Introduction to BASH scripting

~/IFT383/Module2/Lecture02x01

Objectives

- Write and execute simple shell scripts
- include UNIX system commands in shell scripts
- Pass parameters and arguments to shell scripts
- Use arithmetic operations in shell scripts
- Make shell scripts interactive

A simple script

- In its simplest form; a shell script is a collection of commands to be ran in series
- Shell scripts typically have the suffix **.sh** or **.bash**
- The first line contains the interpreter that should be used to execute the script
  - This line is denoted by the special character combination **#!**
    - Sometimes called a crunch-bang or sha-bang

```
#!/bin/bash
touch new_file
cp new_file new_file_copy
rm -f new_file new_file_copy
```

Executing a script

- Make the file executable
  - chmod +x ./filename.sh
- Running the script
  - ./filename.sh
  - bash filename.sh

Script comments

- Adding comments to a script add notes for you and anyone inheriting your script in the future
  - Do the future you a favor; document everything!
- Lines starting with a hash **#** character will be ignored by the shell and can contain comments

```
#!/bin/bash
# This script creates two files and deletes them
touch new_file
cp new_file new_file_copy
rm -f new_file new_file_copy
```

A real-world example in PowerShell

```
#REQUIRES -Version 5.0
# WinPE Automated post-master customizations
#
# Automatically injects MS-LAPS and MBAM extensions
# into a post-mastered SCCM WinPE image
#
# Chelsey Ingersoll <cjingers@asu.edu>
# Engagement and Advising Services
# University Technology Office - Endpoint
# Arizona State University
# (C) Arizona Board of Regents
$script:WORKING_DIR = split-path $SCRIPT:MyInvocation.MyCommand.Path -parent
$script:ISO_DIR = Join-Path "${script:WORKING_DIR}" "\ISO"
$script:nowTIme = (get-date).ToFileTime()

# Fail if RSAT PowerShell modules not installed
if ( !(Test-Path -Path "${ENV:WINDIR}\SysWOW64\WindowsPowerShell\v1.0\Modules\ActiveDirectory") ) {
    Write-Error -Message "RSAT is required for this task. Make sure RSAT is installed with PowerShell mo
    Exit 1
}
```

Real world example (cont'd)

```
# Mount the WIM
$wimDir = Join-Path $ENV:HOMEDRIVE $script:nowTIme
New-Item $wimDir -Type 'Directory' | Out-Null
Mount-WindowsImage -ImagePath $wimFile -Path $wimDir -Index 1 -Optimize | Out-Null

# apply INI files to extracted WIM
Copy-Item -Path (Join-Path $script:WORKING_DIR "\customizations\TSconfig.ini") -Destination (Join-
Copy-Item -Path (Join-Path $script:WORKING_DIR "\customizations\DSStart.bat") -Destination (Join-P

# Copy over all of our DS script files
Copy-Item -Path (Join-Path $script:WORKING_DIR "\customizations\DS") -Destination (Join-Path $temp

# Grant ourselves access to the folders within the WIM so we can copy some addtional modules
$rule = new-object System.Security.AccessControl.FileSystemAccessRule("BUILTIN\Administrators",'Fu
foreach ($file in $(Get-ChildItem -Path (Join-Path $wimdir 'windows') -Directory -recurse -ErrorAc
    $facl = Get-Acl -Path $file.fullName
    $facl.AddAccessRule($rule)
    Set-Acl -Path $file.fullName -AclObject $facl
}
```

Variables

- Declaration
  - general-use variables (contain strings)
    - var_name=word
    - var_name=”a string of characters with spaces”
    - another_variable=9001
  - special types
    - uppercase string: declare -u VAR_NAME=”STRING”
    - lowercase string: declare -l VAR_NAME=”string”
    - integer: declare -i VAR_NAME=9001
    - (more to come in future slides!)
- Assignment
  - VAR_NAME=”something interesting”

Variables

- Retrieving
  - echo $VAR_NAME
  - echo ${VAR_NAME}
    - This form is especially useful when building strings, example;

```
VAR1=123
VAR2=789
echo “${VAR1}456${VAR2}”
```

Too many quotes!

- “Double quotes”
  - The shell interprets the contents looking for variable names for string expansion
- ‘Single quotes’
  - The shell passes the contents as a literal string
- `grave quotes`
  - The key to the left of the 1 on your keyboard
  - Shell runs the contents as a command and uses the result as the string

Example (variables.sh)

```
#!/bin/bash
VAR1=’Hello,’
VAR2=’World!’
VAR3=`date` # using grave quotes
echo “${VAR1} ${VAR2} ${VAR3}”
echo ‘${VAR1} ${VAR2} ${VAR3}’
```

```
Hello, World! Sat Jan 12 13:19:17 …
${VAR1} ${VAR2} ${VAR3}
```

Positional parameters

- Arguments can be provided to your script in the same way they are provided to normal linux commands
  - ./positional.sh ‘arg1’ ‘arg2’ ‘arg3’
  - The value of each **positional parameter** is stored in $1, $2, $3 and so on
  - $0 contains the name of the script
- Tools
  - **shift** shuffles the positional parameters down by 1 (default)
  - set (by default) can be used to overwrite the value of $1

Example (positional.sh)

./positional.sh kaboom sloosh

```
#!/bin/bash
echo $1
shift
echo $1
set "ORB!"
echo $1
```

```
kaboom

sploosh

ORB!
```

Arithmetic Operators (math)

- Integers
  - use the syntax: $((expression)) or ((expression))
    - $(( )) when you want the result returned to stdout
  - Example;
    - echo $((1 + 2 + 3 + 4))
      10
    - echo ((1 + 2 + 3 + 4))
      <error>
- Floating point
  - Use the command **bc**
    - echo "scale=2; (38.00*0.15)+38.00" | bc
      - scale is the floating point precision

Example (tipcalc.sh)

```
#/bin/bash
# NOTE: grave quotes are used on the following line!
TOTAL=`echo "scale=2; ($1*0.15)+$1" | bc`
echo "Your total bill, including 15% tip is; ${TOTAL}"
```

```
./tipcalc.sh 30.00
Your total bill, including 15% tip is; 34.50
```

Adding interactivity

- **read** is a shell command that will prompt the user for input
  - read -p “Instructions to the user” VAR_NAME OPTIONAL_VAR
    - -p is optional; but useful for providing instructions
    - VAR_NAME is the variable the input will be stored
    - OPTIONAL_VAR If the input contains spaces, the second word will be stored here
      - run **read --help** for more options

Example (tipcalc2.sh)

```
#/bin/bash
read -p "Please enter the total on your bill: " BILL
TOTAL=`echo "scale=2; ($BILL*0.15)+$BILL" | bc`
echo "Your total bill, including 15% tip is; ${TOTAL}"
```

```
./tipcalc2.sh
Please enter the total on your bill: 30.00
Your total bill, including 15% tip is; 34.50
```

Arrays and Lists

- An **array** is a collection of values that are stored in one variable and accessed using an **index**
- The first element in the array has an index of 0
- Declaration
  - implicitly
    - vehicles[0]=truck
      vehicles[1]=car
  - Using a list
    - vehicles=(truck car)
  - The **declare** and **read** commands also support arrays
    - **read -a VAR_NAME** places input in an array at VAR_NAME

Arrays and Lists (cont.)

- Accessing an array
  - by index
    - ${MY_ARRAY[0]}
  - all elements
    - ${MY_ARRAY[*]} or ${MY_ARRAY[@]}
  - Get a count of elements in an array
    - ${**#**MY_ARRAY[*]} or ${**#**MY_ARRAY[@]}
    - will come in handy for **loops** and **conditionals**

Example (tipcalc3.sh)

```
#/bin/bash
read -p "Please enter the cost of three items on your bill, separated by spaces: " -a BILL
SUBTOTAL=`echo "scale=2; ${BILL[0]} + ${BILL[1]} + ${BILL[2]}" | bc`
DUE=`echo "scale=2; ($SUBTOTAL*0.15)+$SUBTOTAL" | bc`
echo "You entered: ${BILL[*]}"
echo "Your total bill, including 15% tip is; ${DUE}"
```

```
./tipcalc3.sh
Please enter the cost of three items on your bill, separated by spaces: 3.50 3.50 3
You entered: 3.50 3.50 3
```

Exit codes

- Many commands use **exit codes** between 0-255 to give an indication of what happened
- A common convention is to return 0 on success, and >0 if something went wrong
  - This is not universally true; programmers can use this as they see fit
  - The **man** page will often have a section for exit codes
- The exit code of the last ran command is stored in $?
- From the **ls** man page; an example of exit codes;

```
Exit status:
    0      if OK,

    1      if minor problems (e.g., cannot access subdirectory),

    2      if serious trouble (e.g., cannot access command-line argument).
```

Example (exit.sh)

```
#!/bin/bash
LIST=`ls -l /doesnotexist 2> /dev/null`
echo "I ran ls and it said: ${?}"
```

```
I ran ls and it said: 2
```

Flow control (decisions)

- We can control what portions of the script are evaluated based on conditions
- The pattern is similar to other programming languages such as python

```
if [ condition ]; then
    # do something if our condition is met
elif [ condition ]; then
    # when the first condition fails; “else if” can evaluate another condition
    # elif is optional
else
    # when the condition for the if line fails; evaluate these lines
    # this section is optional
fi
```

Conditions with integers

- Conditions with integers follow the form; [$a -?? $b]
  - where ?? is one of;
    - eq - a=b
    - ge - a>b or a=b
    - gt - a>b
    - le - a<b or a=b
    - lt - a<b
    - ne - a is not equal to b

Example (codematch.sh)

```
#!/bin/bash
# Conditional based on an integer
declare -i PIN
read -p "Please enter your 4-digit PIN: " PIN
declare -i VALID=0
VALID=`echo -e "${PIN}" | grep -Ec "^[0-9]{4}$"`
if [ $VALID -eq '0' ]; then
        echo "You did not enter a valid code!"
elif [ $PIN -eq '1234' ]; then
        echo "Access Granted!"
else
        echo "Sorry, wrong code..."
fi
```

Conditions with strings

- String length
  - [-n $VAR]
    - $VAR is a string with a non-zero length
  - [-z $VAR]
    - $VAR is a string with a length = 0
- String equivalence
  - [$VAR1 = $VAR2]
    - Strings are equal
  - [$VAR1 != $VAR2]
    - Strings are not equal
- more possibilities are detailed on the man page for “test”

Example (quiz.sh)

```
#!/bin/bash
# Conditional based on an integer
declare -l ANSWER=''
read -p "The author of the original version of grep was _______ Ritchie.: "
ANSWER
if [ $ANSWER = 'dennis' ]; then
        echo "Correct!"
else
        echo "Sorry, try again"
fi
```

Conditions with mathematical expressions

- use (( )) instead of [ ] to perform a mathematical condition check
  - Examples;
    - (($a ? $b)) where ? can be; < <= == != >= >
- You can also use mathematical operators in the condition
  - Example;
    - (($a**2 == 8))
      - $a squared equals 8
    - (($a / 2 > 100))
      - $a divided by 2 is greater than 100
- You can increment and decrement integer variables as part of a condition
  - this becomes important when we get to loops
  - (($a++)) and (($a--)) are performed AFTER being evaluated
  - ((++$a)) and ((--$a)) are performed BEFORE being evaluated

Example (secret_number.sh)

```
#!/bin/bash
declare -i NUMBER=0
read -p "Guess the secret number between 1 and 100: " NUMBER
if (( ($NUMBER + 30) / 2 == 36)); then
        echo -e "You got it!"
else
        echo -e "NOPE! try again"
fi
```

Case statement

- Comparing a variable to many possible conditions can result in a very complicated if statement
- Case allows us to take a variable and allow the shell to select which path to take from any number of possibilities
- The general format;

```
case $VARIABLE in
pattern 1 )
pattern 2 | pattern 3 )
*)
esac
```

- The commands to be ran follow each case and end with a ;; (double semicolon)

Example (adventure.sh)

```
#!/bin/bash
declare -u CHOICE1="
read -p "You can move North, East, South or West. " CHOICE1
case $CHOICE1 in
        N|NORTH)
                echo -e "You move towards the house, and notice a
sleeping dog on the porch";;
        E|EAST)
                echo -e "You walk towards the side of the house...";;
        S|SOUTH)
                echo -e "You move away from the house and begin
walking down a street";;
        W|WEST)
                echo -e "You are eaten by a grew. Game over.";;
        * ) echo -e "Invalid selection, please try again";;
esac
```

Loops - for

- The for loop runs a block of commands for each element in a list
- Lists can come from arrays, or strings separated by spaces, tabs or newline chars
  - You can override this by setting the **IFS** variable “Internal Field Separator”
- Single-line form;
  - for ITTERTOR in $LIST; do echo $ITTERATOR; done
  - Example
    - for i in `ls`; do echo $i; done
- multi-line form
  - for ITTERATOR in $VAR; do
    echo $ITTERATOR
    done

Example (echo.sh)

```
#!/bin/bash
read -p "Enter a list of colors, separated by a space: " -a COLORS
for COLOR in ${COLORS[*]}; do
        echo $COLOR
done
IFS=','
read -p "Enter a list of numbers, separated by a comma" NUMBERS
for NUMBER in $NUMBERS; do
        echo $NUMBER
done
unset IFS
```

Example continued (echo.sh)

```
# C-style for loop
read -p "Enter a list of names separated by a space: " -a NAMES
for (( i=0; $i < ${#NAMES[*]}; i++ )); do
        echo -e "Input at position ${i} was ${NAMES[$i]}"
done
```

Loop - while and until

- While
  - Executes a block of commands **while** a condition is true
- Until
  - Executes a block of commands **until** a condition is true
- Example
  - The format is the same; with a differant keyword

```
while [ condition ]; do
    # do something
done
```

Example (multiadd.sh)

```
#!/bin/bash
declare -i SUM=0
declare -i INPUT=0
while (($INPUT > 0 || $SUM == 0)); do
        read -p "enter an integer, or nothing to stop: " INPUT
        SUM=$(( $SUM + $INPUT ))
done
echo -e "Your total is: ${SUM}"
SUM=0
INPUT=0
until (($SUM > 10)); do
        read -p "You have ${SUM} apples, how many apples to add? "
INPUT
        SUM=$(($SUM + $INPUT))
done
echo -e "WOW! ${SUM} apples! that is too many apples!!"
```

Functions

- Functions group commands into a block that can be invoked anywhere following the point they were defined
- A function captures anything sent to stdout and return that to whatever invoked it
- General form;

```
function my_function {
    # do something cool!
    echo “This string will be sent to the caller”
}
```

Example (lunch.sh)

```
#!/bin/bash
declare -l LUNCH=''
declare -i STEP=1
function makeLunch {
        case $STEP in
                2) echo "Adding turkey meat...";;
                3) echo "Adding more bread...";;
                4) echo 'done';;
                *) echo "Adding bread...";;
        esac
}
until [ "${LUNCH}" = "done" ]; do
        LUNCH=$(makeLunch)
        echo -e "Preparing lunch: ${LUNCH}"
        ((STEP++))
        sleep 1
done
```

Dee you in Module 3: AWK scripting