

06 – Wildcards, loops, and variables

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

Matthew Milano

February 4, 2019

Cornell University

Table of Contents

1. Chaining Commands
2. Returning to scripts!
3. Conditional Statements
4. Loops
5. Bash Basics
6. back to loops
7. Customizing your Terminal

Chaining Commands

Your Environment and Variables

- 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*:
 1. **env** ENV_VAR_NAME="value"
 2. **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)
```

```
$ env 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

```
# 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 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:

```
$ 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?

Executing Multiple Commands in a Row

- With exit codes, we can define some simple rules to chain commands together:
- Always execute:

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

- Execute conditioned upon exit code of **cmd1**:

```
$ 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.

Returning to scripts!

Bash Scripting at a Glance

```
#!/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:

```
#!/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

- 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 semicolon to shorten code
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.

```
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: 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

- 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*...
 - 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

```
#!/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!
 - A contrived example.
- Simply put, get in the habit of **always** saving `cmd_exit=$?`
- Then use `$cmd_exit` in your *test* expressions.

Loops

For 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 enclosed 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" needed in
22                    # arithmetic $(( expressions ))
```

Warning on Arithmetic Expansions

- Exponentiation example: $x ** y \Rightarrow x^y$

```
# 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...))"`

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
 - 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 .. $n` into one string.
 - `$*` \Rightarrow `"$1 $2 ... $n"` (one string)
 - `$@` expands `$1 .. $n` into individual strings.
 - `$@` \Rightarrow `"$1" "$2" ... "$n"` (n strings)

See demo file `multiply.sh`.

See [demo file toLower.sh](#).

See demo file `expansion.sh`.

back to loops

While Loops

```
s="s" # Test expression comparison
while [[ "$s" != "ssss" ]]; do
    echo "$s" # prepend s until
    s="s$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"
```

```
# Output:
# s
# ss
# sss
# ssss
# Output:
# x: 0
# x: 1
# x: ...
# x: 11
```

- Print every line in a POSIX-compliant file.
- See [full demo](#) at end of lecture!

Until Loops

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

```
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*:

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

Looping Through Files

See [lecture demo on looping through files](#).

Customizing your Terminal

What is it and Why?

- You will spend **a lot** of time in your terminal.
- It's worth spending a little time to configure it how you want.
- Customizations allow you to be
 1. More effective.
 2. Perform common operations more quickly.
 3. Make your terminal appear more comfortable *for you*.
 4. A super all-star-hacker-pro with l33t skillz.
- Think of it this way: it's like buying a new house. Paint the walls, build a tool shed, meet your neighbors, throw some parties. Why buy it if you weren't going to make it yours?
 - Why use the default terminal just because it came that way?
 - COME ON YOU CAN TOTALLY DO BETTER!

What are Dotfiles?

- “Dotfiles” change, add, or enhance existing functionality.
 - The files reside in your home (~) directory.
 - They are hidden files: their names start with a `.`
- Some common dotfiles you’ll hear about:

<code>~/.bashrc</code>	Controls bash terminal behavior*
<code>~/.bash_profile</code>	Controls bash environment variables*
<code>~/.profile</code>	Controls shell environment variables*
<code>~/.vimrc</code>	Controls the behavior of vim
<code>~/.gitconfig</code>	Controls the behavior of git
<code>~/.tmux.conf</code>	Controls the behavior of tmux (covered later)

- There are **many** possible dotfiles to customize.
- We will focus on configuring **vim** and our shell (**bash**).

* What these *do* depends on what **you** write in them! [See lecture demo.](#)

The Source of All Things

- So we now know a little bit about how a script is structured.
- It just executes from the top to the bottom.
- The shebang says how to run it. But...

Execute **source** in Current Shell

source <filename> [**arguments**]

- Executing script **B** from script **A** runs **B** in a *subshell*.
- Sourcing script **B** from script **A** executes in *current shell*.
 - If script **B** **exits**, then script **A** **exits**!
- Think of it like copy-pasting **B** into **A** at the line where **source B** is written in **A**.
- Just like **#include <header.h>** in **C** if you know it.
- Fundamental to the initial shell setup process:
 - All dotfiles related to your **shell** are *sourced*.

What Happens When

- There is a **lot** going on with dotfiles; no “standard” protocol.
- What happens when depends on:
 1. Your operating system.
 2. The shell you are using.
 3. For graphical logins, what your desktop / window manager is.
- There is an important difference between types of shells:
 - There is a “login” shell, and a “interactive” shell.
 - “Login” shell: takes place *once*, when you login.
 - `~/.profile`, `~/.bash_profile`, `~/.zprofile`, depending on what your shell is.
 - “Interactive” shell: takes *every time* you spawn a new shell.
 - E.g. `ctrl+shift+n` on Linux, `cmd+n` on Mac.
 - Inherits all actions that took place at *login*.
 - `~/.bashrc`, `~/.zshrc` depending on what your shell is.

Login Actions: Precursor

- There is even still an important distinction:
 - A graphical login (logging in through the GUI).
 - A login shell (disabled GUI, or used **ssh** or something).
- Graphical logins:
 - I will not cover this. There is **way** too much going on.
 - Depends on what your GUI (Gnome, KDE, etc) is.
 - A **fantastic** explanation in [4].
 - Hey! Look around the rest of the site!
 - Lots of other *great* information available!!!
- Login shells:
 - For simplicity, assume that when you login through your GUI, it triggers a login shell to be called.
 - This is mostly true, but not exactly.
 - Discussion to come: Bourne shells (**bash**, **ksh**, ...) vs **zsh**
 - Only because Bourne shells and **zsh** are “incompatible”.

Login Shells

- Where do the environment variables like **\$PATH** come from?
- For Bourne Shells:
 1. System level configuration files are sourced. Same for all users.
 - The file **/etc/profile** is sourced.
 - Do **NOT** edit this file directly. It sources *anything* found in **/etc/profile.d/*.sh**. Put additional resources there.
 - This is where **PATH** among many other variables is getting set!
 2. User-level configuration files are sourced (if found).
 - **bash** looks for **~/.bash_profile** first. If it sees it, it sources it.
 - Only if **bash** does not find **~/.bash_profile**, it looks for **~/.bash_login** next and then **~/.profile** last.
 - **ksh**, on the other hand, only looks for **~/.profile**.
- For **zsh**, the same pattern occurs:
 1. System level configuration: **/etc/zprofile**.
 - Typically, it *emulates* **ksh** and sources **/etc/profile**!
 2. Look for **~/.zprofile**.

Know Your Shell

- `$SHELL` reports *your* default shell (`echo $SHELL`).
- How do I know what my shell looks for and in what order?
 - `man <shell>` and search for **INVOCATION** as well as **FILES**.
 - Or cruise the Arch Wiki – they're great! E.g. [Arch on zsh](#).

Change your Login Shell

```
chsh -s /absolute/path/to/new/shell username
```

- GNU and BSD `chsh` are slightly different, **read the man page!**
- Example usage to change `$SHELL` for `username`:
- `$ sudo chsh -s /usr/local/bin/bash username`
 - Above example specific to macOS users who did `brew install bash`
 - `brew` installs the newer `bash` to `/usr/local/bin/bash`
 - macOS cannot ship Bash 4 or later (GPL v3 license).
- **Warning:** do **not** change the `$SHELL` of the `root` user!
- Typically, `chsh` will modify `/etc/passwd`
 - `grep` your `username` and read last field.

Interactive Shells

- Your environment is already setup and ready to go now that you have logged in.
- Now do the lightweight configurations, put in your **rc** file.
 - The `~/.bashrc` for **bash**
 - The `~/.kshrc` for **ksh**
 - The `~/.zshrc` for **zsh**
- Things you put in these files:
 - Shell specific **aliases**, **functions**, etc.
- Things you **never** do:
 - **source** `~/.bash_profile` from `~/.bashrc` for example.
 - It goes the other way: `~/.bash_profile` sources `~/.bashrc`
 - Initial *login* shell is when ***profile** get sourced.
 - The `~/.bashrc` is **not** sourced on login automatically.
 - Only if **you** do it (almost every distribution does this by default).

Aliases

Creating Aliases

`alias <new-name> <old-name>`

- Aliases `new-name` to be `old-name`, e.g. `alias ..='cd ..'`
 - Can now type `..` to go up one directory.
- Should not ever be used in scripts.
 - Disabled by default, battle to use them — **very** bad practice.
 - I don't have your aliases, so now I can't run your script.
- Usually stored in `~/.<shell>rc` file, though `~/.<shell>_aliases` is slowly gaining traction.
 - Make sure you `source ~/.<shell>_aliases` from `~/.<shell>rc` or else they won't be available!!!
 - E.g. `bash: ~/.bashrc` sources `~/.bash_aliases`, or
 - `zsh: ~/.zshrc` sources `~/.zsh_aliases`

Modifying your Terminal Prompt

- The **\$PS1** variable controls what shows up when you type in your terminal.
 - In **zsh** this is **\$PROMPT**.
- List of all options [here](#).
- Common: **export PS1="\u@\h:\w> "**
 - **usr@hostname:current/working/directory>**
- Try changing your **\$PS1** using **export** right now to see how you can modify it.
- Play with colors after, since they are tedious to type in the format needed.

Storing Customizations

- There are many such places that people put things, but generally speaking...
- Your **bashrc** should have things like aliases and functions. Limit the **export** calls to just things related to coloring the terminal.
- Your **bash_profile** should contain any special environment variables you need to define.
 - Typically when you are exporting things like **\$PATH** or **\$LD_LIBRARY_PATH** for something you have installed on your own.
- You should source your **bash_profile** from your **profile**, and you should source your **bashrc** from your **bash_profile**.

Customize!!!

References

- [1] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.
- [2] The Linux Documentation Project. *Exit Codes with Special Meanings*. 2017. URL: <http://tldp.org/LDP/abs/html/exitcodes.html>.
- [3] The Linux Documentation Project. *Introduction to If*. 2017. URL: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html#sect_07_01_01.
- [4] Greg Woledge. *Configuring your login sessions with dot files*. 2015. URL: <http://mywiki.woledge.org/DotFiles>.