Shell Scripting

UBCO Master of Data Science – DATA 541

Today's Class

Shell Basics

Shell Commands

Array and String

Functions

Files

Script

A **script** is a list of commands which are run by a program written in a specific language such as Python

Scripts can be edited by text editors or preferably by a specialized editor that also allows the script to run

- Example:
  - Schedule a job for specific time of the day

Do not require compilation step

- Example: JavaScript, PHP, VBScript, Python

Linux Shells

Provides users an environment to execute commands

Shells provide a user interface (command prompt) to the underlying Unix operating system

Many shells are available, but with some differences

- Bourne Shell (sh)
- C Shell (csh)
- TC shell (tcsh)
- Bourne Again Shell (bash)

Shell Script

Series of Linux commands in a text file that can be executed on a Linux shell in top-down fashion

The Linux shell provides a high-level, general-purpose, interpreted, interactive programming environment

Mainly used for automating Linux tasks but also for writing integrated workflows

- Example: Generating many copies of a file

Advantages of Shell Scripting

Shell scripting is useful for quick automation and advantageous to use it in many applications

- Fewer lines of code than C, Java (similar to Perl, Python)
- No compilation necessary
- Vast command library
- Save coding time and automate computing tasks

Variables

Provide a location to "store" data we are interested in

- Strings, integers, decimals, characters, arrays, …

these are all stored as strings

Variable Example

To navigate to the root directory, use `cd /`

To check directories under root, use `ls`

```
bin   dev  home  lib    lib64   media  opt   root  sbin  srv  tmp  var
boot  etc  init  lib32  libx32  mnt    proc  run   snap  sys  usr
```

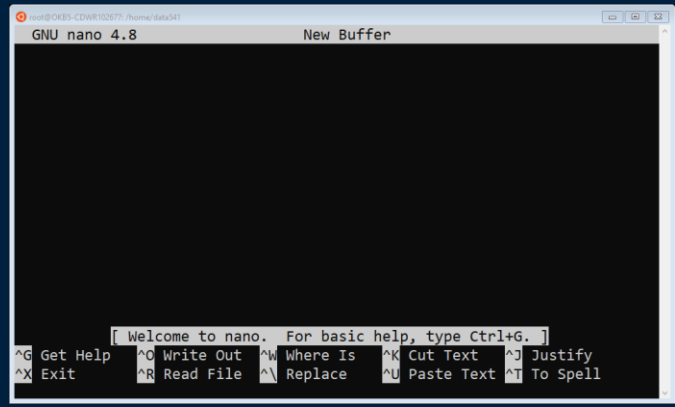
To navigate to the root directory, use `cd /`

To create a new directory, use `mkdir directory_name` and navigate there

To open nano (text editor), just type in `nano`

at the command prompt.

To exit Nano, type `CTRL + X`



Variable Example

```
#!/bin/bash
#Comment: Course information
course="Data541"
school="UBCO"
echo "I am taking  $course at the $school"
```

Path to bash

Comments

Variable declaration

Echoing some text

Save the file a **.sh** extension

To execute the file, **./<filename>.sh**

Common error: -bash: ./<filename>.sh: No such file or directory

You may need execute permission: **chmod +x <filename>.sh**

Rules for Variables

Variable names should represent or describe the data they contain

- Begin variable with alphabet or underscore character (_), followed by one or more alphanumeric or underscore characters
- Variables names are case-sensitive
- No spaces on either side of the equal sign when assigning value to variable

Shell scripting has keywords

- Should not be used as variable names
- They are reserved for writing syntax and logical flow of the program
- Examples: if, then, fi, for, while, do, done, switch, function, etc.

Environment Variables

Environment variables allow for customization and control of the command and system environment.

Current variables are seen using the `set` or `env` command.

```
HOSTTYPE=x86_64
LESSCLOSE=/usr/bin/lesspipe %s %s
LANG=C.UTF-8
USER=khalad
PWD=/home/khalad
HOME=/home/khalad
NAME=A4005069
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop
SHELL=/bin/bash
TERM=xterm-256color
SHLVL=1
LOGNAME=khalad
```

Important variables:

- `$PATH` – list of directories where commands/applications will be found
- `$HOME` – user home directory

Printing Variables

To process a variable, use double quotes with the "$" sign

```
echo $varName
echo "$varName"
echo "${varName}"
```

Use `unset` to delete a variable during the program execution

```
unset varName
```

Variables and Arrays

Variables that hold single value

```
gpa=3.9
course="DATA541"
```

An array is a collection of variables

```
students=("Adam" "David")
temperature=(23 22 18)
```

Printing Arrays

Array variables can also be echoed as a array with a default delimiter, but another way to echo arrays is put them in a loop and echo them as scalars

```
students=("Adam" "David")
```

Indexes have range between 0 to (n-1)

You can access an individual element from the array by using its index

```
echo ${students[0]}
```

Use [@] or [*] to print all elements of an array.

```
echo ${students[@]}
```

```
echo ${students[*]}
```

Array Operations

To print a slice of items in an array

```
${students[@]}                  #whole array
${students[@]:index:length}     #from index to index+length-1
${students[@]::length}          #from 0 to length-1 inclusive
${students[@]:index}            #from index to end of array
```

Array Operations

To add new items to the list

```
students=("Adam" "David")
students=(${students[*]} "Alan" "Hasan")
```

To remove an item at a given position

```
unset students[2]
```

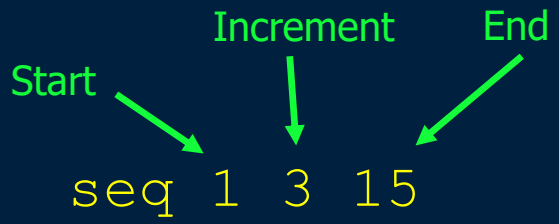
To reassign a value at a given position

```
students[2]="Khalad"
```

Creating Arrays

An Integer array can be created using the command "seq"

- needs a start and end position, along with increment size




```
x=$(seq 1 3 15)
echo $x

Output: 1 4 7 10 13
```

Variables and Array Question

Question: What is the output of the following code?

```
values=("T1" "T2" "T3" "T4" "T5")
echo ${values[@]:2:2}
```

A) `"T1" "T2"`

B) `"T2" "T3"`

C) `"T3" "T4"`

D) `"T4" "T5"`

E) `"T3"`

Operators and Expressions

Math operators:

+ addition	- Subtraction	% modulus	++ Increment
* Multiplication	/ Division	** Exponent	-- Decrement

Logical (boolean) operators

!	NOT	&&	AND	||	OR

Operators and Expressions

Arithmetic expansion and evaluation are done by `$((expression))`

Example:

```
echo "2 + 3 = " $((2+3))
```

Operators and Expressions

Arithmetic operations can be done with external programs e.g., `expr`

```
x=6
y=`expr $x + 4`
```

Note that space is required between operands

We can also perform bash arithmetic operations with `let` command

```
x=6
let y=x+4
echo $y
```

Floating Point Arithmetic

"bc" command is used for command line calculator. It is similar to basic calculator, print an expression and send it to built-in calculator

```
x=1.5
y=2.9
echo "$x/$y" | bc -l
```

Relational Operators

Shell also supports the following relational operators that are specific to numeric values

```
$a -lt $b     # $a < $b
$a -gt $b     # $a > $b
$a -le $b     # $a <= $b
$a -ge $b     # $a >= $b
$a -eq $b     # $a is equal to $b
$a -ne $b     # $a is not equal to $b
```

String Operations

There are many ways to perform string operations

Concatenation:

Using += append to variable

```
str1="Hello"
str2=" world"
str1+=$str2
```

Keep two string variables side by side

```
var1="Hello"
var2=" world"
echo $var1$var2
```

Or

```
echo "Hello" " world"
```

String Manipulation

Operation	Syntax
String Length	`${#string}` *** add echo for all
Extract a Substring	`${string:position}`
Extract substring from $string at $position	`${string:position:length}`
Remove shortest match of $substring from $string	`${string#substring}`
Remove longest match of $substring from $string	`${string##substring}`
Replace only first match	`${string/$pattern/$replacement}`
Replace all the matches	`${string//$pattern/$replacement}`

String Manipulation Example

Remove the shortest and the longest match of $substring from front of $string

shortest

longest

The shortest match: `echo ${str#a*a}` Output: `bc123abc`

The longest match: `echo ${str##a*a}` Output: `bc`

Replacement

```
stringZ=abcABC123ABCabc
echo ${stringZ/abc/xyz}  # xyzABC123ABCabc
                         # Replaces first match of 'abc' with 'xyz'.
echo ${stringZ//abc/xyz} # xyzABC123ABCxyz
                         # Replaces all matches of 'abc' with # 'xyz'
```

What happens if no $replacement string is supplied?

```
echo ${stringZ/abc}      # ABC123ABCabc
echo ${stringZ//abc}     # ABC123ABC
```

String Operation Question

Question: In the following example, which line of code need to be corrected to see the output "Item2"?

```
str1="Item1"          #line 1
str2="Item2"          #line 2
str1+=$str2           #line 3
res=${str1:4:4}       #line 4
echo $res             #line 5
```

A) 1 **B)** 2 **C)** 3 **D)** 4 **E)** 5

Getting User Input Via Keyboard

Syntax

```
read -p "Prompt" variable1 variable2 variable
```

Example:

```
read -p "Enter your name: " name
```

```
Enter your name:
```

```
echo "Welcome $name!!!"
```

Conditional Statement

If-then-else syntax allows logical decision making

Blocks of code can be branched to execute only when certain conditions are met

```
if [condition1 is true];
then
   <statements if condition1 is true>
else
    <statements if condition1 is false>
fi
```

Nested if statements are possible

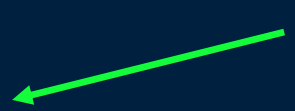
Conditional Statement

Example: Check if a file exists in a directory

```
filename="/mnt/c/DATA541/Lab1.docx"
if [ -f $filename ];
then
    echo "The file exists"
else
    echo "File doesn't exist"
fi
```

Conditional Statement

If-then-elif example




```
read -p "Enter a value: " myvar
if [ $myvar -gt 10 ]; then
    echo "Greater than 10"
elif [ $myvar -eq 10 ]; then
    echo "Equal to 10"
else
    echo "Less than 10"
fi
```

An alternative approach is to use (( ))

```
if (( $myvar > 10 )); then
```

If Statement Question

Question: Which is the correct code to check if a user's input (i.e., `input`) is between 50 and 100 (excluding 50 and 100)?

A)
```
if [ $input -lt 50 ] && [ $input -lt 100 ]; then
```

B)
```
if [ $input -gt 50 ] && [ $input -gt 100 ]; then
```

C)
```
if [ $input -gt 50 ] && [ $input -lt 100 ]; then
```

D)
```
if [ $input -lt 50 ] && [ $input -gt 100 ]; then
```

E)
```
if [ $input -eq 50 ] && [ $input -eq 100 ]; then
```

Try it: Using Conditional Statement

Question 1: Write a script that reads a number and prints out if the number is a positive or a negative number

Question 2: Write a script that reads two integer values via keyboard and print out their sum, difference, product, and quotient

Conditional Statement

A few rules to remember:

- You can combine conditions by using "&&" for "and" and " ||" for "or"
- Invert a condition by putting an "!" in front of it
- End a line with ";" before putting a new keyword like "then"
- Keep spaces between the brackets

```
read -p "Enter your CGPA: " myvar
if [ $(echo "$myvar>3.75" | bc -l) -eq 1 ]; then
    echo "Excellent CGPA"
else
    echo "Work Hard"
fi
```

Switch-case

The case statement is an alternative to multilevel if-then-else-fi

Example:

```
dow=$(date +"%a")
case $dow in
        Mon|Wed)
           echo "DATA541";;
        Tue|Thu)
           echo "DATA530";;
        Fri|Sat|Sun)
           echo "No class";;
        *) ;;
esac
```

Flow Control: For loop

Used to repeat a set of statements a number of times.

```
for var in item1 item2 ... itemN
    do
        command1
        ....
        commandN
    done
```

Flow Control: For loop

Examples:

```
for i in 1 3 5 7 9
do
  echo "Value: $i"
done
```

```
values=$(seq 1 2 9)
for i in  ${values}
do
  echo "Value: $i"
done
```

Examples:

```
for i in {1..9..2}
do
  echo "Value: $i"
done
```

```
for (( i=1; i<=9; i=i+2 ))
do
   echo "Values: $i"
done
```

More Example

Reading multiple .txt file with for loop (assuming we have multiple txt file in the same folder)

```
filename="*txt"
for file in $filename
do
 echo "Contents of $file"
 echo "---"
 cat "$file"
 echo
done
```

For Loop Question

Question: Which loops will execute exactly 10 times?

```
1) for i in {1..10}
2) for i in {1..10..1}
3) for i in {1...10..2}
4) for (( i=1; i<10; i=i+1 ))
```

A) 1, 2 **B) 2, 3** **C) 3,4** **D) 1** **E) 2**

Flow Control: While loop

The while statement also used to execute a list of commands repeatedly.

```
while [ condition ]
      do
            command1
            command2
            ....
            commandN
      done
```

Flow Control: While loop

Example:

```
n=1
sum=0
while [ $n -le 3 ]
do
        read -p "Enter number $n: " numb
        sum=$(( sum+numb ))
        n=$(( n+1 ))
done
echo "Sum: $sum"
```

While Loop Question

Question: What will this `while` loop do?

```
count=1
while (( $count <= 10 ))
do
        exponent=$(($count**$count))
        echo "Exponent: $exponent"
done
```

A) Print 10 lines

B) Print 9 lines

C) Print 8 lines

D) Print 0 lines

E) Cause an infinite loop

Nested Loops

A nested loop is a loop within a loop, an inner loop within the body of an outer one.

- The first pass of the outer loop triggers the inner loop, which executes to completion
- Then the second pass of the outer loop triggers the inner loop again.
- This repeats until the outer loop finishes.

```
for a in 1 2 3 4 5
do
    for b in 1 2 3 4 5
    do
    done
done
```

Nested Loops

```
# Beginning of outer loop.
for a in 1 2 3 4 5
do
  echo "Pass $a in outer loop."
  echo "----------------------"

  # Beginning of inner loop.
  for b in 10 20 30 40 50
  do
    echo "Pass $b in inner loop."
  done
  # End of inner loop.
done
```

Loop Control

The `break` and `continue` loop control commands correspond exactly to their counterparts in other programming languages.

- The `break` command terminates the loop (breaks out of it)
- `continue` causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

Continue example

```
LIMIT=5  # Upper limit
a=0
while [ $a -le $LIMIT ]
do
 a=$(( a+1 ))
 if [ $a -eq 2 ] || [ $a -eq 4 ]  # Excludes 2 and 4.
 then
   continue  # Skip rest of this particular loop iteration
 fi
 echo $a # This will not execute for 3 and 11.
done
```

Output:

1

3

5

6

Why?

break example

```
LIMIT=5  # Upper limit
a=0
while [ $a -le $LIMIT ]
do
 a=$(( a+1 ))
 if [ $a -eq 2 ] || [ $a -eq 4 ]  # Excludes 2 and 4.
 then
   break  # Skip entire rest of the loop
 fi
 echo $a # This will not execute for 3 and 11.
done
```

Output:

```
1
```

More break example

```
for outerloop in 1 2 3
do
  echo "Group $outerloop:   "
  for innerloop in 1 2 3 4 5
  do
    echo "$innerloop "
    if [ $innerloop -eq 2 ]
    then
      break
    fi
  done
  echo
done
```

Output:

Group 1:

1

2

Group 2:

1

2

Group 3:

1

2

Try it: Flow Control

Question 1: Write scripts to show the sum of integer numbers between 1 and 100.

Question 2: Write scripts to calculate and show the factorial of a given number.

Functions

A function as a small chunk of code that use for a certain task

Instead of writing the same code multiple times, you may write it once in a function then call that function when necessary

Create a function:

```
function_name(){
<command>
}
```

Example:

```
func1(){
  echo "I am inside func1"
}
func1
```

Functions

Within a function, the passing arguments can be accessible as $1, $2, etc.

```
add() {
    sum=`expr $1 + $2`
    echo "$1 + $2 = $sum"
}
add 10 20
```

Bash functions don't allow us to use return for sending data back to the main code

However, they send a return status indicates whether it succeeded or not.

Use the keyword `return` to indicate a return status

File access

Shell script can be also used to work with data files

Easy to access with shell commands

Common operations

- Reading data from a file
- Writing data to a file
- Appending data to a file

File Write

To redirect standard output to a file, the ">" character is used

To append to a file, the ">>" is used

```
file="readme.md"
echo "DATA541" > $file
echo "Description of the course" >> $file
echo "$file contains"
cat $file
```

File Read

ASCII/text files can be read line by line using shell script easily.

Common syntax across different bash versions

```
while read line; do
     COMMAND;
done < input_file
```

Example:

```
file="readme.md"
while read line; do
   echo "$line"
done < "$file"
```

Command Line Arguments

Command line arguments are optional data values that can be passed as input to the Shell script program as the program is run

- After the name of the program, place string or numeric values with spaces separating them
- Accessed them by $1, $2, $3 …

Command Line Arguments

Example:

```
echo "You passed: $*"
echo "There are $# arguments in total"
echo "The first argument is $1"
echo "The last argument is ${!#}"
```

Run:

```
./args.sh 10 20 30 40
```

Output:

```
You passed: 10 20 30 40
There are 4 arguments in total
The first argument is 10
The last argument is 40
```

Exit Codes

Shell scripts use exit codes when a Unix command returns control to its parent process

Success is traditionally represented with exit 0;

Failure is normally indicated with a non-zero exit-code. This value can indicate different reasons for failure. Example:

```
touch /home/khalad/course.txt
if [ $? -eq 0 ]; then
   echo "Successfully created the file."
else
   echo "Could not create the file"
fi
```

File and Command Line Arguments Question

Question: How many of the following statements are **FALSE?**

1) Command line arguments can be accessed by $1, $2, $3
2) In exit code, failure is normally indicated with a zero
3) Command line arguments are optional
4) ">>" is used to append to a file
5) `return` keyword is used to return values from a function

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

Pipes

Pipes allow to use two or more commands in a way that output of one command serves as input to the next

The symbol '|' denotes a pipe.

Order of execution is from left to right

```
touch file{1..9}.txt
ls
ls | head -n 4
```

```
file1.txt
file2.txt
file3.txt
file4.txt
```

```
ls | head -n 4 | tail -n 1
```

```
file4.txt
```

Pipes

`grep` command reads the file for a desired information and show present the result in a specified format

```
cat country.txt | grep "Ca"
```

Counting number of countries

```
cat country.txt | grep "Ca" | wc -l
```

A Linux command can be split across multiple lines by using the "\" character at the end

```
cat country.txt | grep \
"Ca" | wc -l
```

Try it: Scripting with I/O Redirection

Question: Using a terminal on your computer, write a script to performs these actions. Before creating the file, create a file called `country.txt` and add a few countries there.

Create a new file, called `myscript.sh`:

1) Write a command to sort `country.txt` and output as `sorted.txt`.
2) Write a command to output the word count the number of countries to `count.txt`.
3) Write commands to take `country.txt` and append its data three times into the file `output.txt`.
4) Use `grep` to search for "`e`" in `output.txt` and write results as file `search.txt`.
5) Output the contents of `sorted.txt`, `output.txt`, and `search.txt`.
6) Run your sh file.

List of useful commands

https://tldp.org/LDP/abs/html/part4.html

Objectives

- Use variables and arrays
- Use string operations and manipulation
- Apply conditional statements
- Looping with `for` and `while`
- Read and write input from/to files
- Use command line arguments and pipes


UBC
THE UNIVERSITY OF BRITISH COLUMBIA