Lecture 12: Bash Scripting Part II

- For Loops
- While Loops
- Command Substitution
- Example: File Name Analyzer
- Summary
- Bash Shell Debugging

Note: As always, you can find examples in `~eric.brauer/uli101`

For Loops

Consider doing the laundry.

You have a big pile of clean clothes back from the dryer, and you want to fold, hang, or otherwise put away each of those articles of clothing. The only way to handle this job is to pick each article of clothing in the pile, handle it however you want, and then move on to the next until there is no more clothing left in the pile.

Consider this example like a programmer. There are two variables that we are working with: the *pile*, and the *individual article of clothing*. We are performing actions on the article, not on the pile.

```
 for article in pile
do
   put away $article
done
```

The *for loop* allows us to repeat a set of actions for each element in our pile, whatever that may be. `article` here is a new variable, you can name it whatever you like.

```
cat -n scripts/SIMPLE-LOOP.sh
```

```
#!/bin/bash

echo
echo "We are entering the loop now."
echo "=============================="
echo
for thing in coffee tea juice
do
    # Any lines between 'do' and 'done' will be repeated for each 'thing.'
    echo "The variable 'thing' in this loop contains the value '$thing'."
    echo -n "Press any key to continue..."
    read
    echo
done

echo
echo "=============================="
echo "Now we are outside the loop."
echo "Try adding more things after 'juice'."
```

In this example we hard-coded three things in our 'pile.' But for loops are also useful when we don't know how many elements we need to deal with. We just know that there is lots of laundry in our pile, and we're going to keep working with it until it's all gone.

We have a useful variable that we can use when dealing with our "pile of arguments" from the user. It looks like this: `$@`. This variable contains all arguments passed to the script, separated by whitespace. We can use this variable in our example to handle each name individually.

```
cat -n scripts/SIMPLE-LOOP-WITH-ARGS.sh

     1  #!/bin/bash
     2
     3  if [ $# = 0 ]
     4  then
     5          echo "Run this script with arguments. See SIMPLE-ARGS.sh"
     6          exit 1
     7  fi
     8
     9  echo
    10  echo "Entering loop..."
    11  echo "Press Enter to step after each loop."
    12
    13  for thing in $@
    14  do
    15      echo "Current value of 'thing' is: $thing."
    16      everything_so_far=$everything_so_far" "$thing
    17      echo
    18      echo "Currently everything so far is: $everything_so_far."
    19      read
    20  done
    21
    22  echo
    23  echo "Exiting loop..."
    24  echo "Everything: $everything_so_far."
```

Keep in mind that there's nothing we can do with our "pile." We want to handle each element individually using the variable we created on line 13.

While Loops

These work exactly how you would expect them to. We will combine the `do ... done` block from our for loop with the `[ condition ]` from our if statement.

```
 counter=0
while [ $counter -lt 10 ]
do
    echo $counter
    counter=$(($counter+1))
done
```

Command Substitution

Let's say that you want to assign the output of a command to a new variable. For example, running `date -I'date'` will give you this output:

```
2019-07-29
```

(note: your output may vary from mine).

Now perhaps you want to use that output in a script. For example, maybe you want to put a timecode into a log file, or create files with today's date as the name. You essentially want to run:

```
touch $today
```

…where the value of `$today` is `2019-07-29`. This is where we can perform *command substitution*. Instead of setting a variable from user input, we can set it from a command:

```
today=$(date -I 'date')

echo $today
```

Everything inside `$(    )` is going to be sent into its own *non-interactive shell*. This means that a second shell is going to be opened, that command will run, and `$(    )` will be replaced with whatever the standard output of that command was.

We can use this in a lot of helpful ways. It allows us to integrate our shell commands in the script. Consider this:

```
cat -n scripts/SIMPLE-COM-SUB.sh
```

```
#!/bin/bash

today=$(date -I 'date')

mkdir $HOME/$today
```

Now we have the output of the `date` command to create a variable, and used the variable to create a directory. We could use this directory to back up our work, for example.

Example: File Name Analyzer

You've learned that 1. Linux filenames can contain all kinds of different characters, and 2. You really shouldn't.

This script is a utility that you can use to test directories. Each file in the directory will be tested to see if it contains some troublesome characters. If it does, then an error message will be printed.

```
cat -n scripts/filename-analyzer.sh
```

```
#!/bin/bash

```

```
targetD=$1
returnCode=0

if [ ! -d $targetD ]
then
    echo "Warning: target directory $targetD does not exist."
    echo "Usage: ./filename-analyzer.sh <name of a directory>"
    exit 2
else
    IFS=$'\n'
    for filename in $(ls $targetD)
    do
        if [ $(echo "$filename" | grep '[ !$%~#]') ]
        then
            echo "Warning: "$filename" contains problematic character. "
            returnCode=1
        fi
    done
    IFS=$' \t\n'
fi
exit $returnCode
```

Let's discuss things one at a time:

```
targetD=$1
returnCode=0
```

We are working with an argument from the user, but it's good practice to assign the value of `$1` to a named variable, since it'll be easier for a reader to understand what it represents.

`returnCode` is going to be used by us for the exit code. The goal of this script is to be useful not only for humans running the script, but to return useful exit codes if it's being called inside another script. I have decided that '1' will indicate the presence of at least one problematic filename, and '0' will represent no problematic filenames.

```
if [ ! -d $targetD ]
then
    echo "Warning: target directory $targetD does not exist. Enter an existing file"
    echo "Usage: ./filename-analyzer.sh <name of a directory>"
    exit 2
```

This is a file test. The user argument is tested to see if it is the name of an existing directory. If it is not, then an error message is printed, and the script exits. This warning also gets printed if the user has not entered any argument.

```
else
    IFS=$'\n'
    for filename in $(ls $targetD)
```

`IFS` is an environment variable. Usually it contains spaces, tabs and newlines. It's here so that spaces in filenames are treated as part of the filename, and not a separator. Feel free to comment out this line to see how the script changes.

This is a for loop connected to a command substitution. The command substitution is an `ls` command of the target directory. So basically we are returning a list of files, and we will look at each filename individually inside `do ... done`.

```
    do
        if [ $(echo "$filename" | grep '[ !$%~#]') ]
```

This is another command substitution.

- `echo "$filename"` will print the filename.
- `grep '[ !$%~#]'` if the filename contains any of these forbidden characters, the *output* of `grep` will be *something*.
- The `if [ ... ]` evaluates whatever's inside. If it's *nothing*, it will be false. If it's *something*, it will be true.

```
        then
            echo "Warning: "$filename" contains problematic character. "
            returnCode=1
        fi
```

We reach these lines when `grep` has returned *something*. We print a message that identifies the offending filename. We also change the value of `returnCode` from 0 to 1. This will indicate a failure.

```
    done
    IFS=$' \t\n'
fi
exit $returnCode
```

Finally, we have to close the loop. We will set the value of IFS back to its default. Then close the if statement, and use `exit` with the value of $returnCode (either 0 or 1).

With 23 lines, you can create a very powerful tool for system maintenance!

Still, there are a lot of ways we could improve this tool. We could choose to *invert* our `grep` and instead only search for acceptable characters. We could learn how to handle globs rather than directories, and so on.

Summary

- for *var* in *array*, do ... done
- $#: number of arguments
- $*: all arguments
- $( ): performs command substitution

Bash Shell Debugging

There are a couple of ways that we can modify Bash options to help us solve problems. To do this, we need to call bash explicitly so that we can give it some options.

Let's use our example `say-hi-to2.sh` to show how a *trace* might work.

```
bash -x say-hi-to2.sh
```

```
  + '[' 0 = 0 ']'
+ echo Usage: please include at least one person to greet.
Usage: please include at least one person to greet.
+ exit 1
```

Here we can see exactly how the interpreter is performing substitutions line-by-line. I didn't run this script with any arguments, so the value of `$#` is `0`. `[ 0 = 0 ]`, so we proceed into all the lines after `then`.

Feel free to use this tool on the scripts you are creating for the assignment, in order to understand how they are working.

Another tool you might want to consider is shellcheck. Shellcheck is a tool that you can install if you have your own Linux system, but it isn't installed by default on Matrix. Fortunately we can use it in the browser.

Shellcheck will tend to generate a lot of warnings that are outside the scope of our course, but useful when you start creating more complex scripts. It's a good to learn some best practices.