Bash

EE 156 - Spring 2023

Introduction to Bash

- Bash stands for **B**ourne **A**gain **S**hell
  - Bash is a shell program in which a user can run commands, programs, and scripts
  - Bash is an open source GNU project
- Bash is similar to other programming languages
  - Includes variables, loops, conditional statements, positional parameters, arithmetics, functions, strings, etc
- Bash commands can be run directly from the command line or from a .sh file
- Other shell programs exists
  - Examples include ZSH, Bourne Shell, CSH, etc.
  - Bash is the default shell for most UNIX based operating systems

Getting Help

- There are many sources of Bash documentation
  - Google, stack overflow, etc.
  - man pages
    - Unix terminals come with an included manual for shell commands
    - Example usage: man ls
  - Office hours!

Bash Commands

Bash - clear

- clear clears the contents of the terminal window
- Usage: **clear**

Bash - pwd

- pwd, or “print working directory”, prints the path to the current working directory
- Usage: **pwd -[option(s)]**
    - Example: pwd - prints the path to the current working directory in the shell window

Bash - cd

- cd is used to change the working directory (folder) inside of a terminal
- Usage: **cd [path to new directory]**
  - The path can be an absolute path or a relative path
  - “.” represents the path to the current directory and “..” represents the path to the parent directory
    - Example: cd . - does nothing
    - Example: cd .. - changes the working directory to the parent directory
    - Example: cd ../new_dir - changes the working directory to new_dir, which is also a child of the parent directory

Bash - ls

- ls lists the contents of the current working directory
- Usage: **ls -[option(s)] [directory paths]**
  - Common options include -l and -a
    - -l : use a long list format
    - -a : do not ignore files starting with “.”
  - Example: ls -la - lists all files and directories in the current directory using long list formatting and does not ignore files starting with “.”
  - Example: ls * - lists all files and directories within the children of the current directory

Bash - touch

- touch creates an empty file at the specified path. If a file with the same name exists at the specified path, touch updates the last modified time of the file
- Usage: **touch -[option(s)] [path to file]**
  - Example: touch example.txt - creates “example.txt” in the current directory
  - Example: touch ../example.txt - creates “example.txt” in the parent directory

Bash - rm

Bash - mkdir

- mkdir makes a new directory at the specified path
- Usage: **mkdir -[option(s)] [path to new directory]**
  - Common options include -p and -v
    - -p : make parent directories as needed
    - -v : print out a message for each directory created
  - Example: mkdir new_dir - makes a new directory called new_dir that is a child of the current directory
  - Example: mkdir -p dir_1/dir_2/dir_3 - makes 3 new directories, dir_3 is a child of dir_2, which is a child of dir_1, which is a child of the current working directory

Bash - rmdir

Bash - cp

- cp copies files and directories
- Usage: **cp -[option(s)] [source path] [destination path]**
  - Common options are -r
    - -r : recursively copy directories
  - Example: cp original.txt new.txt - copies the contents of original.txt to new.txt. If new.txt already exists, its contents will be overwritten
  - Example: cp -r original_dir ../new_dir - copies the contents of original_dir (files and child directories) to ../new_dir

Bash - mv

- mv moves (renames) files and directories
- Usage: **mv -[option(s)] [source path] [destination path]**
  - Common options are -r
    - -r : recursively moves directories
  - Example: mv original.txt new.txt - moves the contents of original.txt to new.txt. If new.txt already exists, its contents will be overwritten
  - Example: cp -r original_dir ../new_dir - moves the contents of original_dir (files and child directories) to ../new_dir

Bash - rm

- rm removes files from a directory
- Usage: **rm -[option(s)] [path to file(s)]**
  - Common options include -r and -f (use both with extreme caution!)
    - -r : recursive remove (used to remove directories)
    - -f : force remove
  - Example: rm f1.txt f2.txt - removes ‘f1.txt’ and ‘f2.txt’
  - Example: rm *.txt - removes all ‘.txt’ files from the current directory
  - Example: rm -r child_dir/ - removes ‘child_dir’, and all files and directories within ‘child_dir’, from the current directory

Bash - rmdir

- rmdir removes empty directories from the filesystem
- Usage: **rmdir -[option(s)] [path to directories(s)]**
  - Common options include -p
    - -p : each of the directory argument is treated as a pathname of which all components will be removed, if they are already empty, starting from the last component
  - Example: rmdir child_dir/ - removes ‘child_dir’ if it is empty
  - Example: rmdir -p child_dir/grandchild_dir - tries to remove ‘grandchild_dir’, if it is empty, then tries to remove ‘child_dir’ if it is empty

Bash - echo

- echo prints a string
- Usage: **echo -[option(s)] [string]**
    - Example: echo “hello” - prints “hello”

Bash - cat

- cat is used to concatenate files and print the concatenation
- Usage: **cat -[option(s)] [path to file(s)]**
  - Common options include -n
    - -n : numbers all output lines
  - Example: cat f1.txt - prints the contents of f1.txt
  - Example: cat f1.txt - f2.txt - concatenates f1.txt with whatever is read from the standard input and f2.txt. Use Ctrl-D to exit the standard input

Bash - head

- head prints the beginning of a specified file. By default the number of lines printed is 10
- Usage: **head -[option(s)] [path to file]**
  - Common options include -n
    - -n (-)NUM: used to specify the number of lines to print. If a "-" is included, the last NUM lines will be omitted
  - Example: head -n 20 f1.txt - prints the first 20 lines of f1.txt
  - Example: head -n -50 f1.txt - prints all but the last 50 lines of f1.txt

Bash - tail

- tail prints the end of a specified file. The last 10 lines are printed by default.
- Usage: **tail -[option(s)] [path to file]**
  - Common options include -n
    - -n (+)NUM: prints the last NUM lines. If a “+” is included, the output starts at line number NUM
  - Example: tail -n 20 f1.txt - prints the last 20 lines of f1.txt
  - Example: tail -n +50 f1.txt - prints the contents of f1.txt from line 50 on

Bash - chmod

- chmod is used to change the mode bits, or permissions, of a file
- Usage: **chmod [option(s)] [permission(s)] [path to file]**
  - Permissions include read (r), write (w), and execute (x)
  - Permissions can be added (+) and removed (-)
  - Example: chmod +x test.py - changes permissions of test.py to make the file executable
  - Example: chmod -rw test.py - removes read and write permissions of test.py

Bash - find

- find searches for files in a directory hierarchy
- Usage: **find -[option(s)] [expression]**
  - Common options include -name, -type, and -exec
    - -name : the name of the file or directory to look for
    - -type [f/d] : specifies to look for files (f) or directories (d)
    - -exec : executes a command against every item in the result of the find command
  - Example: find . -type f -name *.txt - starting from the current directory, lists all files ending with .txt
  - Example: find . -type d -name my_dir -exec rm -r {} + - starting from the current directory, returns all directories with name my_dir. For each of these directories, executes the command rm -r
    - The “{}” is used as a placeholder for each item returned by the find command.
    - The “+” terminates the executed command.

Bash - less

- less is used to view the contents of a file without any risk of editing the file
- Usage: **less -[option(s)] [path to file]**
    - Example: less f1.txt - brings up the contents of f1.txt in the terminal

Bash - grep

- grep is used to search for text patterns in a file or stream. All lines matching the pattern will be returned
- grep can be used with regex
- Usage: **grep -[option(s)] [pattern] [path to file]**
  - Common options include - E, -n, -i, and -c
    - -E : allows for extended regex pattern matching
    - -n : includes line numbers in the output
    - -i : performs case insensitive matching
    - -c : only a count of matching lines is written to output
  - Example: grep ‘a’ test.txt - returns lines containing an word with “a” in test.txt

Bash - sed

- sed is used to edit files without the use of a text editor
- sed can be used with regex
- Usage: **sed -[option(s)] 's/[search pattern]/[replace string]/[instance]' [path to file]**
  - Common options include -E
    - -E : Allows for extended regex pattern matching
  - Example: sed 's/unix/linux' f.txt - replaces first instance of 'unix' with 'linux' in f.txt
  - Example: sed 's/unix/linux/2' f.txt - replaces second instance of 'unix' with 'linux' in f.txt
  - Example: sed 's/unix/linux/g' f.txt - replaces all instance of 'unix' with 'linux' in f.txt
  - Example: sed 's/unix//g' f.txt - deletes all instances of 'unix' in f.txt

Bash - awk

- awk is used for manipulating data and generating reports. awk allows for an action to be taken if a text pattern is found
- Usage: **awk -[option(s)] '[selection criteria] {[action]}' [path to input file]**
  - Common options include -f
    - -f [path to input file] : use the path provided after -f as the input file instead of the first command line arg
  - Examples: https://www.geeksforgeeks.org/awk-command-unixlinux-examples/

Output Redirection

Output Redirection - Piping Outputs

- Outputs of bash commands are written to the standard output by default
- Outputs of bash commands can be used as the input to different commands using the pipe (|) operator
- Piping is commonly used with the grep command
- Usage: **[command 1] | [command 2] | [command 3] | ...**
    - Example: head -n 7 f.txt | tail -n 5 - selects the first 7 lines of f.txt (through head) and then selects the last 5 lines from the output (lines 3-7) of f.txt (using tail)
    - Example: head f.txt | grep “hello” - searches for patterns matching the string “hello” in the first 10 lines of f.txt

Output Redirection - Overwriting files

- The standard output can be sent to a file using >
- The contents of the file will be overwritten
  - If the file does not exist, it will be created
- Usage: **[command] > [path to file]**
  - Example: echo “Hello, World!” > hello.txt - hello.txt will now only contain “Hello, world!”
  - Example: ls -l > directory_contents.txt - directory_contents.txt will now only contain the output of ls -l

Output Redirection - Appending to files

- The standard output can be sent to a file using >>
- The contents of the file will NOT be overwritten
  - If the file does not exist, it will be created
- Usage: **[command] >> [path to file]**
  - Example: echo “Hello, World!” > hello.txt - hello.txt will have “Hello, World!” appended to it
  - Example: ls -l > directory_contents.txt - directory_contents.txt have the output of ls -l appended to it

Pattern Matching with Regex

Introduction to Regex

- Regex, or regular expressions, are used for string pattern matching
- Regex can be used in bash to find a string pattern in a file, or to filter the output of shell commands
- https://regex101.com/ - Great resource for testing regex
- Regex can be used with many commands, but all examples are shown with the grep command

Regex - Literal Matches

- Literal matches do a search for an exact match of a character string
- Literal matches can be for a single character or a sequence of characters
- Example: grep a f.txt - match all lines containing a word with the character “a” in f.txt
- Example: grep “hello 123” f.txt - match all lines containing the string “hello 123” in f.txt

Regex - Anchor Matches

- Anchor matches are used to search for patterns at the beginning and end of a string
- ^ is used to search for patterns at a beginning of the string
  - Example: grep ^hello f.txt - match all lines starting with “hello” in f.txt
- $ is used to search for patterns at the end of a string
  - Example: grep “goodbye user”$ f.txt - match all lines ending with “goodbye user” in f.txt
- Using both ^ and $ together can be used to search for lines comprised of a specific string
  - Example: grep ^”hello my name is Bharat”$ f.txt - match all lines only containing “hello my name is Bharat”
  - Example: grep -n ^$ - match all empty lines and lists their line numbers

Regex - Placeholders

- Placeholders are used to match any character
- . is used as the placeholder character
    - Example: grep c.t f1.txt - match all lines with words that match the pattern “c followed by any character followed by t” in f1.txt

Regex - Bracket Expressions

- Bracket expressions allow matching multiple characters or a character range at a position
- **[expression]** is used for bracket expressions
  - Example: grep [ae]nd f1.txt - match all lines that contain “and” or “end” in f1.txt
- Using **^** within bracket expressions allows for the exclusion of the characters
  - Example: grep [^ae]nd f1.txt - match all lines that contain a 3 character string ending with “nd”, but not “and” or “end” in f1.txt
- Match a range of characters using **-** within bracket expressions
  - Example: grep [A-Z] f1.txt - match all lines that contain a capital letter in f1.txt
  - Example: grep ^[A-Z] f1.txt - match all lines that start with a capital letter
  - Example: grep [^a-zA-Z] f1.txt - match all lines that contain a non-alphabet character

Regex - Character Ranges

Syntax	Description	Equivalent
`[[:alnum:]]`	All letters and numbers.	`"[0-9a-zA-Z]"`
`[[:alpha:]]`	All letters.	`"[a-zA-Z]"`
`[[:blank:]]`	Spaces and tabs.	`[CTRL+V<TAB> ]`
`[[:digit:]]`	Digits 0 to 9.	`[0-9]`
`[[:lower:]]`	Lowercase letters.	`[a-z]`
`[[:punct:]]`	Punctuation and other characters.	`"[^a-zA-Z0-9]"`
`[[:upper:]]`	Uppercase letters.	`[A-Z]`
`[[:xdigit:]]`	Hexadecimal digits.	`"[0-9a-fA-F]"`

Regex - Quantifiers

- Quantifiers are metacharacters that specify the number of appearances of a character or character string

Syntax	Description
`*`	Zero or more matches.
`?`	Zero or one match.
`+`	One or more matches.
`{n}`	`n` matches.
`{n,}`	`n` or more matches.
`{,m}`	Up to `m` matches.
`{n,m}`	From `n` up to `m` matches.

Regex - Quantifiers

- The * quantifier is used to search for 0 or more matches of a character string
  - Example: grep m*and f1.txt - matches all lines containing some number of “m’s” followed by “and” (mand, command, and, etc) in f1.txt
- The + quantifier is used to search for 1 or more matches of a character string
  - Example: grep “m\+and” f1.txt - matches all lines containing some number of “m’s” followed by “and” (mand, command, etc) in f.txt
    - Need to wrap in quotes and use escape character “\”
- The {n, m} quantifier is used to search for n up to m matches
  - Example: head f1.txt | grep -E “m{3,5}”and - matches all lines containing 3 or 4 “m’s” followed by “and” from the output of head f1.txt
  - Example: grep -E “[aeiouAEIOU]{2}” f1.txt - match all lines that contain words with a two vowel sequence
    - Do not need escape characters when using extended regex

Regex - Alternation

- Alternation can be used to search for different strings
- The pipe, |, character is used for alternation
    - Example: grep -E 'bash|alias' f1.txt - matches all lines containing “bash” or “alias” in f1.txt

Regex - Grouping

- Grouped expressions are used to specify groups of characters in regex
- Parentheses, **()**, are used for grouped expressions
    - Example: grep -E “bash(rc)”?$ f1.txt - match all lines ending with “bash” or “bashrc”

Regex - Boundaries

- Boundaries are used to specify where character strings can be matched
- **\b** is used to set boundaries
  - Example: grep “\bse[et]\b” f1.txt - match lines with the words set or see (cannot be a part of other words)
  - Example: grep “\bse[et]” f1.txt - match lines with words beginning with set or see
  - Example: grep “se[et]\b” f1.txt - match lines with words ending with set or see

Bash Scripting

Bash Scripting - Creating a Script

Bash commands can be organized together in a Bash script. The file extension for bash scripts is **".sh"**. A Bash script must contain a "**#!**" followed by the path to the Bash interpreter at the top of the file.

Example: hello.sh

```
#!/bin/bash

echo "Hello"
```

To run "hello.sh", make it an executable file with **"chmod +x hello.sh"**, then run it with **"./hello.sh"** or **sh hello.sh**

- If you're working directory is different than the directory that holds the script, you can still run the script wit **"[(absolute or relative) path to script directory]/hello.sh"**

Bash Scripting - Command Line Arguments

You can pass command line arguments to a bash script with **sh <filename> arg1 arg2 arg3 .....**

These variables can be accessed in the script using **$[arg num]**

Example: print_args.sh

```
#!/bin/bash

echo $1

echo $2
```

```
$ sh print_args.sh hello world
```

- prints ‘hello’ ($1), then ‘world’ ($2) to the screen

Bash Scripting - Variables

Variable assignments cannot contain a space character on either side of the assignment operator.

Examples of variable assignment in Bash scripting:

`greeting=Welcome` - sets greeting to welcome

`num=10` - sets num to 10

Bash Scripting - Accessing Variables

Variables can be accessed using **$**

Examples of variable accessing in Bash scripting:

```
name=Steve
```

```
echo “hello $(name)”
```

- $(name) will be replaced with “Steve” in the output

Bash Scripting - Subshells

Sometimes it is necessary to assign a variable to the output of a bash command. This can be done with **[variable name] =$(command)**

user=$(whoami)	- assigns “user” to output of “whoami”
my_file=$(cat f1.txt)	- assigns my_file to output of “cat f1.txt”

Bash Scripting - Environment Variables

Environment variables differ from standard shell variables in that they are inherited by child processes of the current shell. To set an environment variable, use **export [variable name]=[value]**. Environment variable names are capitalized by convention.

`export PATH=/usr/bin` - sets “PATH” environment variable to “/usr/bin”

Environment variables can also be accessed with **$[variable name]**

`echo $PATH` - Writes “/usr/bin” to stdout

Bash Scripting - Array Basics

Arrays can be initialized with the following syntax: **array_name=(value1 value2 value3 …)**

`array=(Mike 2 file.txt…)` - creates an array called “array”

The values in an array do not need to be of the same type

Arrays in bash are 0 indexed. Array elements can be accessed with **${array_name[index]}**

`echo ${names[0]}` - prints “Mike” to stdout

Bash Scripting - Appending to an Array

Appending to an existing array can be done with **array_name+=(value(s))**

```
things=(1 pizza file.txt)

things+=(12.5)
```

- appends 12.5 to “things”

Bash Scripting - Removing Item from Array

To remove an item from an array, use **unset array_name[index]**

```
things=(1 pizza file.txt)
unset  things[1]
```

- removes “pizza” from “things” array

Bash Scripting - Arithmetic

There are many ways to perform arithmetic in Bash, but the preferred way is to use **$((expression))**

`$((2+3))` - evaluates to 5

`x=$((10*10))` - sets variable x to 100

Bash Scripting - Conditionals

The syntax for a conditional expression in bash is as follows:

if [(expr1) (comparison op) (expr 2)]; then

(do something)

else

(do something else)

fi

Bash Scripting - Comparison Operators

Arithmetic Comparisons

-lt	<
-gt	>
-le	<=
-ge	>=
-eq	==
-ne	!=

String Comparisons

=	equal
!=	not equal
<	less than
>	greater than
-n s1	string s1 is not empty
-z s1	string s1 is empty

Source: https://linuxconfig.org/bash-scripting-conditionals

Bash Scripting - Conditionals Example

```
#!/bin/bash

echo "enter the first number"
read num1

echo "enter the second number"
read num2

if [ $num1 -eq $num2 ]; then
        echo "the numbers match"
else
        echo "the number do NOT match"
fi
```

Source: https://linuxconfig.org/bash-scripting-conditionals

Bash Scripting - for Loops

for loop syntax:

```
for item in [LIST]
do
  [COMMANDS]
done
```

Example:

```
for element in Hydrogen Helium Lithium Beryllium
do
  echo "Element: $element"
done
```

Source: https://linuxize.com/post/bash-for-loop/

Bash Scripting - for Loop Over Range

for loops can loop over a range by assigning the iterator list to **{start..end}** or **{start..end..increment}**

Example:

```
for i in {0..3}
do
  echo "Number: $i"
done
```

Source: https://linuxize.com/post/bash-for-loop/

Bash Scripting - for Loops Over Array

for loops can loop over a range by assigning the iterator list to **“${array_name[@]}”**

Example:

```
BOOKS=('In Search of Lost Time' 'Don Quixote' 'Ulysses' 'The Great Gatsby')

for book in "${BOOKS[@]}"; do
  echo "Book: $book"
done
```

Source: https://linuxize.com/post/bash-for-loop/

Bash Scripting - for Loops Over Directory (cont.)

To loop over a directory with a for loop, replace the iterator list with **[path to dir]/***

Example: Loop over current directory

```
for item in *;
do
    [Command]
done
```

Bash Scripting - for Loops Over Directory (cont.)

Example: Loop over child directory

```
for item in child/*;
do
        [Command];
done;
```

Bash Scripting - for Loops Over Directory (cont.)

Example: Loop over only files in current directory

```
for file in *; do
        if [ -f "$file" ]; then
          echo "$file"
        fi
done;
```

Bash Scripting - while Loops

while loop syntax:

```
while [CONDITION]
do
  [COMMANDS]
done
```

Example:

```
i=0

while [ $i -le 2 ]
do
  echo Number: $i
  ((i++))
done
```

Source: https://linuxize.com/post/bash-while-loop/

Bash Scripting - while Loops with Files

Example: Reading a file line by line (file is passed into while loop at bottom)

```
file=/etc/passwd

while read -r line; do
  echo $line
done < "$file"
```

Source: https://linuxize.com/post/bash-while-loop/