09 - Bash Scripting II

CS 2043: Unix Tools and Scripting, Spring 2016 [1]

Stephen McDowell

February 17th, 2016

Cornell University

Table of contents

Some Logistics

- All materials have been updated: `>` became `>>>`.

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!

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`.

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`.

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.

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.

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.

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes:* `FOO="value"`

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

  - Note: for safety, always expand variables *inside* double quotes.

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

  - Note: for safety, always expand variables *inside* double quotes.

```
>>> echo 'Singles joining'"$FOO"' in doubles...'
```

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

  - Note: for safety, always expand variables *inside* double quotes.

```
>>> echo 'Singles joining'"$FOO"' in doubles...'
```

- Single quotes are the one ring to rule them all.

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

  - Note: for safety, always expand variables *inside* double quotes.

```
>>> echo 'Singles joining'"$FOO"' in doubles...'
```

- Single quotes are the one ring to rule them all.
  - Things are read *literally*, including special symbols.

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

  - Note: for safety, always expand variables *inside* double quotes.

```
>>> echo 'Singles joining'"$FOO"' in doubles...'
```

- Single quotes are the one ring to rule them all.
  - Things are read *literally*, including special symbols.

```
>>> echo '$USER'
```

Review I

- A script just executes from the top to the bottom.
  - Calling functions or using variables? They must be defined *first*.
- We are doing `bash`. Use the proper Shebang (`#!/bin/bash`).
- Declaring variables: cannot have spaces on side of equals signs!
  - *Yes*: `FOO="value"`
  - *No*: `FOO = "value"`
- Dereference the value with the `$` symbol.

```
>>> echo "$FOO"
```

  - Note: for safety, always expand variables *inside* double quotes.

```
>>> echo 'Singles joining'"$FOO"' in doubles...'
```

- Single quotes are the one ring to rule them all.
  - Things are read *literally*, including special symbols.

```
>>> echo '$USER'
```

  - Refer to [3] for more.

Review II

- When you need to execute a command and store it in a variable, you have two options:

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):

    ```
    >>> VAR=`echo value`
    ```

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):

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

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):

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

    ```
    >>> VAR=$(echo value)
    ```

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):

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

    ```
    >>> VAR=$(echo value)
    ```
  - Both still work, but you should prefer `$(...)` to backticks, as backticks are *deprecated*.

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):
    ```
    >>> VAR=`echo value`
    ```
  - Surround it with `$(...cmd...)`:
    ```
    >>> VAR=$(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:

Review II

- When you need to execute a command and store it in a variable, you have two options:
  - Surround it with backticks (`` `...cmd...` ``):
    ```
    >>> VAR=`echo value`
    ```
  - Surround it with `$(...cmd...)`:
    ```
    >>> VAR=$(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:

```
#!/bin/bash
STATUS=$(echo "error string" > /dev/null)
echo "$STATUS"
```

Remember the Exit Codes

Recall from `lec04` that commands have exit codes:

Remember the Exit Codes

Recall from `lec04` that commands have exit codes:

- Always execute:

Remember the Exit Codes

Recall from `lec04` that commands have exit codes:

- Always execute:

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

Remember the Exit Codes

Recall from `lec04` that commands have exit codes:

- Always execute:

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

- Execute conditioned upon exit code:

Remember the Exit Codes

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
```

Remember the Exit Codes

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.

Remember the Exit Codes

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

Arithmetic Expansion

- The shell will expand arithmetic expressions that are encased in `$(( expr ))`

Arithmetic Expansion

- The shell will expand arithmetic expressions that are encased in `$(( expr ))`

```
>>> 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           # set a variable
>>> echo $((x++))  # post increment: only for variables,
10                 # does it AFTER...
>>> echo "$x"      # ...but see it did increment
11
>>> echo $((++x))  # pre increment: only for variables,
12                 # does it BEFORE....
>>> echo "$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
```

Syntax Notes

- The Shebang does not need a space, but can have it if you want. The following all work:

Syntax Notes

- 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
```

Syntax Notes

- 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
```

- Just needs whitespace, the **#!** is the *magic*. Just need:

Syntax Notes

- 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
```

- Just needs whitespace, the **#!** is the *magic*. Just need:
  - The **#!** to be the very first two characters, and

Syntax Notes

- 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
```

- 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*.

Syntax Notes

- 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
```

- 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).

Passing Arguments to Scripts

- When you pass arguments to a bash script, you can access them in a few different ways:

Passing Arguments to Scripts

- 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.

Passing Arguments to Scripts

- 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*.

Passing Arguments to Scripts

- 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.

Passing Arguments to Scripts

- 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`).

Passing Arguments to Scripts

- 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.

Passing Arguments to Scripts

- 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.

Passing Arguments to Scripts

- 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).

Passing Arguments to Scripts

- 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.

Passing Arguments to Scripts

- 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.
    - `$*` $\Longrightarrow$ `"$1 $2 ... $n"`

Passing Arguments to Scripts

- 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.
    - `$*` $\Longrightarrow$ `"$1 $2 ... $n"`
  - `$@` expands `$1` .. `$n` into individual strings.

Passing Arguments to Scripts

- 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.
    - `$*` $\Longrightarrow$ `"$1 $2 ... $n"`
  - `$@` expands `$1` .. `$n` into individual strings.
    - `$@` $\Longrightarrow$ `"$1" "$2" ... "$n"`

Simple Examples

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

Simple Examples

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

```
./multiply.sh 5 10
```

Simple Examples

```
#!/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
```

Simple Examples

```
#!/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
```

Simple Examples

```
#!/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 *
echo 'This is the *:'
for var in "$*"; do
    echo "Var: $var"
done
echo 'This is the @:'
for var in "$@"; do
    echo "Var: $var"
done
```

Simple Examples

```
#!/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 *
echo '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"
```

Conditonal Statements

If Conditionals

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

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
```

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
```

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 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
```

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.

Test Expressions

- Bash has a special set of commands that allow various checks.
- Numerical comparisons (often used with variables):

Test Expressions

- Bash has a special set of commands that allow various checks.
- Numerical comparisons (often used with variables):
  - `n1 -eq n2` tests if $n1 = n2$.

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 $n1 \neq n2$.

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 $n1 < n2$.

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 $n1 \leq n2$.

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 $n1 > n2$.

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 $n1 \geq n2$.

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.

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:

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.

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.

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!

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*...

Path Testing

- If `path` is a string indicating a path, we can test its validity and attributes:

Path Testing

- If `path` is a string indicating a path, we can test its validity and attributes:
  - `-e path` tests if `path` exists.

Path Testing

- 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.

Path Testing

- 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.

Path Testing

- 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.

Path Testing

- 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.

Path Testing

- 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.

Path Testing

- 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.

Path Testing

- 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

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

For Loops

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

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

For Loops

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

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

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

While Loops

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

While Loops

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

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

While Loops

```
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:

More on Loops

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

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

More on Loops

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

```
#!/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**.

More on Loops

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

```
#!/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.

More on Loops

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

```
#!/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.

More on Loops

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

```
#!/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

References I

[1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.
Previous cornell cs 2043 course slides.

[2] TLDP.
Introduction to if.
http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html#sect_07_01_01.

[3] H. to Geek.
What's the difference between single and double quotes in the bash shell?
http://www.howtogeek.com/howto/29980/whats-the-difference-between-single-and-double-q

References II