06 – Wildcards, loops, and variables

CS 2043: Unix Tools and Scripting, Spring 2019 [1]

Matthew Milano
February 4, 2019

Cornell University
1. As always: Everybody! ssh to wash.cs.cornell.edu

2. Quiz time! Everybody! run `quiz-02-04-19`

3. Chaining Commands

4. Returning to scripts!

5. Conditional Statements

6. Loops

7. Bash Basics
As always: Everybody! ssh to wash.cs.cornell.edu
Quiz time! Everybody! run quiz-02-04-19
Chaining Commands
There are various *environment* variables defined for your shell.

They are almost always all capital letters.

You obtain their value by dereferencing them with a `$`.

```
$ echo $PWD       # present working directory
$ echo $OLDPWD    # print previous working directory
$ printenv       # print all environment variables
```

There are also *local* variables you can use / set.

Primary difference:

- *Environment* variables are available in your shell, *and* in scripts.
- *Local* variables are *only* available in your shell.

   “Shell” here just means “current terminal session.”
What is Defined?

• The environment:
  • `env`: displays all environment variables.
  • `unsetenv <var_name>`: remove an environment variable.
  • Create an environment variable*:
    • `export ENV_VAR_NAME="value"`
  • `export` is the most common. Exceptional explanation here.

• The local variables:
  • `set`: displays all shell / local variables.
  • `unset <var_name>`: remove a local shell variable.
  • Create a local variable*:
    1. `set local_var="value"
    2. `local_var="value"

* These only last for the current shell session; we will learn how to make them “permanent” soon.
Brief Example: Environment Variable Manipulation

# MY_ENV_VAR is not set yet, so nothing prints
$ echo "My env var is: $MY_ENV_VAR"
My env var is:

# Set the environment variable (can also use `export` in bash)
$ export MY_ENV_VAR="Lemming King"

# Now that we have set it, print it
$ echo "My env var is: $MY_ENV_VAR"
My env var is: Lemming King

# "Delete" with `unsetenv`. Print again, confirming it's gone
# Emphasis: there *is* an `env` after `unset`
$ unsetenv MY_ENV_VAR
$ echo "My env var is: $MY_ENV_VAR"
My env var is:
Brief Example: Local Variable Manipulation

```bash
# my_local_var is not set yet, so nothing prints
$ echo "My local var is: $my_local_var"
My local var is:

# Just declare it (can also use the `set` command)
$ my_local_var="King of the Lemmings"

# Now that we have set it, print it
$ echo "My local var is: $my_local_var"
My local var is: King of the Lemmings

# "Delete" with `unset`. Print again, confirming it's gone
# Emphasis: there is *not* an `env` after `unset`
$ unset my_local_var
$ echo "My local var is: $my_local_var"
My local var is:
```
Exit Codes

- When you execute commands, they have an “exit code”.
  - This is how you “signal” to others in the shell: through exit codes.
- The exit code of the last command executed is stored in ` $?`.
- There are various exit codes, here are a few examples:

```bash
$ super_awesome_command
bash: super_awesome_command: command not found...
$ echo $?  
127
$ echo "What is the exit code we want?"
What is the exit code we want?
$ echo $?  
0
```

- The success code we want is actually 0. Refer to [2].
- Remember `cat` with no args? You will have to `ctrl+c` to kill it, what would the exit code be?
• With exit codes, we can define some simple rules to chain commands together:

• Always execute:

\[
$ \text{cmd1; cmd2} \quad \# \text{exec cmd1 first, then cmd2}
\]

• Execute conditioned upon exit code of cmd1:

\[
$ \text{cmd1 && cmd2} \quad \# \text{exec cmd2 only if cmd1 returned } 0 \\
$ \text{cmd1 || cmd2} \quad \# \text{exec cmd2 only if cmd1 returned NOT } 0
\]

• Kind of backwards, in terms of what means continue for \textit{and}, but that was likely easier to implement since there is only one \texttt{0} and many \textit{not 0}'s.
Returning to scripts!
#!/usr/bin/env bash
# declare some variables
NAME="Sven Nevs"
MSK_ID=$(id -u)
# A simple if statement
if [[ $MSK_ID -eq 0 ]]; then
echo "Executing as root."
else
echo "Executing as normal user."
fi
# Expand variable inside string:
# Only because using _double_ quotes
echo "You are: $NAME"
# A simple for loop using a {} range
for n in {1..11}; do
    # String concatenation is easy!
    echo "$n is: ""$n"
    # Single quotes for literal $,
    # or use \$ in double quotes
done

• Use the shebang:
  #!/usr/bin/env bash
• Declare variables...
  • ...no spaces!
• Use variables...
  • ...dereference with $
• Execute commands...
  • $(command ...)
  • `command ...`
• If statements and loops.
• NEVER use aliases in bash scripts. EVER.
Storing command output

- Two options for storing output of command in variable:
  - Surround it with backticks `...cmd...`:
    ```
    var="`echo hello world`"
    ```
  - Surround it with $(...cmd...):
    ```
    var="$(echo hello world)"
    ```
  - Prefer $(...), backticks are deprecated.

- Print debugging with **echo** can be very helpful, a bad example:
  ```bash
  #!/usr/bin/env bash
  # status will be empty because we redirected `stdout`
  # from `echo` to `/dev/null`!
  status="$(echo "error string" > /dev/null)"
  echo "status is: '$status'"
  ```
Conditional Statements
If Conditionals

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

- Double brackets (**bash only!**) `[[ expr ]]` allow for more features e.g., boolean operations.
- **both [ and [[ are actually commands!**

```bash
if [[ CONDITION_1 ]] || [[ CONDITION_2 ]]; then
  # statements
fi
```

- **elif and else clauses allowed, not required.**
BE VERY CAREFUL WITH SPACES!

- Spaces on both the outside and the inside necessary!

```bash
# bash: syntax error near unexpected token `then`
if[[ 0 -eq 0 ]]; then echo "Hiya"; fi

# bash: [[0 command not found...
if [[0 -eq 0 ]]; then echo "Hiya"; fi

# bash: syntax error in conditional expression:
# unexpected token `;`
# bash: syntax error near `;`
if [[ 0 -eq 0 ]]; then echo "Hiya"; fi

# This has spaces after if, and before brackets (works)!
if [[ 0 -eq 0 ]]; then echo "Hiya"; fi
```
Test Expressions

- [ ] and [ ] have a special set of commands that allow 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...
  - For strings in particular, use double quotes!
    - If string has spaces and no double quotes used, it will fail.
Path Testing

- Test if `/some/path` exists: `-e /some/path`
- Test if `/some/path` is a file: `-f /some/path`
- Test if `/some/path` is a directory: `-d /some/path`
- Test if `/some/path` can be read: `-r /some/path`
- Test if `/some/path` can be written to: `-w /some/path`
- Test if `/some/path` can be executed: `-x /some/path`
- Test if `/some/path` is an empty file: `-s /some/path`
  - Many more of these, refer to [3] for more.
Path Testing Example

```bash
#!/usr/bin/env bash
path="/tmp"
if [[ -e "$path" ]]; then
    echo "Path '$path' exists."
    if [[ -f "$path" ]]; then
        echo "--> Path '$path' is a file."
    elif [[ -d "$path" ]]; then
        echo "--> Path '$path' is a directory."
    fi
else
    echo "Path '$path' does not exist."
fi

• Output from script:

Path '/tmp' exists.
--> Path '/tmp' is a directory.
```
Warning About Saving Exit Codes

- If you need to work with the exit code more than once...
- ...**always** save it!
- Simply put, get in the habit of **always** saving `cmd_exit=$?`
- Then use `$cmd_exit` in your test expressions.
Loops
# Delineate by spaces, loop:
# s1, then s2, then s3, then s4
for var in s1 s2 s3 s4; do
echo "Var: $var"
done

# Brace expansion:
# 00, 01, ..., 11
for var in {00..11}; do
echo "Var: $var"
done

# "Traditional" for Loop:
# 0, 1, ..., 11
for (( i = 0; i <= 11; ++i )); do
echo "i: $i"
done

# Output:
# Var: s1
# Var: s2
# Var: s3
# Var: s4
# Output:
# Var: 00
# Var: 01
# Var: ...
# Var: 11
# Output:
# i: 0
# i: 1
# i: ...
# i: 11
Bash Basics
Arithmetic Expansion

- Arithmetic expressions are encased in `$( ( expr ) )`

```bash
$ echo $(( 2 + 3 ))  # standard addition
5
$ echo $(( 2 < 3 ))  # less than: true is 1
1
$ 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" needed in
22  # arithmetic $(( expressions ))
```
Warning on Arithmetic Expansions

- Exponentiation example: $x \; ** \; y \rightarrow x^y$

  ```bash
  # bash: syntax error near unexpected token `('  
  $ x=((( 2 ** 3 )))
  # Execute ls: I have only one file 'multiply.sh'
  $ x="((( 2 ** 3 )))"
  $ echo $x
  (( 2 multiply.sh 3 ))
  # That $ before the (( expr )) is NECESSARY!
  $ x=$((( 2 ** 3 )))
  $ echo $x
  8
  ```

- Leading $ in $$(( \; expr \; ))$$ is syntactically required.
  - Just like $x$ to read value
  - or var="$((...cmd...))"
• 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
    • If 3 arguments given, $4, $5, ... higher are 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 `man exit`).
      • No explicit call to `exit` same as `exit 0` (aka success).
  • $$ is the current process identification number (PID).
  • $* expands $1 .. $n into one string.
    • $* ⇒ "$1 $2 ... $n" (one string)
  • $@ expands $1 .. $n into individual strings.
    • $@ ⇒ "$1" "$2" ... "$n" (n strings)
Demo files!

- /course/cs2043/demos/06-demos/multiply.sh
- /course/cs2043/demos/06-demos/toLower.sh
- /course/cs2043/demos/06-demos/expansion.sh
back to loops
While Loops

```bash
s="s" # Test expression comparison
while [[ "$s" != "ssss" ]]; do
    echo "$s" # prepend s until
    s="ss$s" # target length reached
done

x=0 # Arithmetic comparison
while (( x <= 11 )); do
    echo "x: $x"
    (( ++x ))
done

# Loop through lines in file
file="filename.txt"
while read -r line; do
    echo "Line: $line"
done < "$file"
```

- Print every line in a POSIX-compliant file.
- See full demo at end of lecture!
- (see more_demos.txt)
Until Loops

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

```bash
x=0
until (( x == 4 )); do
    echo "x: $x"
    (( x++ ))
done
```

# Output:
# x: 0
# x: 1
# x: 2
# x: 3

- The `until` loop is exactly how it sounds: execute the loop body *until* the condition evaluates to **true**.
- So once `x` is **4**, `(( x == 4 ))` is **true**, loop stops.
  - Loop body not executed when `x == 4`, so `x: 4` not printed.
  - Like `for` and `while`, can also use *test expressions*:

```bash
until [[ $x -eq 4 ]]; do
```

23
See lecture demo on looping through files.
References

[1] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.
