Table of contents

1. Scripting Recap
2. Bash Basics
3. Conditional Statements
4. Loops
Some Logistics

- All materials have been updated: `>` became `>>>`.
- Great job with HW1: only about 20 `git` mishaps I am aware of. Out of 200, that's stellar!
- Today is more scripting. The first bit was in `lec06`.
- `VIM` will be coming back soon when we hit `ssh`...review `lec06`.
- Lecture demos 7 and 8 are up.
  - `lec07` is just a transcript of what we did at the end.
  - `lec08` is definitely worth taking a look at...`sed` is very powerful.
Scripting Recap
Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined \textit{first}.
- We are doing \texttt{bash}. Use the proper Shebang (\texttt{#!/bin/bash}).
- Declaring variables: cannot have spaces on side of equals signs!
  - \texttt{Yes}: \texttt{FOO="value"}
  - \texttt{No}: \texttt{FOO = "value"}
- Dereference the value with the \$ symbol.
  >>> \texttt{echo "\$FOO"}
  - Note: for safety, always expand variables \textit{inside double} quotes.
  >>> \texttt{echo 'Singles joining'"\$FOO"' in doubles...'}
- Single quotes are the one ring to rule them all.
  - Things are read \textit{literally}, including special symbols.
  >>> \texttt{echo '\$USER'}
  - Refer to [3] for more.
When you need to execute a command and store it in a variable, you have two options:

- Surround it with backticks (`...cmd...`):
  ```bash
  >>> VAR=`echo value`
  ```
- Surround it with $(...cmd...):
  ```bash
  >>> VAR=$(echo value)
  ```

Both still work, but you should prefer $(...) to backticks, as backticks are deprecated.

Not all commands work out as you expect. If you are not getting the results you expect, print out the variable. A bad example:

```bash
#!/bin/bash
STATUS=$(echo "error string" > /dev/null)
echo "$STATUS"
```
Recall from **lec04** that commands have exit codes:

- Always execute:

  ```
  >>> cmd1; cmd2  # exec cmd1 first, then cmd2
  ```

- Execute conditioned upon exit code:

  ```
  >>> cmd1 && cmd2  # exec cmd2 only if cmd1 returned 0
  >>> cmd1 || cmd2  # exec cmd2 only if cmd1 returned NOT 0
  ```

- Kind of backwards, in terms of what means continue for *and*, but that was likely easier to implement since there is only one `0` and many `not 0`'s.

- Reference the exit code of the previous command with `$$`?
Bash Basics
The shell will expand arithmetic expressions that are encased in `$( ( expr ) )`

```bash
>>> echo $((2+3))  # standard addition
5
>>> echo $((2<3))  # less than: true is 1
1
>>> echo $((2>3))  # greater than: false is 0
0
>>> echo $((2/3))  # division: BASH IS ONLY INTEGERS!!!
0
>>> x=10
>>> echo $((x++))  # post increment: only for variables, does it AFTER...
10
>>> echo "$x"      # ...but see it did increment
11
>>> echo $((++x))  # pre increment: only for variables, does it BEFORE....
12
>>> echo "$x"      # ...only one increment took place
12
>>> sum=$((x+10))  # use variables like normal,
>>> echo "$sum"     # note: no quotes "$x" (it is a number)
22
```
• The Shebang does not need a space, but can have it if you want. The following all work:

```
#!/bin/bash
#! /bin/bash
#! /bin/bash
#! /bin/bash
#! /bin/bash
```

• Just needs whitespace, the `#!` is the *magic*. Just need:
  • The `#!` to be the very first two characters, and
  • the executable separated by whitespace *on the same line*.

• In bash, you use `#` to start a comment (line / end of line that will not execute).
• When you pass arguments to a bash script, you can access them in a few different ways:
  • $1, $2, ..., $10, $11: values of the first, second, etc arguments to the script.
    • If you do not have that many arguments, the variable value is just empty.
  • $0 is the name of the script.
  • $# is the number of arguments (argc in C).
  • $? is the exit code of the last program executed.
    • You can have your script set this with `exit <number>`, read the man page.
  • $$ is the current process identification number (PID).
  • $* expands $1 .. $n into one string.
    • $* \rightarrow "$1 $2 \ldots $n"
  • $@ expands $1 .. $n into individual strings.
    • $@ \rightarrow "$1" "$2" \ldots "$n"
Simple Examples

```bash
#!/bin/bash
# File: multiply.sh
echo $(( $1 * $2 ))  # print out arg1 * arg2

./multiply.sh 5 10

#!/bin/bash
# File: toLower.sh
tr '[A-Z]' '[a-z]' < $1 > $2  # read in arg1 and tr into arg2

./toLower.sh input_file output_file

#!/bin/bash
# File: expansion.sh
# note the use of single quotes to get a literal *
edo 'This is the *:'
for var in "$*"; do
  echo "Var: $var"
done
echo 'This is the @:'
for var in "$@"; do
  echo "Var: $var"
done

./expansion.sh hello there "billy bob"
```
If Conditionals

• If statements are structured just as you would expect...

    if [ CONDITION_1 ]
    then
        # statements
    elif [ CONDITION_2 ]
    then
        # statements
    else
        # statements
    fi # fi necessary

# The `then` is necessary...
# use a semicolon to shorten code
if [ CONDITION_1 ]; then
    # statements
elif [ CONDITION_2 ]; then
    # statements
else
    # statements
fi # fi necessary

• Double brackets [[ expr ]] allow for more features e.g. boolean operations. You generally should always use double brackets.

    if [[ CONDITION_1 ]] || [[ CONDITION_2 ]]; then
        # statements
    elif [[ CONDITION_3 ]] && [[ CONDITION_4 ]]; then
        # statements
    else
        # statements
    fi # fi necessary

• Note that you need spaces before and after the brackets!!!
Test Expressions

- Bash has a special set of commands that allow various checks.
- Numerical comparisons (often used with variables):
  - `n1 -eq n2` tests if \( n1 = n2 \).
  - `n1 -ne n2` tests if \( n1 \neq n2 \).
  - `n1 -lt n2` tests if \( n1 < n2 \).
  - `n1 -le n2` tests if \( n1 \leq n2 \).
  - `n1 -gt n2` tests if \( n1 > n2 \).
  - `n1 -ge n2` tests if \( n1 \geq n2 \).
- If either `n1` or `n2` are not a number, the test fails.
- String comparisons:
  - `s1 == s2` tests if `s1` and `s2` are identical.
  - `s1 != s2` tests if `s1` and `s2` are different.
  - Make sure you have spaces!
    - `s1==s2` will fail...
• If `path` is a string indicating a path, we can test its validity and attributes:
  • `-e path` tests if `path` exists.
  • `-f path` tests if `path` is a file.
  • `-d path` tests if `path` is a directory.
  • `-r path` tests if you have permission to read the file.
  • `-w path` tests if you have write permission.
  • `-x path` tests if you have execute permission.
  • `-s path` tests if the file is empty.
  • There are many of these, refer to [2] for more.
Loops
For Loops

```bash
for var in s1 s2 s3; do
    cmd1
    cmd2
done
```

```bash
for var in {000..22}; do
    cmd1
    cmd2
done
```

```bash
for (( i = 0; i < 10; i++ )); do
    cmd1
    cmd2
done
```
While Loops

```bash
while [[ condition ]]; do
    cmd1
    cmd2
done

FILE="filename.txt"
while read line; do
    cmd1
    cmd2
done < "$FILE"

FILE="filename.txt"
for line in $(cat "$FILE"); do # NEVER DO THIS
    cmd1
    cmd2
done
```
More on Loops

• For whatever reason, **bash** is one of the few languages that has an **until** loop:

```bash
#!/bin/bash
x=0
until [[ "$x" -eq 11 ]]; do
  echo "$x"
  (( x++ ))
done
```

• The **until** loop is exactly how it sounds: execute the loop body **until** the condition evaluates to **true**.
• So once `x` is **11**, the condition is false.
• This means that only `0..10` actually get printed.
• Lets get some practice!

https://github.com/cs2043-sp16/lecture-demos/tree/master/lec09

