Shell Scripting

The Shell

- The shell is generally considered to be the interface between the user and the operating system
  - Graphical User Interface
  - Command Line Interface

A Little History

- Shells in command line interfaces have been programmable in a limited form since at least the first UNIX shell
- The UNIX shell was completely rewritten in the late 1970s by Steve Bourne
  - A shell modeled after C was also written around this time
- UNIX isn't open source, so an open source implementation of the UNIX shell was developed, known as the Bourne again shell, or **bash**

Shells Today

- **bash** is the default shell on most Linux operating systems as well as macOS
  - Ubuntu and Debian use a shell known as **dash** for some startup scripts
  - Korn Shell (**ksh**) and Z Shell (**zsh**) are other common Bourne-like shells
- The C shell (**csh**) is another common shell
  - The default shell on GL at UMBC is **tcsh** or Turbo C Shell
- PowerShell is available on many Windows based computers

Non-Scripting Features of Shells

- Tab Completion
- History
  - Global (most shells)
  - Context-based (**fish**)
- Prompt Customization

Selecting a Shell

- All operating system come with a default shell
- In standard Linux installations the `chsh` command can be used to select any installed shell

```
chs -s SHELL_PATH USER_NAME
```

- To change your shell in GL, you must go to https://webadmin.umbc.edu/admin and click on "Edit my Shell and Unix Settings"

Bash

- For this class we will be using **bash**
- Even if a system does not use bash as the default shell, almost all systems have it
  - This makes scripts written in **bash** very portable
- **bash** has been managed since it's creation by the GNU Project
  - Code is open source, and can be contributed to at https://git.savannah.gnu.org/cgit/bash.git

Unix Utilities

- Bash scripts commonly rely on many simple programs to accomplish various tasks
- These programs are sometimes called Unix Utilities
  - Usually do only one thing
  - Most operate on STDIN and STDOUT by default
- macOS has many of these, but some are only available in the GNU Core Utils library

Getting Help In the Shell

- Most Unix utilities, and many other programs, install a manual along with program
- To access, use the `man` command followed by the command name
  - Sometimes help is access using the `info` command instead....
- `bash` also has it's own help utility, `help` which provides help on bash specific commands

In [ ]:
```
man cp
```

In [ ]:
```
help cd
```

Utilities You Already Use

- ls
- rm
- mv
- cp
- mkdir
- pwd

echo

- Echo is the most commonly used command to print something to the screen
- By default, newlines and other escapes are not "translated" into the proper character
  - Use the `-e` flag to accomplish this
  - To suppress the newline at the end of echo use the `-n` flag
- Echo can take multiple arguments, and will separate them by a space by default
  - To prevent separation by a space, use the `-s` flag

In [ ]:

```
echo "This will print as expected"
echo This will too
echo "This\ndoesn't\nhave\nnewlines"
echo -e "This\ndoes\nhave\nnewlines"
```

cat

- cat is used to con**cat**enate files together
- It is also used by lazy programmers (me included) to display the contents of a file to a screen, but usually there are better utilities for that
  - less
  - more

In [ ]:
```
cat src/perl/anchored.pl
```

In [ ]:
```
cat -n src/perl/anchored.pl
```

In [ ]:
```
cat src/perl/anchored.pl src/perl/unanchored.pl
```

sort

- sort sorts the lines of a file!
- By default this is done lexicographically
  - By using flags you can sort by numbers, months, etc.
- The `-r` flag will sort in revers order
- By using the `-u` flag, each unique line will be printed only once

In [ ]:
```
sort data/elements.txt
```

In [ ]:
```
sort -n data/elements.txt
```

In [ ]:
```
sort --key=2 data/elements.txt
```

In [ ]:
```
sort -n data/dir_sizes.txt
```

In [ ]:
```
sort -h data/dir_sizes.txt
```

uniq

- `uniq` in its default form accomplishes the same as `sort -u`
- Input to `uniq` is assumed to be sorted already
- `uniq` is useful to:
  - Count the number of times each unique line occurs
  - Ignore case when comparing lines
  - Only compare the first N characters of a line

In [ ]:
```
sort data/git_files.txt | uniq -c
```

In [ ]:
```
sort data/git_files.txt | uniq -c -w3
```

shuf

- `shuf` randomly permutes the lines of a file
- This is extremely useful in preparing datasets

In [ ]:
```
shuf data/elements.txt
```

head & tail

- The `head` and `tail` commands display the first 10 or last 10 lines of a file by default
  - You can change the number of lines displayed using the `-n` option
  - The value passed to `-n` when using `head` can be negative. This means return everything but the last n lines

In [ ]:
```
cat data/git_files.txt
```

In [ ]:
```
head -n1 data/git_files.txt
```

In [ ]:
```
tail -n1 data/git_files.txt
```

In [ ]:
```
head -n-1 data/git_files.txt
```

cut

- The cut command extracts columns from a file containing a dataset
- By default the delimiter used is a tab
  - Use the `-d` argument to change the delimiter
- To specify which columns to return, use the `-f` argument

In [ ]:
```
#head regex_starter_code/food_facts.tsv
cut -f1,2 data/food_facts.tsv | head
```

In [ ]:
```
cut -f1-4,10 -d, data/states.csv | head
```

paste

- `paste` does the opposite of `cut`
- Each line of every file is concatenated together, separated by a tab by default
  - Use the `-d` flag to change the delmiter

In [ ]:
```
paste data/elements.txt data/dir_sizes.txt
```

In [ ]:
```
paste -d, data/elements.txt data/dir_sizes.txt
```

find

- `find` is like an extremely powerful version of `ls`
- By default, `find` will list all the files under a directory passed as an argument
  - Numerous tests can be passed to find as arguments and used to filter the list that is returned

Common find Tests

- `-name` matches the name of the file or directory
- `-type` restricts the output to files (`f`) or directories(`d`)
- `-maxdepth` restricts the amount of recursion done
- `-size` restricts results to directories or files of the exact size
  - To look for a range, add a `+` or – before the number

In [ ]:
```
find . | head
```

In [ ]:
```
find . -type d | head
```

In [ ]:
```
find . -maxdepth 1 -type d
```

In [ ]:
```
find . -name "*ipynb"
```

In [ ]:
```
find ~/Teaching -type f -size +50M
```

Find Exercise

- Using find, return results that meet the following criteria
  - Are files, not directories
  - End in the characters "*.py"
  - Are greater than 20k in size

WC

- In some cases, it is convenient to know basic statistics about a file
- The `wc` or word count command returns the number of lines, words, and characters in a file
  - To only print ones of these, use the `-l`, `-w` or `-m` flags respectively

In [ ]:
```
wc to_sort1.txt
```

In [ ]:
```
wc -l to_sort1.txt
```

Other Helpful Utilities

- arch
- uname
- whoami
- yes

Shell Script Setup

- A shell script in the simplest form is just a list of commands to execute in sequence
- Is run using sh (or bash if you are not sure what shell you are in) script_file

In [ ]:
```
bash hello_simple.sh
```

Shebang Line

- On UNIX-like systems, if the first line of a file starts with `#!`, that line indicates which program to use to run the file
- Can be used with most any interpreted language
- Must be the full path of the command

```
#!/bin/bash
#!/bin/python
#!/bin/perl
```

- File must be executable

```
chmod +x FILE
```

In [5]:
```
./src/shell/hello.sh
```

```
Hello World
```

Variables

- Variables in bash can hold either scalar or array
  - Arrays are constructed using parentheses ()
- To initialize a variable, use the equals sign **with no spaces**

Declaring Variables Examples

In [7]:
```
a_scalar=UMBC
another_scalar="This needs quotes"
more_scalars=40
even_more=3.14
an_array=(letters "s p a c e s" 1.0)
#Don't do this
bad= "not what you want"
```

```
not what you want: command not found
```

Accessing Variables

- To access a variable a dollar sign (**$**) must be prepended to its name
- To access an array element, the variable name and index must occur inside of curly braces (**{}**)
    - Scalar values can be accessed this way to, but it is optional

Accessing Variables Examples

In [ ]:
```
echo $a_scalar
```

In [ ]:
```
echo ${a_scalar}
```

In [ ]:
```
echo $more_scalars
```

In [ ]:
```
echo $even_more
```

In [ ]:
```
echo ${an_array[1]}
```

In [ ]:
```
#Don't Do This
echo $an_array
```

In [ ]:
```
echo ${an_array[1]}
```

In [ ]:
```
echo ${an_array[*]}
```

Accessing Variables Exercise

- Given the following variable declarations, how would you print:
  - The letter d
  - All the letters

In [ ]:
```
letters=(a b c d e f g h i j)
```

String Interpolation

- Variables will be interpolated into strings when double quotes are used
  - If there are spaces, curly braces aren't needed, but its a good habit

In [ ]:
```
echo 'This class is at ${a_scalar}'
```

In [ ]:
```
echo "This class is at $a_scalar"
```

In [ ]:
```
echo "The schools website is www.$a_scalar.edu"
```

In [ ]:
```
echo "The athletics website is www.$a_scalarretrievers.com"
```

In [ ]:
```
echo "The athletics website is www.${a_scalar}retrievers.com"
```

String Operations

- Bash has numerous built in string operators allowing for
  - Accessing the length (**${#string}**)
  - Accessing a substring (**${#string:pos}**)
  - Performing a search and replace on a substring (**${#string/pattern/substitution}**)
  - Removing substrings

String Operation Examples

In [ ]:
```
echo ${a_scalar} ${#a_scalar}
```

In [ ]:
```
echo ${a_scalar} ${a_scalar:1}
echo ${a_scalar} ${a_scalar:2:2}
echo ${a_scalar} ${a_scalar::2}
```

In [ ]:
```
echo ${a_scalar} ${a_scalar/U/u}
echo ${a_scalar} ${a_scalar/V/u}
echo ${another_scalar} ${another_scalar/e/x}
echo ${another_scalar} ${another_scalar//e/x}
echo ${another_scalar} ${another_scalar//[a-z]/x}
```

In [8]:

```
#From the front of the string
echo ${another_scalar} "->" ${another_scalar#T*s}
#Longest possible match
echo ${another_scalar} "->" ${another_scalar##T*s}

#From the back of the string
echo ${another_scalar} "->" ${another_scalar%e*s}
#Longest possible match
echo ${another_scalar} "->" ${another_scalar%%e*s}
```

```
This needs quotes -> needs quotes
This needs quotes ->
This needs quotes -> This needs quot
This needs quotes -> This n
```

String Operation Exercises

- Given the following variable, change the output to be:
  - Lecture01
  - ipynb
  - CMSC433/Lecture01.ipynb
  - Lecture01.html

In [ ]:
```
string_to_change="Lecture01.ipynb"
```

Default Values

- Bash also allows default values to be used when the variable is **accessed**
  - Can either use just for that statement
  - Or set to be default for all future statements

Default Value Examples

In [ ]:
```
an_empty_var=
echo "1." $an_empty_var
echo "2." ${an_empty_var:-Default}
echo "3." $an_empty_var
echo "4." ${an_empty_var:=Default}
echo "5." $an_empty_var
```

Environmental Variables

- Environmental Variables are global variables in the widest sense
  - Used by all processes in the system for a user
  - Often set in initialization scripts or during boot
- Shells may modify but more often than not simply access them
- By convention, environmental variables are written in all uppercase letters

Environmental Variable Examples

In [ ]:
```
echo "Your home dir is: $HOME"
echo "You are logged into: $HOSTNAME"

echo "Your shell is: $SHELL"
echo "Your path is: $PATH"
echo "Your terminal is set to: $TERM"
```

Command Line Arguments

- Command line arguments are placed in the special variables $1 through $9
  - You can have more arguments, but they need to be accessed like ${10}
- The name of the script being executed in stored in $0
- The number of arguments is stored in $#

Command Line Argument Examples

In [9]:
```
cat src/shell/cla_examples.sh
```

```
#!/bin/bash
echo "The name of the file is $0"
echo "You passed $# arguments"

echo "The first argument is $1"
echo "The second argument is $2"

echo "All the arguments are $@"
```

In [11]:
```
./src/shell/cla_examples.sh --some-flag a_path additional_options another_one
```

```
The name of the file is ./src/shell/cla_examples.sh
You passed 4 arguments
The first argument is --some-flag
The second argument is a_path
All the arguments are --some-flag a_path additional_options another_one
```

Special Variables

- bash uses many other special variables to refer to convenient values to have
  - $$ is the process id of the currently executing script
  - $PPID is the process id of the process that the script was launched from
  - $? is the status of the last command executed

In [ ]:

```
echo "Process ID (PID) is: $$"
echo "Parent PID (PPID) is: $PPID"
whoami
echo "Status of last command: $?"
```

Putting it all together

- Write a simple bash script that takes in a file name as an argument, and does the following:
  - Sorts that file, and outputs the results to the screen
  - Paste that file to another file with the same name, but all o's replaced with e's, and outputs it to the screen

In [17]:
```
./src/shell/demo1.sh data/noodles
```

```
Gnochi
Penne
Ramen
Rice
Soba
Ramen   Sharp
Rice    Embroidery
Penne   Beading
Gnochi  Doll
Soba    Tapestry
        Leather
```