Windows Scripting

OPS102 Week 10 Class 1

Chris Tyler/John Sellens

July 19, 2024

Seneca Polytechnic

Outline

Scripting on Windows Introduction

Windows vs. Linux Scripting

Windows shell scripting is similar to Linux shell scripting in many ways.

However, since Windows and Linux (and other Unix-like operating systems) have different technical heritages, some details of the syntax and approaches are very different ...

In this slide deck, many of the examples are roughly equivalent to the examples in the “Linux Shell Scripting” slides (from weeks 8 and 9), and it may be useful to compare them side-by-side.

An Abundance of Shells

We've looked at scripting on Linux systems using the **bash** shell –- which is the most widely-deployed shell on Linux and other Unix-like systems. There are many other shells on similar systems, including:

- sh – an early Unix shell ("Bourne Shell"), written by Steven Bourne
- ksh – the Korn shell, written by David Korn
- csh – the C shell, which has a C-like syntax, from UCBerkeley
- zsh – the Z shell, similar to ksh
- fish – the Friendly Interactive shell

Many of these have a syntax based on and similar to the Bourne shell, which is standardized in the POSIX.1 standard (or IEEE Standard 1003.1).
https://en.wikipedia.org/wiki/Unix_shell

An Abundance of Shells cont'd

On Windows, there are two main shells:

- CMD – The Windows command shell, which is the traditional Windows shell. Although based on the old DOS command syntax, the CMD shell has been considerably expanded, with many new features added over the last few years.
- PowerShell – this is a new Windows command shell, which combines scripting with object-oriented programming. It is pre-installed for interactive use on current Windows systems; however, the execution of PowerShell scripts is disabled by default on all Windows "client" (non-server) systems, including Seneca lab computers.

Windows Scripting – Why Not PowerShell?

Due to PowerShell scripting being disabled by default, and because object oriented programming is not taught in first semester programming courses, we will focus on scripting using the traditional Windows CMD shell.

Windows CMD is a useful basic skill to have, but is perhaps most useful for relatively simple tasks. More complicated tasks are likely better suited to less rudimentary scripting languages.

Cross-Platform Scripting

There are several possible approaches to writing scripts that will work on both Linux and Windows systems, as well as other common operating systems such as Mac OS:

- Bash or Zsh – bash is the default shell on most Linux systems, and zsh is the default shell on most Mac OS systems; either shell can be easily used on either system. Both shells are available as third-party add-ons for the Windows operating system, usually shipped with a collection of GNU utilities compiled for use with Windows (e.g. https://gitforwindows.org/)
- PowerShell – installed by default on current Windows systems, it is also available for Linux and Mac OS systems, although it is not commonly used on those systems yet (see https://github.com/PowerShell/PowerShell).

Other Interpreted Languages

In addition to shell scripting languages, there are other interpreted languages that are well suited for cross-platform development, including Python and Perl.

When should you use a shell scripting langauge?

- Shell scripting languages are ideally suited for process control – executing, managing, and combining external programs to accomplish tasks.
- Shell scripting languages are not well-suited for implementing advanced processing algorithms, because they generally lack features such as good floating-point support, typed variables, and strong support for large arrays and hashes.

Windows Shell Scripts vs. Batch Files

DOS and early Windows systems were inherently interactive in nature, and early scripts were called “batch files” because they were viewed as similar to non-interactive batch processing on mainframe systems.

This terminology has stuck, and Windows shell scripts are still often called “batch files” (hence the occasional use of the `".bat"` extension instead of `".cmd"`).

The concept of a “batch file” and a “shell script” are roughly equivalent.

Windows CMD Has a Few Quirks

- The Windows CMD shell has a few “quirks” and limitations. We shall outline a few of these in these notes.
- We are not going to cover all the possibilities here.
- If you want to learn even more about Windows CMD, see https://en.wikibooks.org/wiki/Windows_Batch_Scripting

Windows CMD Scripting

Basic Requirements for Shell Scripts

Remember these requirements? They apply to Windows scripts too:

1. Create a text file containing shell commands.
2. Tell the operating system *which* shell to use to execute the commands.
3. Ensure that the script file has the appropriate permissions.

1 – Create a file containing shell commands

- Use any text editor
- Use the same commands that you would type at the command-line (though in Windows, there are a few small syntax differences between the command-line and script files)
- Save the file

Moving text files between Systems: Line Endings

Windows traditionally places a carriage-return (CR, code 13, \r) and a newline character (NL, code 10, \n) at the end of each line.

Linux traditionally places only a newline character at the end of each line.

MacOS uses only a carriage-return at the end of each line.

When transferring files between systems, the end-of-line codes will often be translated, but when this does not take place, it can cause confusing issues. The `"dos2unix"` and `"unix2dos"` utilities on Matrix can be used to force a conversion of the end-of-line characters.

2 – Tell the operating system which shell to use

- In Windows, the filename extension is used to associate a file with a program, and this mechanism is used to associate a script with a command interpreter.
- For CMD scripts, the extension `".cmd"` is used.
  - For historical reasons, the extension `".bat"` is also accepted.
- For PowerShell scripts, the extension `".ps1"` is used.
  - The reason that `".ps"` isn't used is that that extension was already used for PostScript (page description language) files.
  - We're not going to write PowerShell scripts in this course.

3 – Set the correct permissions on the file

- On Windows, the ability to read the script file is sufficient.
- This is the default permission, so no change is usually required for scripts that you create for your own use.
- The situation may be different for scripts that are shared to other machines over the network or to other users on your system.

A Word about Command Echos

- Windows defaults to echoing each command in a script before executing it.
  - The opposite of the default behaviour in the bash shell.
  - In bash, echoing can be turned on with the `"-x"` option).
- If you do not want each command to be displayed as it runs, you can:
  - Add an @ sign in front of each command, or
  - Issue the `"echo off"` command.
- Usually, you'll start your CMD scripts with this as the first line:

```
@echo off
```

A Word About Comments

- In Windows CMD scripts, comments are created with the `"remark"` command.
- It is often shortened to `"rem"`.

```
rem This is a comment
```

- You can sometimes get away with using `::` (two colons) as a comment indicator – it's actually an invalid label (see the discussion of `GOTO` later in these notes) – but it doesn't always work e.g. a label must usually be followed by a command.

```
:: this sometimes works as a comment, but not always
```

  See how-do-i-comment-on-the-windows-command-line from Stack Overflow for more discussion.

A Word About Processes

- In Linux, bash scripts are run by a separate copy of bash, in a separate process.
  - A script doesn't change the settings of, or variables in, its parent bash shell.
- In Windows CMD, a script runs in the same CMD instance and process as the command line.
  - e.g. Any variable set in a CMD script is also set in your CMD window.
  - This can cause unexpected behaviour if you're not careful with variables in your CMD script.
  - e.g. Run a script twice, a variable might already be set in the second run.
  - But see `SETLOCAL` on slide 25

Demo: Basic scripts

- Let's write some simple scripts using commands that we know.
- Scripts act like any other executable file, so we can type the script name as a command. It is not necessary to include the `".cmd"` extension.
- By default, the current directory is searched when looking for commands, so as long as the script name does not collide with an existing command name and the script is in the current directory, just typing the name (without a leading .\) is sufficient:

```
C:\Users\jsmith> scriptname
```

Setting Variables

- To set a variable:

  ```
  set VARIABLE=VALUE
  ```

- Variable names start with a letter and can contain letters, numbers, and underscores.
- Case does not matter! `"A"` and `"a"` are the same variable.
- Don't put spaces on either side of the equal sign (just like in bash).
- Note: Like bash, and unlike other languages such as C, you don't need to declare the variable or specify the type of data (e.g., integer, string) which it will hold.

Accessing Variables

- To access a variable value, surround the variable name with percent signs: `%VARIABLE%`
- The value of the variable is substituted into the command.
- Just like the use of `"$variable"` in bash.

Variables: A Simple Example

```
C:\> set WHAT=World
C:\> echo Hello %WHAT%
Hello World
C:\> echo Hello %what%
Hello World
```

```
C:\> type hello.cmd
@echo off
set WHAT=World
echo Hello %WHAT%
C:\> hello
Hello World
```

Quoting in the Windows Shell

- Quoting in the Windows shell is very different from bash!
- Using single or double quotes causes the quotes themselves to be included as part of the string or argument in most cases, **but not** when dealing with a filename:

```
C:\> echo "Hello"
"Hello"
C:\> echo test > "test file"
```

- Think of it as CMD handling redirection, and each command handling quoting (or not).

Escaping in the Windows Shell

- Escaping in the Windows shell is very different from bash!
- Escaping characters to remove their special meaning is performed using the caret (^) symbol in Windows:

```
c:\> echo Lost ^& Found
Lost & Found
```

- When piping, a CMD subshell is started for each command in the pipeline, and it is necessary to use triple caret symbols ^^^ to escape characters:

```
c:\> echo Lost ^^^& Found | find "Lost"
Lost & Found
```

- Multiple commands can be separated by `&`, just like `;` in bash.

Escaping in the Windows Shell – % is Special

- The percent sign (%) is a special case.
- On the command line, it does not need quoting or escaping unless two of them are used to indicate a variable, such as **%OS%**
- But in a batch file, you have to use a double percent sign (%%) to yield a single percent sign (%).
- Enclosing the percent sign in quotation marks or preceding it with caret does not work.
- From https://en.wikibooks.org/wiki/Windows_Batch_Scripting on “Quoting and Escaping”

Variables vs Environment Variables

- By default, all variables are environment variables, inherited by child processes.
  - And remember, variables set in a CMD script are also set in the enclosing CMD window.
- You can stop script variables from becoming environment variables with
  `SETLOCAL`
  See also the `ENDLOCAL` command.
- Environment variables are commonly used to pass configuration information to programs and to configure how programs operate.
- Environment variables are used by all processes, not just the shell!

Common Environment Variables

Env Variable	Purpose
CD	Current directory
TIME	Current time in HH:MM:SS format
DATE	Current date in local format
ERRORLEVEL	The error code / exit status of the last command executed
PATH	A semi-colon (;) (not colon) separated list of directories to be searched for commands and scripts
PROMPT	The interactive shell prompt (see help – `"prompt /?"` – for special escape sequences that may be included)
RANDOM	A random integer (0-32767)

Viewing Environment Variables

- See all current environment variables and their values:
  ```
  C:\> set | more
  ```
  Redirection and pipes in CMD work similarly to bash.
- To unset a variable use:
  ```
  C:\> set varname=
  ```

PATH and PROMPT Environment Variables

The "PATH" and "PROMPT" environment variables are usually set using the corresponding "PATH" and "PROMPT" commands instead of the "SET" command. See the online documentation ("help prompt" or "help path") for details.

```
C:\> set varname=
C:\> PROMPT COMMAND:
COMMAND:PROMPT $G$S
>

> PATH %PATH%;D:\SomeNewDirectory
```

Reading a Variable Value from Stdin: set /p

You can read values from standard input (stdin) and assign them to a variable with the `"set"` command using the `"/p"` option (aka switch):

```
C:\> set /p NAME=Enter your name:
Enter your name: J. Doe

C:\> echo %NAME%
J. Doe
```

Demo: Variables in a Script

```
@echo off
set /p NAME=Please enter your name:
echo Please to meet you, %NAME%
set /p FILE=Please enter a filename:
echo Saving your name into the file...
echo NAME=%NAME% >> %FILE%
echo Done.
```

Arithmetic!

- CMD can do *integer* arithmetic (sound familiar?)
- To evaluate a arithmetic expression and store the results in a variable, use the `"SET"` command with the `"/a"` option.
- When used interactively, the result of the expression evaluation will be output to stdout; this doesn't happen in scripts.

```
> set A=100
> set B=12
```

```
> set /a X=A*B
1200
> echo %X%
1200
```

```
> set /A A+=1 >NUL:
> echo %A%
101

> set /a C=A*B*2 >NUL:
> echo The answer is %C%
The answer is 2424
```

Arithmetic! cont'd

- You can perform more than one arithmetic evaluation and assignment in one `"SET /A"` command by separating the expressions with a comma (,).
- Some characters used in arithmetic expressions, such as the caret symbol, may need to be quoted or escaped to function correctly.
  - The caret likely isn't used in arithmetic, there is no exponentiation operator, so this may be a bad example.
- Percent signs, when used in arithmetic expressions (as the modulo operator), need to be doubled (%%) to avoid confusion with the percent signs placed around variable names

Exit Status Code: ERRORLEVEL

- The special variable %ERRORLEVEL% can be used to find out the exit status of the last command executed:

```
> dir \foo\bar\baz
The system cannot find the path specified.
> echo %ERRORLEVEL%
1
```

- Similar to Linux/Unix, zero means ok, non-zero means not ok.

Setting Exit Status Code

- To set the `"ERRORLEVEL"` and exit from a CMD script, use

  ```
  EXIT /b n
  ```

  where *n* is the desired `"ERRORLEVEL"` value.
- The `"/b"` flag is important – it means only exit out of the current CMD (batch) script, and not out of the entire CMD session.

Conditional logic: if / else

The `"if"` command takes a test, and uses the result of the test to control the execution of one or more commands. An `"else"` clause is optional. If included, the first conditional commands should be placed in parenthesis.

`if` *`test`* `then` *`list`*

```
rem Note the grouping parenthesis are required before else
if test then ( list ) else list2
```

A *list* is a single command, or multiple commands inside parenthesis, separated by newlines.

Conditional logic: if / else – Examples

```
C;\> set A=Blue
C;\> set B=Orange
C;\> set C=Blue

C;\> if %A%==%C% echo Strings A and C match
Strings A and C match
C;  if %A%==%B% (echo Same!) else echo Different!
Different!
```

Tests 1: Filesystem entries

Tests if a filename exists (regardless of the entry type – file or directory).

```
EXIST filename
```

Evalutes true if *filename* is an existing file or directory.

Tests 2: String Equality

Test for string eqality

```
string1==string2
```

Evalutes true if the strings match.

Tests 3: String and Numeric Comparisons

These tests accept two string arguments, both strings or both integers, which are compared. Adding the `"/i"` switch will make string equality (== and EQU) comparisons case-insensitive (UPPER/lowercase).

`value1 EQU value2`	True if the values are equal
`value1 NEQ value2`	True if the values are not equal
`value1 LSS value2`	True if value1 is less than value2
`value1 LEQ value2`	True if value1 is less/equal to value2
`value1 GTR value2`	True if value1 is greater than value2
`value1 GEQ value2`	True if value1 is greater/equal to value2

Tests 4: Variable Definition, Errorlevel

- Test to see if a variable is defined:

  ```
  DEFINED variable
  ```

  Evaluates true if variable is defined
- Test to see if the ERRORLEVEL is above a threshold:

  ```
  ERRORLEVEL value
  ```

  True if ERORRLEVEL >= value
- Or just use: `%ERRORLEVEL% GEQ value`

Notes about IF and these Tests

- These tests work only with the `IF` command.
- The `IF` command can be used with `GOTO` and a label:

```
IF test GOTO :skip
...
:skip
```

- Using a `GOTO` in a loop will make the shell forget about the loop, regardless of where the label is located.

Examples of using test: ERRORLEVEL

```
@echo off
VER | FIND "Version 10" >NUL:
IF ERRORLEVEL 1 (ECHO Not Windows 10.) ELSE ECHO Windows 10.

rem The test above could be rewitten as %ERRORLEVEL% GTR 0
rem The NUL: device is the Windows equivalent of /dev/null
```

Negating and Combining Tests

- You can negate (invert) a test with the NOT operator:

```
SET N=myfile.txt
IF NOT EXIST %N% ECHO The file "%N%" does not exist.
```

- You cannot combine tests – there are no “and” or “or” operators.

Examples of using test: Integers vs Strings

```
@echo off
rem intcmp.cmd - compare as integers
SET /A A=11,B=2
IF %A% GTR %B% (
    ECHO %A% is greater than %B%
) ELSE ECHO %A% is less than or equal to %B%
@echo off
rem strcmp.cmd - compare as strings
SET /A A=11,B=2
IF "%A%" GTR "%B%" (
    ECHO %A% is greater than %B%
) ELSE ECHO %A% is less than or equal to %B%
```

Examples of using test: Integer Numbers

```
@echo off
REM need double %% in a file, single % on command line
SET /A COIN=%RANDOM% %% 2
IF %COIN% EQU 0 (ECHO Heads!) ELSE ECHO Tails
```

See note about % being special on slide 24

Script Parameters

- It's useful to be able to call a script with positional parameters (arguments).
- These can be accessed within a script as `%0`, `%1`, `%2`, `%3`, and so forth.
- `%0` is the name of the script itself.
- The `shift` command gets rid of the first parameter and shifts every parameter to a lower number.
- Very similar to Linux bash, except uses `%` rather than the `$` used in bash.
- Note that these "variables" don't need a trailing `%` to be substituted in.

Script Parameters cont'd

- `%*` returns all of the parameters
- On the command line, parameters may be separated by:
  - Space (or Tab)
  - Comma (,)
  - Semicolon (;)
  - Equal sign (=)

Script Parameters cont'd

```
C:\> type params.cmd
@echo off
ECHO PARAM 0: %0
ECHO PARAM 1: %1
ECHO PARAM 2: %2

C:\> params red green blue
PARAM 0: params
PARAM 1: red
PARAM 2: green
```

Script Parameters cont'd

```
C:\> type params-shift.cmd
@echo off
ECHO List of all arguments: %*
:start
IF "%1"=="" GOTO :done
ECHO %1
SHIFT
GOTO :start
:done
C:\> params-shift yellow orange red
List of all arguments: yellow orange red
yellow
orange
red
```

Looping

- Loops through list of files:
  `FOR %variable IN (`*`files`*`) DO `*`list`*
- Loops through list of directories:
  `FOR /D %variable IN (`*`files`*`) DO `*`list`*
- Loop through a range of numbers:
  `FOR /L %variable IN (start; increment; end ) DO `*`list`*
- A *list* is a single command, or multiple commands inside parenthesis, separated by newlines.
- For help: `for /?`

Looping – Variables

The names of the controlling variables in FOR loops are different from other variables:

- Must be a single letter.
- The variable name is case sensitive.
- The variable name is written with a single preceding % sign at all times (not just when reading) when used from the command line, or double preceding %% signs inside a script

This inconsistency in variable usage seems “unfortunate”.

Looping - Delayed Expansion with IF or FOR

- When the body of an `IF` statement or `FOR` loop is executed, the variables are expanded (replaced by their values) before the `IF` or `FOR` loop begins.
- That means that any variables that are contained in the body of the `IF` or `FOR` loop have their values locked in and they can't be changed while the loop is executing.
- To allow updated variable values to be accessed within an `IF` or `FOR` loop:
  - Set the EnableDelayedExpansion option:
    `SETLOCAL EnableDelayedExpansion`
  - Change any variables which will be updated during the execution of the loop by replacing the percent-signs (%) with exclamation marks (!):
    `%ERRORLEVEL%` becomes `!ERRORLEVEL!`

Looping: FOR *variable* IN (*files*) DO *list*

- This type of loop accepts a list of files, or a filename pattern.
- The loop will iterate through the list of files, assigning each one sequentially to the control variable.
- Other strings can be used as though they are filenames as long as they do not contain wildcards – there is no test performed to see if the names actually exist or are potentially valid.

Looping: FOR *variable* IN (*files*) DO *list*

```
@echo off
SETLOCAL EnableDelayedExpansion
FOR %%F IN (*) DO (
  rem The CHOICE command asks the user a Y/N question
  CHOICE /M "DELETE %%F"
  IF !ERRORLEVEL!==1 (
    ECHO ...Deleting %%F
    DEL %%F
  ) ELSE (
    ECHO ...Skipping %%F
  )
)
```

Looping: FOR /L (*start*, *step*, *end*) DO *list*

This type of loop counts forward or backwards from *start* to *end* by a given *step*. For example:

```
@echo off
rem Count from 0 to 5 in increments of 1
FOR /L %%I IN (0, 1, 5) DO ECHO ... %%I ...

rem Count from 4 to 0 in increments of -1
FOR /L %%I IN (4, -1, 0) DO ECHO ... %%I ...
```

Summary

Summary of Windows Scripting

- Windows CMD scripts are similar in structure to Linux bash scripts.
- But there are significant differences and limitations.
  - e.g. If you want a while loop, you'll likely have to implement it with `IF`, `GOTO`, and labels.
- And some elements of the syntax are just a little unexpected.
- We did not cover all the possibilities here.
- You can learn more at https://en.wikibooks.org/wiki/Windows_Batch_Scripting