

CS571 Advanced Programming Techniques

Bash Scripting

Today's agenda

- Select answers to A2
- Some more useful bash commands
- Bash shell scripting
- Online tutorial on bash scripting

Select Answers to A2

- Q8: Specify a command that permit the owner and the group members to read and write the file while removing all privileges from anyone else.

Select Answers to A2

- Q8: Specify a command that permit the owner and the group members to read and write the file while removing all privileges from anyone else.

u & g – give rw, do we keep x the way it was?

o – no r, no w, no x

chmod ug+rw, o-rwx file (keeps x the way it was for ug)

chmod 660 (removes x for all)

chmod 770 (adds x to ug)

Select Answers to A2

- Q10: Specify a single line command that gives the owner and the group permission to execute the file myFile, while giving the owner sole permission to read and write the file. Remove permission of others to execute.

Select Answers to A2

- Q10: Specify a single line command that gives the owner and the group permission to execute the file myFile, while giving the owner sole permission to read and write the file. Remove permission of others to execute.

u – give r,w,x

g – give x, not r or w

o – no permissions

chmod 710

chmod u=rxw,g=x,o= myFile

Select Answers to A2

- Q11: Specify a single line command that gives the owner and the group permission to execute the file myFile, while giving the owner sole permission to read and write the file. Do not change any other permissions, i.e., do not remove permission from others to execute, if they already have that permission.

u – give r,w,x

g – give x, not r, not w

o – not r, not w (if o has x, o keeps x)

chmod 710 (this won't work because o loses x if o has x)

chmod u=rwx,g=x,o-rw myFile

Bash Scripting

The printf command

- The command syntax

```
printf format [arguments ...]
```

- prints its optional arguments under the control of the format, a string which contains three types of objects:
 - plain characters, which are simply copied to standard output
 - character escape sequences which are converted and copied to the standard output, e.g., `\t` (tab), `\n` (newline), `\a` (bell), etc.
 - and format specifications, each of which causes printing of the next successive argument, e.g., `\s` (string), `\c` (character), etc.

```
$ printf "hi "  
$ printf "hi\n"  
$ printf "hi\a\a\a\a"  
$ printf "hi %s\n" "Hello"
```

```
Prints hi without a newline  
Prints hi and then a newline  
Prints hi and rings the bell 3 times  
Prints hi Hello and a newline
```

Bash Command Execution

Bash commands can be executed one at a time or they can be joined together in several ways

1. Sequenced
2. Grouped
3. Subshell Group
4. Conditional

1. Sequenced Commands

- Commands to be executed serially
- No direct relationship between them
- Commands can be separated by a newline or ;

```
cmd1 ; cmd2 ; cmd 3
```

```
$ echo a ; javac test.java
```

2. Grouped commands

```
{ cmd1 ; cmd2 ; cmd2 ; }
```

- Sequences can be grouped using { }
- List must be terminated by a ;
- Runs in the context of the current shell
- Useful for redirecting I/O
- Return value is the status of the last command executed
- Spaces are important to the right of { and to the left of }

```
$ echo a ; echo b ; echo c > out
a
b
$ cat out
c
```

```
$ { echo a ; echo b ; echo c; } > out
$ cat out
a
b
c
```

3. Grouped commands for subshell

```
( cmd1 ; cmd2 ; cmd2 )
```

- Sequences grouped with ()
- Also handy for redirecting I/O
- Runs in a subshell
- Can be run in the background
- No changes persist
- Return value is the status of the last command executed

```
$ y=30 ; ( y=20 ; echo $y ) ; echo $y
20
30
```

- No spaces on either side of the = symbol
 - `y = 30` won't work but `y=30` will

Assignment: No spaces before or after =

There should be no spaces before or after = when assigning a value

So this is correct

```
a=b
```

while this is not correct (note the spaces)

```
a = b
```

4. Conditional Commands

- Operators `&&` and `||`
- Conditional execution depends on the return value of the command on the left
- This value available to the caller (parent shell)
 - Look at special variable `$?` for return status of last command
- Value on `[0, 255]`
 - Zero (0) signals success; is true
 - We enumerate errors (failure), starting at 1
- `&&` and `||` have the same precedence, associate left-to-right

Conditional Execution Operators

`cmd1 && cmd2`

- `cmd1` is executed first
- If `cmd1` succeeds, then `cmd2` is executed

`cmd1 || cmd2`

- `cmd1` is executed first
- If `cmd1` fails, then `cmd2` executed

```
$ cp file1 file2 && echo "Copy succeeded"
Copy succeeded
$ cp no_such_file file2 2> /dev/null || echo "Copy failed"
Copy failed
```


Bash scripting

What is a shell?

- A program that interprets your requests to run other programs
- A shell is a high-level programming language
- Most common Unix shells:
 - Bourne shell (sh)
 - C shell (csh - tcsh)
 - Korn shell (ksh)
 - Bourne-again shell (bash)
- In this course we focus on bash shell (`bash`)

What is a script?

- A sequence of bash commands
- A bash script is a program
 - Stored as a text file
 - Interpreted by the bash shell
 - Made up of
 - Variables
 - Control structures

Why write scripts?

- Convenience
- Sequences of performed operations that are performed often can be placed in a script, executed as a single command
- Shell provides access to many useful utilities

Shell scripts vs Java-like languages

- Shell scripts are generally not as well-suited to large tasks
 - they run more slowly
 - are more resource-intensive
 - do not give the programmer the same degree of control over resources
- Languages such as Java and C/C++ allow for much more structured programs

Shell Scripts

- Text files that contain one or more shell commands
- The first line is always identifying the interpreter (the shell) that will execute the script

`#!/bin/bash (sha-bang)`

or

`#!/bin/sh`

- Except on line 1, # indicates a comment

```
% cat welcome
#!/bin/bash
echo 'Hello World!'
```

Shell scripts must be executable

```
% welcome
```

```
welcome: Permission denied.
```

```
% chmod 744 welcome
```

```
% ls -l welcome
```

```
-rwxr--r-- 1 dv35 ...
```

```
% welcome
```


```
welcome: command not found
```

```
% ./welcome
```

```
Hello World!
```



command will run if
the current directory
(the . directory) is in
the \$PATH



tells the shell to run
the command from
the . directory

Just like a command

```
% welcome > greet_them
```

```
% cat greet_them
```

```
Hello World!
```


What's in a shell script?

Bash Scripts are text files that include one or more shell commands. In addition, they can have

1. Shell Script Variables
2. Control structures

Shell Script Variables

Shell Script Variables

There are five possible types of shell script variables

1. Command-line arguments
2. Process-related variables
3. Environment variables
4. Shell variables
5. User-defined variables

1. Command Line Arguments

`$0` is the name of the script

`$1` is the first command-line argument

`$2` is the second command-line argument

...

`$#` is the number of arguments

2. Process-related variables

\$?	The exit status of the last command 0 – successful execution of last command Non-zero – something went wrong
\$#	The number of arguments
\$*	All arguments
\$@	All arguments (individually quoted)
\${n}	The n-th positional argument
\$\$	The process ID (pid) of the shell

Redirection Tricks

- Want to run a command to check its exit status and ignore the output?

```
diff f1 f2 > /dev/null
```

- Want to ignore both standard error and standard output?

```
diff f1 f2 >& /dev/null
```

In Unix, `/dev/null` is a device that discards all data written to it.

3. Environmental Variables

- Contain information about the system
- Available in all shells
- Examples: \$USER, \$HOME, \$PATH
- To display your environment variables, type

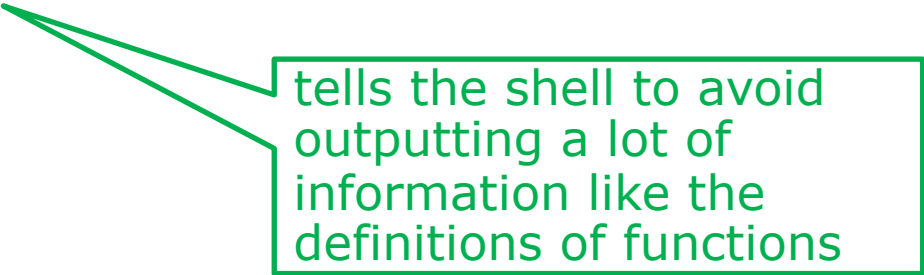
`printenv`

4. Shell Variables

- Used to tailor the current shell
- A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.
- Examples: `cwd`, `prompt`
- To display your shell variables, type

```
set -o posix
```

```
set
```



tells the shell to avoid outputting a lot of information like the definitions of functions

5. User-Defined Variables

- Variable name: combination of letters, numbers, and underscore character (_) that do not start with a number
- Avoid existing commands and shell/environment variables
- Assignment:

`name=value`

- No space around the equal sign!

Variables

- To use a variable: `$varname`
- Operator `$` tells the shell to substitute the value of the variable name

Control Structures

Control Structures

- We have branching:

`if`

`if-else`

`if-elif-else`

`case`

- And loops:

`while`

`until`

`for`

`select`

- And they all need conditions / tests

Conditions

There are many and often confusing ways to test for a condition

`test` and `[]` Provides string, numeric, and file tests

`[[]]` Similar to `[]`, but gentler syntax

`let` and `(())` Provides numeric tests and arithmetic

1. Using test

1. General form

```
test expr
```

2. Example

```
test $name = "Dimitra"
```

Spaces here are important to the left and right of =

```
if test $name = "Dimitra"
then
    echo 'Hello Dimitra!'
fi
```

2. Using []

1. General form

```
[ expr ]
```

2. Example

```
[ $name = "Dimitra" ]
```

Spaces are important to the left and right of =

Spaces here are important to the right of [and to the left of]

```
if [ $name = "Dimitra" ]  
then  
    echo 'Hello Dimitra!'  
fi
```

Warning – Be careful with those spaces!

- These are ok

```
[ $a = $b ]
```

```
[ $a=$b ]
```

- These are not ok

```
[ $a = $b ] (common mistake)
```

```
[ $a = $b ]
```

```
[ $a =$b ]
```

```
[ $a= $b ]
```

```
[ $a=$b ]
```

```
[ $a = $b ]
```


[] and File Conditions

<code>-a file</code>	True if <code>file</code> exists
<code>-e file</code>	True if <code>file</code> exists
<code>-f file</code>	True if <code>file</code> is an regular file
<code>-c file</code>	True if <code>file</code> is an character file
<code>-d file</code>	True if <code>file</code> is a directory
<code>! -d file</code>	True if <code>file</code> is not a directory

```
if [ -e $file ]  
then  
    echo "file exists";  
fi
```

[] and File Conditions

<code>-r file</code>	True if file <code>file</code> is readable
<code>-w file</code>	True if <code>file</code> is writable
<code>-x file</code>	True if <code>file</code> is executable
<code>-s file</code>	True if size of <code>file</code> is more than 0

```
if [ -r $file ]  
then  
    echo "file is readable";  
fi
```

[] and Numeric Conditions

Bash condition	Java Condition	Python Condition
n1 -eq n2	n1 == n2	n1 == n2
n1 -lt n2	n1 < n2	n1 < n2
n1 -gt n2	n1 > n2	n1 > n2
n1 -ne n2	n1 != n2	n1 != n2
n1 -le n2	n1 <= n2	n1 <= n2
n1 -ge n2	n1 >= n2	n1 >= n2

```
if [ $a -lt $b ]  
then  
    echo "I found a to be less than b";  
fi
```

NOTE:
you can't use <, >
inside [] to compare
numbers

WARNING

Use this

```
[ 1 -lt 2 ]
```

Not this

```
[1 -lt 2 ]
```

```
[ 1 -lt 2]
```

```
[ 1 < 2 ] - error
```

```
[ 13 < 2 ] - error
```

```
[ 1 \< 2 ] (this is syntactically ok but it compares strings TRUE)
```

```
[ 13 \< 2 ] (this is syntactically ok but it compares strings TRUE)
```

```
[ 43 \< 2 ] (this is syntactically ok but it compares strings FALSE)
```

3. Using `[[]]`

- Supports all the same tests as `[]`
- Is a built-in, so, syntax is gentler
 - Shell metacharacters `<` , `>` , etc., don't need to be escaped
 - Shell knows it's in a test

Again don't use `[[]]` for arithmetic tests using `>`, `<`, `..`

Similar to `[]`, `[[]]` does string comparisons

4. let

1. Using the `let` command

```
let arithmetic-expr
```

2. Example

```
let "X = 1 + 1"
```

```
let "X=X + 1"
```

```
let "X = $X+1"
```

```
echo $X
```

Spaces here are not important!

5. Using (())

- Treats values in double parameters as integers
- Performs arithmetic test
- Spaces don't matter

```
x=5
if (( x > 7 ))
then
    echo "$x is greater than 7";
fi
```

Variables – what is the difference?

`let a=1+1` variable creation with simple arithmetic expansion

`a=$((1+1))` arithmetic expansion, simple arithmetic; the
result of the expression replaces the expression

`a=$[1+1]` old syntax, don't use

`a=$(expr 1 + 1)` calls the Unix `expr` command

if

if tests

then

cmds;

fi

Also

if tests; **then** cmds; **fi**

if

if tests; then cmds; fi

- tests is executed
- If the exit status is 0 (success), cmds is executed

```
if grep Waldo * &> /dev/null ; then # std out/err go to bit bucket
echo "Found Waldo!"
fi
```

```
if [[ -d "$paris" && -r "$paris" ]] ; then
echo "I see $paris"!'
fi
```

```
if (( cats > 3 )) ; then
echo "Too many cats"
echo "People will talk"
fi
```

if-else

```
if tests
then
    cmds;
else
    cmds;
fi
```

if-else

```
if tests; then cmds; else cmds; fi
```

```
if grep Waldo * &> /dev/null ; then
echo "Found Waldo!"
else
echo "Waldo's a slippery one"
fi
```

```
if [[ -d "$paris" && -r "$paris" ]] ; then
echo "I see $paris"'"!'
else
echo "Might be on the wrong continent"
fi
```

```
if (( cats > 3 )) ; then
echo "Too many cats"
echo "People will talk"
else
echo "You might yet be sane"
fi
```

if-elif-else

if tests

then

cmds;

elif

cmds;

...

else

cmds;

fi

if-elif-else

if tests; then cmds; elif cmds; else cmds; fi

```
read grade
if (( grade >= 90 )) ; then
echo "A"
elif (( grade >= 80 )) ; then
echo "B"
elif (( grade >= 70 )) ; then
echo "C"
elif (( grade >= 60 )) ; then
echo "D"
else
echo "F"
fi
```

case

```
case word in  
pattern1) cmds;;  
pattern2 | pattern3) cmds;;  
...  
patternN) cmds;;  
'*' ) cmds;; #otherwise  
esac
```

Patterns can contain wildcards

It uses patterns not regular expressions

case

```
case word in {pattern } cmds ;;} esac
```

- **Selectively execute** `cmds` if `word` matches the corresponding `pattern`
- Commands are separated by `;`
- Cases are separated by `;;`

```
#!/bin/bash

case $1 in
  n ) DRY_RUN=1 ;;
  x ) ECHO=1 ;;
  \? | h | H ) echo "Use option n or option x" ; exit 1 ;;
  ?) echo "Unkown character" ;;
esac
```


while Loop

```
while condition
do
    command(s)
done
```

while Loop

```
while tests; do cmds; done
```

- `tests` is executed
- If the exit status is 0 (success), `cmds` is executed
- Execution returns back to `tests` , start again

```
i=0
while (( i<=12 )) ; do
echo $i
(( i+=1 ))
done
```

```
cat list | while read f ; do
# Assume list contains one filename per line
stat "$f"
done
```

until Loops

until condition

do

command(s)

done

for Loops

for variable **in** list

do

 command(s)

done

- variable is a user variable
- list is a sequence of strings separated by spaces

for loop

```
for name [in list]; do cmds ; done
```

- Executes `cmds` for each member in `list`
- "\$@" used if list isn't there

```
for i in a b c ; do
> echo $i
> done
a
b
c
```

```
for id in $(cat userlist) ; do
# assumes no spaces in userIDs
echo "Mailing $id..."
mail -s "Good subject" "$id"@someschool.edu < msg
done
```

for loop

- bash has a C/Java-like for loop:

```
for (( i=0; i<3; ++i )) ; do
> echo $i
> done
0
1
2
```

```
for (( i=12; i>0; i-=4 )) ; do
> echo $i
> done
12
8
4
```

select

```
select name [in list]; do  
    cmds ;  
done
```

select

```
select name [in list]; do cmds ; done
```

- Much like the `for` loop
- Displays enumerated menu of `list`
- Puts user's choice in `name`

```
$ select resp in "This" "That" "Quit" ; do  
> echo "You chose $resp"  
> [ "$resp" == Quit ] && {echo `bye!` ; break ; }  
> done  
1) This  
2) That  
3) Quit  
#? 2  
You chose That  
#? 3  
You chose Quit  
bye!
```


break and continue

- Interrupt loops (`for`, `while`, `until`)
- `break` jumps to the statement after the nearest done statement
 - terminates execution of the current loop
- `continue` jumps to the nearest done statement
 - brings execution back to the top of the loop

Loops – continue, break

- `break` exits a loop
- `continue` short-circuits the loop, resumes at the next iteration of the loop

```
$ for i in {1..42} ; do
> (( i%2 == 0 )) && continue
> (( i%9 == 0 )) && break
> echo $i
> done
1
3
5
7
```

Reading User Input

- Syntax: `read varname`
 - No dollar sign
- Reads from standard input
- Waits for the user to enter something followed by `<RETURN>`
- Stores what is read in user variable
- To use the input: `echo $varname`
 - Needs dollar sign

Reading User Input

- More than one variable may be specified
- Each word will be stored in separate variable
- If not enough variables for words, the last variable stores the rest of the line

More on command-line arguments

- \$1, \$2, ... normally store command line arguments
- Their values can be changed using the `set` command

```
set newarg1 newarg2 ...
```

All command-line arguments

- Both `$@` and `$*` get substituted by all the command line arguments
- They are different when double-quoted
- `"$@"` expands such that each argument is quoted as a separate string
- `"$*"` expands such that all arguments are quoted as a single string

Quoting Issues

- What if I want to output a dollar sign?
- Two ways to prevent variable substitution:

```
echo '$dir'
```

```
echo \ $dir
```
- Note: `echo "$dir"` is the same as `echo $dir`

User Variables and Quotes

- If value contains no space, no need to use quotes:
`dir=/usr/include/`
- Unless you want to protect the literal
- If value contains one or more spaces:
- Use single quotes for NO variable substitution
 - A dollar sign is a dollar sign
- Use double quotes for variable substitution
- A dollar sign followed by a variable name

Back Quotes

- Enclosing a command invocation in back quotes ([the character usually to the left of 1](#)) results in the whole invocation substituted by the output of the command

```
% dateVar=`date`
```

```
% echo $dateVar
```

```
Mon 16 Sep 2019 10:29:26 EDT
```

Arithmetic Operations Using `expr`

- The shell is not intended for numerical work
- However, the `expr` utility may be used for simple arithmetic operations on integers

```
sum=`expr $1 + $2`
```

- Note: spaces are required around the operator `+` (but not allowed around the equal sign)

Shell Script Functions

- Syntax:

```
function_name()  
{  
    command(s)  
}
```

- Allows for structured shell scripts

Going a little deeper
More refined definitions
More examples

Tests for branches and loops

We need tests for branches and loops

- We've already seen the return value of commands
 - If the return value is Zero (0), the command was successful
 - If the return value is non-zero, there was some failure
 - Can be negated using ! before
- There are special utilities and bash built-ins to provide various tests

`test, []` Provides string, numeric, and file tests

`[[]]` Similar to `[]`, but gentler syntax

`let, (())` Provides numeric tests and arithmetic

test or []

A condition in a script is designated in one of two equivalent ways:

1. Using the test command

```
test $name = "Dimitra"
```

2. Using the square bracket notation

```
[ $name = "Dimitra" ]
```

Spaces here are important!

[] – Tests

[expr]

- Built-in into bash
- Note, the spaces around the [] are necessary
- Provides:
 - String tests
 - File tests
 - Numeric tests
 - Logical operators

[] – String Tests

We have the normal binary, relational operators

< = != >

- e.g.,

```
[ $a = $b ]
```

```
[ $a \< $b ]
```

```
[ $a \> $b ]
```

```
[ $a != $b ]
```

Whoops! < is a shell metacharacter. Needs to be escaped

[] – String tests

We have unary tests for strings

-z True if string is empty

-n True if string is not empty

e.g,

```
[ -z "$1" ]
```

Checks to see if the first argument is empty

[] – File tests

Many unary tests for files. Here are a few

- `-e file` True if file exists
- `-d file` True if file is a directory
- `-f file` True if file is a regular file
- `-L file` True if file is a symbolic link
- `-r file` True if file is readable by you
- `-w file` True if file is writable by you
- `-x file` True if file is executable by you
- `-O file` True if file is effectively owned by you

See `man test` or `help test` for more

[] – File tests

There are some binary operators for files:

`f1 -nt f2` True if `f1` is newer than `f2`

`f1 -ot f2` True if `f1` is older than `f2`

`f1 -ef f2` True if `f1` is a hard link to `f2` (they are the same file)

e.g.,

`[-e "$log"]` to see if the log file exists

`[-r "$input"]` to see if the input file is readable

`["$input" -nt "$log"]` to see if the input file is newer than the log file

[] – Arithmetic tests

- We have different relational operators for arithmetic
 - All parameter values are just strings
 - Shell can't tell from context which comparison is meant

`-lt -le -eq -ne -ge -gt`

e.g.

```
[ 13 -lt 2 ]
```

[] – Logical Operators

`! expr`

NOT - True when `expr` is false, false otherwise

`exp1 -a exp2`

AND - True when both `exp1` and `exp2` are true, false otherwise

`exp1 -o exp2`

OR - False when both `exp1` and `exp2` are false, true otherwise

[[]]

- Supports all the same tests as []
- Is a built-in, so, syntax is gentler
 - Shell metacharacters < , > , etc., don't need to be escaped
 - Shell knows it's in a test/condition

[[]] – New Features

- Familiar logical operators ! && ||
- == and = are equivalent
- == and != treat the right operand as a pattern

e.g.,

```
[[ abcde.f == a*e.? ]]
```

- New operator, =~, treats the right operand as an extended regular expression

```
[[ abcde.f =~ a.*e\..? ]]
```

(()) – let, relational operators

- `let` is a bash built-in

```
let "myVar=2"
```

- bash provides syntactic sugar, `((x == 13))` works!
- Treats values stored in parameters as integers
 - Only does integer arithmetic (division)
- Allows you to evaluate relational expressions
 - Same logical operators `<` `<=` `==` `!=` `>=` `>`

E.g.,

```
(( x > 7 ))
```

```
(( x!=0 && y/x >= 6 ))
```


(()) – let, arithmetic operators

let can be used to evaluate arithmetic exceptions

- Arithmetic: `**` `*` `/` `%` `+` `-`
- Bit-wise: `~` `<<` `>>` `^` `&` `|`
- Pre- and post-fix increment/decrement: `++` `--`
- A Java-like ternary operator: `?:`
- Assignment (`=`), and the usual operator/assignment operators: `+=` `-=` `&=`, etc.

(()) - examples

```
$ x=13
```

```
$ echo $(( x+15 ))
```

```
28
```

```
$ echo $x
```

```
13
```

```
$ (( y = x*4 ))
```

```
$ echo $y
```

```
52
```

```
$ (( y-=1 ))
```

```
$ echo $y
```

```
51
```

```
$ echo $((x>>2))
```

```
3
```

{x..y} – Brace Expansion

{x..y}

- Generates sequences in a natural way

```
echo {5..13} # no spaces around the { and } symbols  
5 6 7 8 9 10 11 12 13
```

```
echo {a..g}  
a b c d e f g
```

- Brace expansion will pad numbers on the left

```
for i in {0..5} ; do echo -n "$i " ; done  
0 1 2 3 4 5
```

- This is quite handy in loops:

```
for i in {0..5} ; do  
  \rm proc${i}.log  
done
```

bash functions

```
function name {body}
```

```
function name() {body}
```

- Executed in the same environment
- Arguments to function are handled the same as arguments to a script
- Can be called recursively
- Built-in `return rv` can be used in a function, to return execution (and optional status `rv`) to caller

Function Examples

```
function hello
{
echo "hello $1"
if [[ -n "$2" && "$2" -gt 1 ]] ; then
hello $1 $(( $2 - 1 ))
fi
}
```

Called as a script from the shell prompt the output will be

```
$ hello Dimitra 3
hello Dimitra
hello Dimitra
hello Dimitra
```

Local variables in functions

```
local {var}
```

- Defines variable(s) local to function
- Won't step on caller's environment

```
function hello {  
  local USER='Elmer Fudd'  
  FOO='Hunting Wabbit'  
  echo "Hello, $USER, you are $FOO"  
}
```

```
$ FOO='Baking Cookies'  
$ echo $USER  
dv35  
$ hello  
Hello, Elmer Fudd, you are Hunting Wabbit  
$ echo $FOO  
Hunting Wabbit  
$ echo $USER  
dv35
```

What's Next

- Assignment 4 due February 4 at 11:59pm
- Give yourself a bit more time for assignment 4
- Try the examples and do the exercises from the following tutorial

[Link to Bash Tutorial](#)

- Next week there will be in-class tutorial exercises, bring your computer