Scripting

- Windows Batch files
  - Extension of .bat
  - Set of commands for the command line
  - e.g. copying files
- Windows Script Files
  - Extension of .ps1
  - Set of commands for the power shell - far more powerful than batch
  - Server/PC management (copied idea from Linux Script Files)
  - e.g. dynamic back-ups or log all users out
- Linux Script Files
  - Extension of .sh
  - Has always been part of the OS
  - Server/PC management

Command Line Scripting

- PowerShell (Start→Windows Power Shell)
  - Will run command prompt commands
- Commands are called *cmdlets*
- Can call windows programs – notepad.exe

Power Shell Exercise 1

- Start Power Shell
- Type *write-host "Hello World"*
- Options can be found by typing *help write-host*
- Type *write-host -foregroundcolor yellow "Hello World"*
- Get Power shell to with "Hello World" with blue text on a yellow back ground.
- What does *help clear-host -online* do?

Power Shell Aliases

- Linux and Windows users get commands mixed up between the two platforms
- pwd used often in linux
- Type *help pwd* (it is an alias for get-location)

Power Shell Variables

- Variables are named memory locations that can be used to store (remember) data that can vary
- In power shell they are referenced using $
- Variables can be the following data types:
  - Integers (whole positive or negative numbers)
  - doubles (positive or negative numbers with decimal places)
  - strings (a list of characters)
  - arrays (list of other variables referenced by an integer index)
  - hash tables (key pair values)
  - objects (a complex set of variable types)

Power Shell Variables

Ignore the prompt and only type what follows the PS C:\>

```
PS C:\> $a=5
PS C:\> $b=6
PS C:\> $a
5
PS C:\> $b
6
PS C:\> $a+$b
11
PS C:\>
```

Power Shell Variables

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

Can you explain the last result?

Power Shell Variables

```
PS C:\> $day="Saturday"
PS C:\> $day
Saturday
PS C:\>
```

Power Shell User Input

- Use read-host (help read-host -online)

```
PS C:\> $a=2018
PS C:\> $year = read-host "What year were you born? "
What year were you born? : 1969
PS C:\> $age = $a-$year
PS C:\> write-host "Your age is " $age
Your age is  49
PS C:\>
```

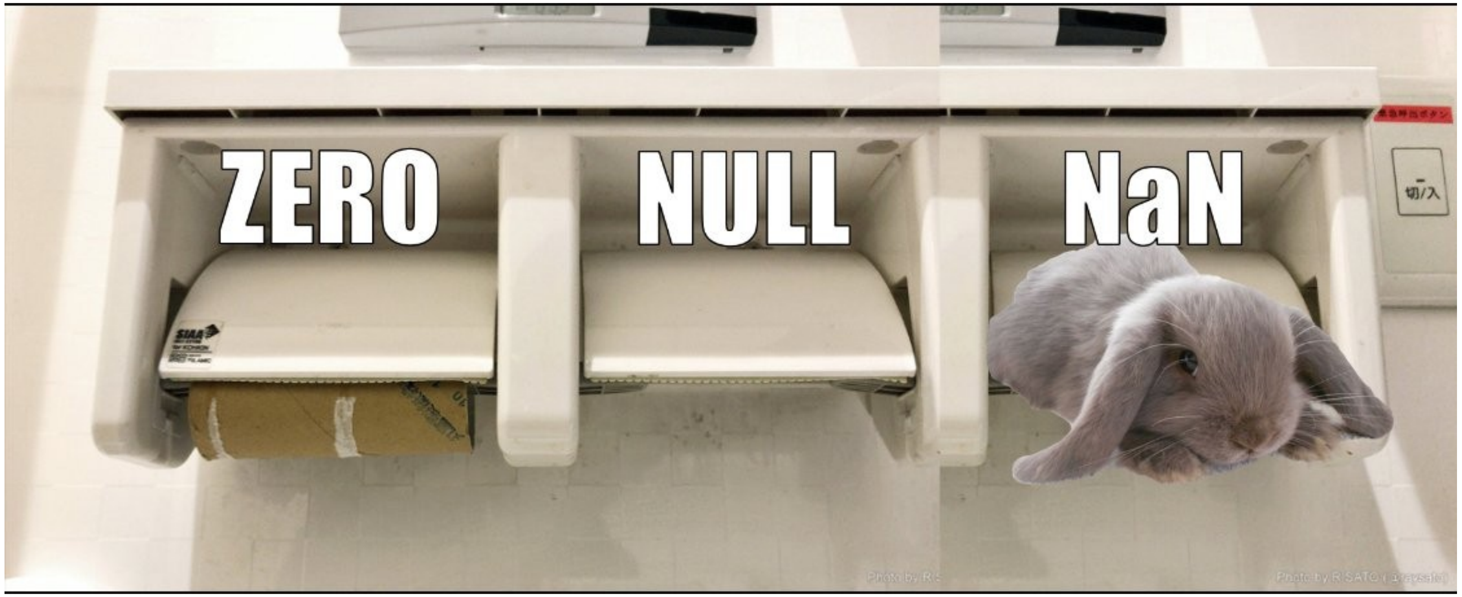
Power Shell Strings

Can be more than one line

```
PS C:\> $collegeAddress = "Sheepen Road,
>> Colchester,
>> Essex.
>> Co3 3LL"
PS C:\> $collegeAddress
Sheepen Road,
Colchester,
Essex.
Co3 3LL
```

Power Shell Special Variables

- $true (if a command succeeds it returns true)
- $false
- $null



Power Shell Arrays

- Arrays are variables with multiple values
- Index starts at 0

```
PS C:\> $city=("Paris","London","Munich","Rome","Geneva")
PS C:\> $city[2]
Munich
PS C:\> $city.Length
5
```

Power Shell Hashes

- Arrays of key-value pairs

```
PS C:\>
$city=@{"Paris"=970;"London"=1765;"Munich"=309;"Rome"=908;
"Geneva"=321}

PS C:\Users\bryan> $city

Name                     Value
----                     -----
London                   1765
Geneva                    321
Paris                    970
Munich                   309
Rome                     908
```

Power Shell Hashes Adding Values

```
PS C:\> $city.Add("Glasgow",125)

PS C:\> $city

Name                Value
----                -----
Glasgow              125
Paris                 970
Munich              309
Rome                908
London               1765
Geneva              321
```

Power Shell
Accessing Hash using a Key

```
PS C:\> $city."London"
1765
PS C:\> $uk="London"
PS C:\> $city.$uk
1765
```

Power Shell
Accessing Hash from user input

```
PS C:\> $uk = read-host "Enter the Capital of England"

Enter the Capital of England: London

PS C:\> $city.$uk

1765
```

Power Shell Deleting a variable

- Quickest way is to set the variable to null

```
PS C:\> $city

Name                    Value
----                    -----
Glasgow                 125
Paris                   970
Munich                  309
Rome                    908
London                  1765
Geneva                  321


PS C:\> $city=$null

PS C:\> $city
PS C:\>
```

Exercise

- Create a variable $a and assign the value 3 to it
- Use write-host to display the value of $a
- Create a variable $b and assign the value 3.6 to it
- Use write-host to display the value of $b
- Display the variable type of $a and $b
- Create a variable $c and assign the value “3.6” to it, include the quotes
- Display the variable type of $c
- Assign to variable $d the sum of $a and $b

Power Shell Environment

- Environment describes the settings
- Has built in variables

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

Name                         Value
----                        -----
USERNAME                   bryan
```

- Easier to use

```
PS C:\> $env:username

bryan
```

Power Shell Redirection

- Allows the output to be sent to file
- Use > to send (redirect output) to a file

```
PS C:\>
$city=@{"Paris"=970;"London"=1765;"Munich"=309;"Rome"=908;"Geneva"=321}
PS C:\> $city
Name                 Value
----                 -----
London               1765
Geneva                321
Paris                970
Munich               309
Rome                 908

PS C:\> $city > city.txt
```

Power Shell Files

- *get-content* <*file*> reads the content of the file
- Can assign to a variable
- *$variable* = *get-content* <*file*>

```
PS C:\> $content= get-content city.txt
```

Power Shell Pipes

- Screen is called standard output
- | is the pipe symbol and redirects from standard output
- Takes the output of the left command and pipes it to the right command

```
PS C:\> $content | out-file test.txt


PS C:\> get-content test.txt
```

Power Shell more Pipes

```
PS C:\> $content = get-content city.txt
PS C:\> $content.GetType()
IsPublic IsSerial Name          BaseType
-------- -------- ----          --------
True     True     Object[]      System.Array
```

- What can you do with an object type?

```
PS C:\> $content | gm
```

Power Shell get-member

- get-member (or alias gm)
- Returns the properties of that type and what you can do with that type

```
PS C:\> $name="bryan"

PS C:\> $name | gm
```

Power Shell Exercise

- Use notepad.exe to create a file that contains your address. Call the file *address.txt*
- Assign the contents of address.txt to $address
- Display the contents of $address
- Create a multiline string variable called $workAddress with your work address
- Create a variable $myName with your first and last name
- Pipe $myName to gm and work out the methods to make $myName uppercase
- Display the name in uppercase and lowercase

Power shell Scripts

- Save a series of commands to a file
- Invoke repeatedly
- Files have the .ps1 extension
- Built in integrated environment
- Start → Accessories → Windows Power Shell ISE

Power Shell Scripts

- # at the start of a line is a comment
- Write commands in script to be executed sequentially
-

Power Shell Scripts

```
#This is a comment. Always comment your scripts to ease maintenance

###Store today’s year in a variable called “year”
$year=(get-date -Uformat “%Y”)

###Ask the user for their name and store the inputted value in “name”
$name=read-host “Please enter your name?”

###Ask the user for their birth year and store the inputted value in
“birthYear”
$birthYear=read-host “Please enter the year you were born?”

$age=$year-$birthYear

###Respond to the user with the variables
write-host “Hello $name. This year you will be $age”
```

Power Shell Logic and Loops

- A loop allows script to run parts of the script more than once
- Loop is dependant on something or a value
- Saves time for mundane processes

if

- Tests a condition and executes code IF statement is true

```
if (statement)
{
    #enter code to execute
}
```

Simple IF statements

```
$score = read-host "What score did you get in the exam?"
if($score -lt 50)
{
   write-host "The score $score is a fail."
}
if($score -gt 50)
{
    write-host "The score $score is a pass."
}
```

- Note: there is an error in this scripts logic. What is it?

If else

```
$score = read-host "What score did you get in the exam?"
if($score -lt 50)
{
    write-host "The score $score is a fail."
}
else
{
    write-host "The score $score is a pass."
}
```

Nested IF

```
$score = read-host "What percentage did you get in the exam?"
if($score -lt 50)
{
    write-host "$score% is a fail."
}
else
{
    write-host "$score% is a pass."
    #This is a nested if – an if inside an if
    if($score -gt 90)
    {
        write-host "$score% is a really good mark."
    }
}
```

Do Until

```
Do
{
    code
}until (the condition is true)
```

- The code will always be run

Do Until Example

```
Clear-Host
$strPassword ="123"
$strQuit = "No"
Do {
        $Guess = Read-Host "`n Guess the Password"
        if($Guess -eq $StrPassword)
        {
                " Correct guess"; $strQuit ="n"
        }
        else
        {
                  $strQuit = Read-Host " Wrong `n Do you want another guess? (Y/N)"
        }
} # End of 'Do'
Until ($strQuit -eq "N")
"`n Program Completed"
```

Do While

```
Do {
        code

}while (the condition is true)
```

- The code will always be run

Do While Example

```
Clear-Host
$strPassword ="house"
$strQuit = "Guess again"
Do
{
        $Guess = Read-Host "Guess the Password"
        if($Guess -eq $StrPassword)
        {
             " Correct guess"; $strQuit ="n"
        }
        else
        {
             $strQuit = Read-Host " Wrong - Do you want another guess? (Y/N)"
        }
} # End of 'Do'
While ($strQuit -ne "N")
"Program Completed"
```

While Loops

- Easier than Do While/Until

```
while (the condition is true)
{
        code
}
```

- Note code might never get run

While Example

```
Clear-Host
$strPassword ="house"
$strQuit = "Guess again"
While ($strQuit -ne "N")
{
        $Guess = Read-Host "Guess the Password"
        If($Guess -eq $StrPassword)
        {
                " Correct guess"; $strQuit ="n"
        }
        else
        {
                $strQuit = Read-Host " Wrong - Do you want another guess? (Y/N)"
        }
} # End of block statement
"Program Complete."
```

For Loops

- Repeats a block of code a number of times
- For (<initialisation>; <condition>; <iterator>)
```
{
        code
}
```
- For help type “Get-Help about_For”

More For Loops

- The initializer section sets the initial conditions. The statements in this section run only once, before you enter the loop.
- The condition section contains a boolean expression that’s evaluated to determine whether the loop should exit or should run again.
- The iterator section defines what happens after each iteration of the body of the loop.
- The body of the loop consists of a statement, an empty statement, or a block of statements enclosed in braces.
- To set up a for loop that repeats forever, you can leave the initializer, condition and iterator blank:

```
for (; ; )
{
        code
}
```

For Loop Example

```
$table = 5
$count = 0
for ($i = $count; $i -le 100; $i+=5)
{
    write-host $count " x " $table " = " $i
    $count+=1 #same as $count = $count +1
}
```

break command

- You can break out of a for loop by using the break keyword

```
clear-host
$table = 5
$count = 0
for ($i = $count; $i -le 100; $i+=5)
{
    write-host "in loop before if"
    if ($i -eq 25)
    {
        write-host "in if before break"
        break;
        write-host "in if after break" #this line will never be reached
    }
    write-host "in loop after if"
    write-host $count " x " $table " = " $i
    $count+=1 #same as $count = $count +1
}
```

continue command

- you can step to the next iteration by using the continue keyword.

```
clear-host; $table = 5; $count = 0
for ($i = $count; $i -le 100; $i+=5)
{
    write-host "in loop before if"
    if ($i -eq 25)
    {
        write-host "in if before continue"
        continue;
        write-host "in if after continue" #this command is never reached
    }
    write-host "in loop after if"
    write-host $count " x " $table " = " $i
    $count+=1 #same as $count = $count +1
}
```

- I know this ruins the output, but that helps to demonstrate the command

More Date and Time

- Date and Time values held in a specific variable type called *datetime*
- From the powershell prompt type:

  [datetime] $birthday=”3:15pm 19 May 1969”

  $birthday

- The [datetime] tells the environment the type of variable
- Whenever two datetime values are subtracted from each other, the result is of type *timespan*

DateTime example

```
clear-host
$birthday ="3:15pm 19 May 1969"
$birthday
[datetime]$birthday ="3:15pm 19 May 1969"
$birthday
```

timespan

```
clear-host
[datetime]$birthday ="3:15pm 19 May 1969"
[datetime]$today = get-date
$age = $today - $birthday
$age
```

- $age is automatically of type *timespan*

Objects

- $age is an object
- Object.property to get values

```
clear-host
[datetime]$birthday ="3:15pm 19 May 1969"
[datetime]$today = get-date
$age = $today - $birthday
$age.Days
```

Exercises

- Display how old a person is in years using the timespan object
- With if statements calculate if you have lived for
  - A million second
  - A million minutes
  - A million hours
- Write a times table program
  - Ask the user for the table to be calculated
  - Ask the user how many times they want to calculate
  - Implement using one of the while loops
  - Implement using a for loop