Week 3 Notes

Combining Commands and Files

- Executing Multiple Commands
  - `command1; command2; command3;`
    - Each command will be executed one after the other.
  - `command1 && command2`
    - command2 will be executed only if command 1 succeeds
    - If the return code is 0 it is true and if it is greater than 0 it is false
    - `ls && date -Q && wc -l /etc/profile` will display the dir listing followed by error that -Q is invalid; wc is not executed.
  - `command1 || command2`
    - command2 will not be executed if command1 succeeds
    - `ls /blah || date` will display current date after "No such file or directory"
    - `ls || date` will display just the directory listing
    - command2 is like a Plan B if command1 doesn't succeed.
  - Example `ls /blah ; date ; wc -l /etc/profile ;`
  - If we use parenthesis ie `(ls /blah ; date ; wc -l /etc/profile ;)` the command gets executed in a subshell and is returned back to the shell we are using.
  - We can use `echo $BASH_SUBSHELL` to return an integer which tells us at what level of execution we are.
    - `(echo $BASH_SUBSHELL)` will report a value of 1
    - `(ls; (date; echo $BASH_SUBSHELL))` will report a value of 2
  - Launching too many subshells could be expensive computationally.
- File Descriptors
  - Every command in linux has 3 file descriptors - `stdin` (0) , `stdout` (1), `stderr` (2).
    - `stdin` is a pointer to a stream that is coming from the keyboard or use input
    - `stdout` or `stderr` usually points to the screen where the display or output is made.
    - the three pointers are looking at only the stream of characters.
    - they can be directed to a file or a command, or the default behaviour can be left as it is.
  - Combining a command and a file
    - `command > file1`
      - `stdout` is redirected to `file1`
      - `file1` will be created if it does not exist
      - if `file1` exists, its contents will be overwritten
      - example : `ls -1 /usr/bin > file1` - displays no output on the screen because there is no error
      - `ls -1 /blah > file1` - displays an error. file1 is overwritten and is now 0 Bytes.
      - `hwinfo > hwinfo.txt`
      - trying this command in a folder where there is no w permissions will generate an error

        - The `cat` command tries to read from the provided file name if not given it tries to read from stdin (keyboard)
            - `cat > file1` will allow you to type content. The feature could be used to create text files on the command line. You can come out using the `Ctrl + D` option.
            - `cat file1` displays the content of `file1`
            - `cat` takes input from the keyboard and displays it on the screen (line by line; when you press enter) - Finish by pressing `Ctrl + D` to signify end of file.
    - `command >> file1`
        - contents will be appended to file1
        - new file1 will be created if it does not exist.
        - Example : `date >> file2 ; wc -l /etc/profile >> file2 ; file /usr/bin/znew >> file2 ;`
        - `cat >> file1` to append text to a file from command line. Come out using `Ctrl + D`

Redirections

- combining command and file (continued ..)
    - (contd..)
        - `command 2> file1`
            - redirects `stderr` to `file1`
            - `file1` , if it exists, will be overwritten.
            - `file1` will be created if it does not exist.
            - Example `ls $HOME /blah 2> error.txt`
        - `command > file1 2> file2`
            - `stdout` is redirected to `file1`
            - `stderr` is redirected to `file2`
            - Contents of file1 and file2 will be overwritten.
            - The output is in one file and the errors are in another file.
            - Example : `ls $HOME /blah > output.txt 2> error.txt`
            - `ls -R /etc > output.txt 2> error.txt` - permission related errors in error.txt
        - `command < file1`
            - `stdin` is redirected - a command expecting input from the keyboard could take the input from a file.
            - Example : `wc /etc/profile` behaves similar to `wc < /etc/profile`
        - `command > file1 2>&1`
            - command output will be redirected to `file1`
            - `2>` indicates `stderr` and that is being redirected to `&1` (first stream) which is `stdout`
            - contents of `file1` will be overwritten
            - Example : `ls $ HOME /blah > file1` output alone is sent to file1. Error on screen
            - Example : `ls $ HOME /blah > file1 2>&1` output and error is sent to file1.
        - `command1 | command2` Pipe

        - `stdout` output of command 1 is sent to `stdin` of command2 as input
        - Example `ls /usr/bin | wc -l`
    - `command1 | command2 > file1`
        - command1 and command2 are combined and the `stdout` of command2 is sent to `file1` . Errors are still shown on the screen.
        - Example `ls /usr/bin | wc -l > file1` - file1 has the number of lines counted by wc
    - `command > file1 2> /dev/null`
        - `/dev/null` file - A sink for output to be discarded. Like a "black hole"
        - We normally don't do anything with the `/dev` folder as there are sensitive system files there.
        - If you are confident that the script is running well and you do not want to display any error on the screen, you can redirect the `stderr` to `/dev/null`
        - `stderr` is redirected to `/dev/null`
        - Example : `ls $HOME /blah > file1 2> /dev/null`
        - Example : `ls -R /etc > file1 2> /dev/null` - file1 contains the output except errors
    - `command1 | tee file1`
        - Used in sitiations where you want to have a copy of the output in a file as well as on the screen.
        - The `tee` command reads from `stdin` and writes to `stdout` and file/s.
        - Example : `ls $HOME | tee file1` also `ls $HOME | tee file1 file2` for creating multiple copies
        - `diff file1 file2` comapares files line by line
            - no output if the files are identical
        - Example : `ls $HOME /blah | tee file1 file2 | wc -l` - Here `tee` keeps copy of output in a file and also sends output to `wc -l` for further processing.
        - Example : `ls $HOME /blah 2> /dev/null | tee file1 file2 | wc -l` to supress errors. Note location of `2>` is since the error is generated there.

Shell Variables - Part 1

- Creation, inspection, modification, lists
- Creating a variable
    - `myvar="value string"`
        - `myvar` can't start with a number, but you can mix alphanumeric and `_`
        - No space around the `=`
        - `"value string"` is the number,string or `command` . Output of a command can be assigned to myvar by enclosing the command in back-ticks.
- Exporting a variable
    - `export myvar="value string"` or
    - `myvar="value string" ; export myvar`
    - This makes the value of the variable available to a shell that is spawned by the current shell.
- Using variable values

    - `echo $myvar`
    - `echo ${myvar}`
        - can manipulate the value of the variable by inserting some commands within the braces.
    - `echo "${myvar}_something"`
- Removing a variable
    - `unset myvar`
    - Removing value of a variable `myvar=`
- Test is a variable is set
    - `[[ -v myvar ]] ; echo $?`
        - 0 : success (variable myvar is set)
        - 1 : failure (variable myvar is not set)
    - `[[ -z ${myvar+x} ]] ; echo $?` (the `x` can be any string)
        - 0 : success (variable myvar is not set)
        - 1 : failure (variable myvar is set)
- Substitute default value
    - If the variable `myvar` is not set, use "default" as its default value
    - `echo ${myvar:-"default"}`
        - if `myvar` is set display its value
        - else display "default"
- Set default value
    - If the variable `myvar` is not set then set "default" as its value
    - `echo ${myvar:="default"}`
        - if `myvar` is set display its value
        - else set "default" as its value and display its new value
- Reset value if variable is set
    - If the variable `myvar` is set, then set "default" as its value
    - `echo ${myvar:+"default"}`
        - if `myvar` is set, then set "default" as its value and display the new value
        - else display null
- List of variable names
    - `echo ${!H*}`
        - displays the list of names of shell variables that start with H
- Length of string value
    - `echo ${#myvar}`
        - Display length of the string value of the variable `myvar`
        - if `myvar` is not set then display 0
- Slice of a string value
    - `echo ${myvar:5:4}` (5 is the offset and 4 is the slice length)
        - Display 4 characters of the string value of the variable `myvar` after skipping first 5 characters.
    - if the slice length is larget than the length of the string then only what is available in the string will be displayed.
    - the offset can also be negative. However you need to provide a *space* after the *:* to avoid confusion with the earlier usage of the `:-` symbol. The offset would come from the right

hand side of the string.

- Remove matching pattern
  - `echo ${myvar#pattern}` - matches once
  - `echo ${myvar##pattern}` - matches maximum possible
  - Whatever is matching the pattern will be removed and the rest of it will be displayed on the screen.
- Keep matching pattern
  - `echo ${myvar%pattern}` - matches once
  - `echo ${myvar%%pattern}` - matches maximum possible
- Replace matching pattern
  - `echo ${myvar/pattern/string}` - match once and replace with string
  - `echo ${myvar//pattern/string}` - match max possible and replace with string
- Replace matching pattern by location
  - `echo ${myvar/#pattern/string}` - match at begining and replace with string
  - `echo ${myvar/%pattern/string}` - match at the end and replace with string
- Changing case
  - `echo ${myvar,}` - Change the first character to lower case.
  - `echo ${myvar,,}` - Change all characters to lower case.
  - `echo ${myvar^}` - Change first character to uppercase
  - `echo ${myvar^^}` - Change all characters to upper case
  - The original value of the variable is not changed. Only the display will be modified as the trigger commands are within braces.
- Restricting value types
  - `declare -i myvar` - only integers assigned
  - `declare -l myvar` - Only lower case chars assigned
  - `declare -u myvar` - Only upper case chars assigned
  - `declare -r myvar` - Variable is read only
  - Once a variable is set as read only you may have to restart the bash to be able to set it
- Removing restrictions
  - `declare +i myvar` - integer restriction removed
  - `declare +l myvar` - lower case chars restriction removed
  - `declare +u myvar` - upper case chars restriction removed
  - `declare +r myvar` - *Can't do once it is read-only*
- Indexed arrays
  - `declare -a arr`
    - Declare `arr` as an indexed array
  - `$arr[0]=”value”`
    - Set value of element with index 0 in the array
  - `echo ${arr[0]}`
    - Value of element with index 0 in the array
  - `echo ${#arr[@]}`
    - Number of elements in the array. The `@` symbol is a wild character to run through all the elements in the array
  - `echo ${!arr[@]}`

        - Display all indices used
    - `echo ${arr[@]}`
        - Display values of all elements of the array
    - `unset 'arr[2]'`
        - Delete element with index 2 in the array
    - `arr+=("value")`
        - Append an element with a value to the end of the array
- Associative arrays
    - `declare -A hash`
        - declare `hash` as an associative array
    - `$hash["a"]="value"`
        - set the value of element with index a in the array
    - `echo ${hash["a"]}`
        - value of element with index a in the array
    - `echo ${#hash[@]}`
        - number of elements in the array
    - `echo ${!hash[@]}`
        - display all indices used
    - `echo ${hash[@]}`
        - display values of all elements of the array
    - `unset 'hash["a"]'`
        - delete an element with index a in the array
    - Can do everything in the indexed array except append because there is nothing called the end of the array as there is no sequence for the elements of a hash
- Examples
    - `true` always returns exit code 0
    - `false` always returns exit code 1 (Check with `echo $?`)
    - To check whether a variable is present
        - `[[ -v myvar ]] ; echo $?` returns 1 if the variable is not present in the memory
        - `[[ -z ${myvar+x} ]] ; echo $?` returns 0 if variable is not present and 1 if it is present. `x` is a string that will be used as a replacement if the variable was not present.
    - Use of Braces
        - `myvar=FileName`
        - `echo $myvar`
        - `echo "$myvar.txt"` prints FileName.txt
        - `echo "$myvar_txt"` does not print anything as the variable `myvar_txt` does not exist
        - `echo "${myvar}_txt"` prints Filename_txt
        - Braces are useful in stating clearly the name of the variable.
        - Can also be used outside quotes `echo ${myvar}`
    - Does the variable we have created get passed on to the shell or any other program created within the shell
        - `myvar=3.14 ; echo $myvar`
        - `bash` one more level of bash
        - `ps --forest` to show that we are one level below

        - `echo $myvar` not present
        - Use `export myvar=3.14` to ensure this variable is available to all spawned sub shells.
        - Change value of variable within the child shell
        - modification of value is not reflected in the value of the variable in the parent shell
        - even if you do export of the variable within the child shell it will not change the value within the parent shell.
    - Use of back-ticks
        - `mydate=`date`` value of mydate will be output of date.
        - `mydate=`echo Sunday that is today` ; echo $mydate`
    - Manipulations for variables within the shell environment
        - We would like to have echo display a default value if variable is not available
        - `echo ${myvar:-hello}` the `-` indicates if the value is not present what is the display value
        - `echo ${myvar:-"myvar is not set"}`
        - Set the value if it was not set already
        - `echo ${myvar:=hello}` if absent / not set then set it to the value after `=`
        - If it is present it will not change
        - `echo ${myvar:?}`