CSE 391

Bash Scripting

Slides created by Josh Ervin. Based off slides made by Marty Stepp, Jessica Miller, Ruth Anderson, Brett Wortzman, and Zorah Fung

Course Evals

- First 10 minutes today is for filling out course evals
- Follow this link to the evals for our class:

https://uw.iasystem.org/survey/228742

(and thank you for all your hard work during this difficult time!)

Announcements

- Homework 9 is due on SUNDAY night
- End of quarter grades are due earlier in summer, normal Tuesday due date would be past the deadline
- You may have enough credits to pass already, check gradescope and the syllabus to be sure.

ROADMAP

- Introduction to the command line
- Input/output redirection, pipes
- More input/output redirection, tee, xargs
- Git: Fundamentals
- Git: Branches, merging, and remote repositories
- Regular expressions
- More regular expressions, sed
- Users and permissions
- Bash scripting

SHELL SCRIPTING

- A shell script is a file that executes a chain of shell commands in order.
- By convention, shell scripts use a `.sh` file extension
- In addition to many of the commands that we've learned (ls, cd, grep, etc), there are control structures and objects just like other programming languages (i.e. for loops, if/else statements, variables, arithmetic, etc).

RUNNING A SHELL SCRIPT

simple.sh

```
#!/bin/bash

echo “Printing wd”
pwd
echo “Printing contents”
ls
```

- Suppose we have the file `simple.sh` on the left.
- In order to run this file, we would type `./simple.sh` while inside of the same directory.
- Alternatively, we could add this file to a directory within our path.
- Our file must also be executable. A fast way to add execute privileges is to `chmod +x` the file.

RUNNING A SHELL SCRIPT

simple.sh

```
#!/bin/bash

echo “Printing wd”
pwd
echo “Printing contents”
ls
```

- The `#!` (called a shebang) At the top of the file tells the command line how to interpret your file (if you were writing a Python script, for example, you would replace `/bin/bash` with `/bin/python`)

ENVIRONMENTS

changedir.sh

```
#!/bin/bash

mkdir dir1
cd dir1
```

Console output

```
$ pwd
/home/user
$ ./changedir.sh
$ pwd
/home/user
```

- Running a shell script is almost equivalent to simply running the commands one after another on the command line.
- Any changes to the filesystem persist (i.e. making files, directories, etc), but changes to the shell environment do not change (i.e. working directories, variables, etc)

VARIABLES

var1.sh

```
#!/bin/bash

NAME=”Josh”
echo “Hi, my name is $name”
```

Console output

```
$ ./var1.sh
Hi, my name is Josh
$
```

- Variables are created using the = operator
- WARNING: There must not be spaces on either side of the equal sign, or else the assignment will not work correctly.
- Variables are referenced using the $ operator

VARIABLES

var2.sh

```
#!/bin/bash

name=”Josh”
echo ‘Hi, my name is $name’
```

Console output

```
$ ./var2.sh
Hi, my name is $name
$
```

- If you use single quotes, it will **not** expand any variables that you reference inside of it.

VARIABLES

var3.sh

```
#!/bin/bash

NAME=”Josh”
echo “Hi, my name is $name”
```

Console output

```
$ ./var3.sh
Hi, my name is
$
```

- Variable names are case sensitive
- **If the variable name you referenced is not found, bash does not throw an error.** Instead, the reference just returns the empty string.

VARIABLES

var4.sh

```
#!/bin/bash

name=Josh Ervin
echo “Hi, my name is $name”
```

Console output

```
$ ./var4.sh
line 4: Ervin: command not found
Hi, my name is
$
```

- It is good practice to wrap all your variables in quotes. If not, bash has trouble interpreting what you are writing (for multi-word strings) and will cause an error on that line.
- If there is an error on one line of the program, it does not cause the whole script to fail - just that one line.

VARIABLES

var5.sh

```
#!/bin/bash

contents=$(ls)
echo “The dir contains: $contents”
```

Console output

```
$ ls
var5.sh file.txt
$ ./var5.sh
The dir contains: var5.sh file.txt
$
```

- The output of running a command, such as `ls`, can be saved to a variable and referenced later.

COMMAND LINE ARGUMENTS

args.sh

```
#!/bin/bash

echo “The name of this script is $0”
echo “The first argument is $1”
echo “The second argument is $2”
echo “There are $# arguments”
echo “And their values are $@”
```

Console output

```
$ ./args.sh Foo Buzz Bar
The name of this script is args.sh
The first argument is Foo
The second argument is Buzz
There are 3 arguments
And their values are Foo Buzz Bar
$
```

- There are special variable names that apply to command line arguments:
  - $0 refers to the name of the script
  - $1, $2, $3, … refer to the argument in each position from left to right
  - $# refers to the number of arguments
  - $@ is a list of all the arugments

ARITHMETIC

math1.sh

```
#!/bin/bash

a=1
let b=”$a + 3”
echo $b
```

Console output

```
$ ./math1.sh
4
$
```

- The first method of performing arithmetic is using the let command, in which you assign a variable and the right side is a mathematical expression.
- WARNING: You must include quotes around the mathematical expression.

ARITHMETIC

math2.sh

```
#!/bin/bash

a=1
b=$(( $a + 3 ))
echo $b
```

Console output

```
$ ./math2.sh
4
$
```

- The other method of performing mathematical operations is using `$(( expr ))` where `expr` is some sort of mathematical expression
- WARNING: You must include spaces inside of the parenthesis.

ARITHMETIC

math2.sh

```
#!/bin/bash

a=1
b=$(( $a + 3 ))
echo $b
```

Console output

```
$ ./math2.sh
4
$
```

- Bash supports the following arithmetic operators
  - * multiplication
  - / integer division
  - + addition
  - subtraction

FOR LOOPS

loop1.sh

```
#!/bin/bash

for i in $(seq 1 4); do
     echo $i
done
```

Console output

```
$ ./loop1.sh
1
2
3
4
$
```

- You may use for loops to iterate over certain iterable structures.
- For example, to iterate over the numbers 1 through 4, you can use the `seq` command to generate a list of numbers and then loop over it.

FOR LOOPS

loop2.sh

```
#!/bin/bash

for file in $(ls); do
     echo $file
done
```

Console output

```
$ ./loop2.sh
file1.txt
file2.txt
loop.sh
$
```

- To iterate over all files in the current directory, you can loop over the output of `ls`

THINK

mystery.sh

```
#!/bin/bash

mkdir dir1
cd dir1
touch file1.txt
for file in $(ls); do
  echo $file
done
```

Console output

```
$ ls
./mystery.sh
$ ./mystery.sh
-> What gets printed here?
```

What would be the output of running `./mystery.sh` ?

IF STATEMENTS

if_math1.sh

```
#!/bin/bash

a=1
b=42
if [ $a -lt $b ]; then
     echo “a is less than b”
else
     echo “a is not less than b
fi
```

Console output

```
$ ./if_math1.sh
$ a is less than b
```

- You can use if statements just like in many other programming languages
- The true/false statement must be surrounded by [ ] **with spaces on either side**
- For arithmetic comparison
  - -gt : greater than
  - -lt : less than
  - -ge : greater than or equal to
  - -le : less than or equal to
  - -eq : equals
  - -ne : not equals

IF STATEMENTS

if_math2.sh

```
#!/bin/bash

a=1
b=1
if [ $a -lt $b ]; then
     echo “a is less than b”
elif [ $a -eq $b ]; then
     echo “a equals b”
else
     echo “a is not less than b
fi
```

Console output

```
$ ./if_math2.sh
$ a equals b
```

- You may also include else if conditions using the `elif` keyword.

IF STATEMENTS

Comparison operators	description
`if [ expr1 -a expr2 ]; then`	and
`if [ expr1 ] && [ expr2 ]; then`	and
`if [ expr1 -o expr 2 ]; then`	or
`if [ expr1 ] || [ expr2 ]; then`	or
`if [ ! expr 1 ]; then`	negation

```
if [ $a -lt 10 ] && [ $a -gt 5 ]; then
  echo “variable a is between 5 and 10”
fi
```

IF STATEMENTS

Comparison Operator	Description
`=, !=`	String operator comparison
`-z, -n`	Test if a string is empty (-z) or nonempty (-n)
`-f, -d`	Test if a file (-f) or a directory (-d) exists
`-r, -w, -x`	Test if a file exists and is readable (-r), writeable (-w) and executable (-x)

```
if [ -z $NAME ]; then
  echo ‘Variable $NAME exists’
fi
```

EXIT CODES

exit_code.sh

```
#!/bin/bash

rm file.txt
cat file.txt # this command fails!
echo $?
```

Console output

```
$ ./exit_code.sh
$ 1
$
```

- Whenever a program is run, it will return 0 if executed correctly or not 0 if it failed.
- The status of the previously run command can be accessed using the $? variable