Basic unix commands

Working with directories

- pwd prints the working directory
- ls lists files in the current directory. Add -a to include hidden files (such as ones whose names start with .)
- cd *directory* changes the current directory to *directory* (use . for the current directory, .. for the parent directory and - for the previous directory)
- mkdir *directory* creates a directory called *directory*

Printing to the terminal

- echo *"string"* prints *string* to the terminal
- date prints the current date to the terminal

Working with files

- cat *file* prints the contents of *file* to the terminal (provide more file arguments to concatenate all the provided files)
- touch *file* creates a file called *file* or updates the last accessed date of *file* if it already exists
- stat *file* shows when *file* was last modified and accessed as well as *file*'s size.
- rm *file* deletes file (you can also delete all files in a directory with $ rm -rf *directory*)
- mv *file directory* moves *file* to *directory* (or renames *file* to *directory* if *directory* doesn't exist, ex: $ mv file1 file2)

Working with processes

- jobs lists all current and suspended processes. Note that each job has an id.
- ctrl+z suspends the currently running process
- ctrl+c forcefully terminates the currently running process
- fg foregrounds/continues the execution of the most recently suspended processes in the foreground
- bg backgrounds/continues the execution of the most recently suspended process in the background
- kill kills a process.
- Note that fg/bg/kill can take %job id as an argument to specify that you want to foreground/background/kill that particular job. Example: $ kill %1

Other

- man *command* brings up the manual for *command*
- sleep *number* waits *number* seconds
- source *file* runs *file* in the current shell instance rather than executing it in a new shell instance

Shell operators

Comparison operators

- true always evaluates to true (exit status 0)

- false always evaluates to false (exit status 1)
- && is a conditional and
- || is a conditional or
- -gt is a number greater than
- -lt is a number less than
- -eq is a number equal to

Working with files

- *command < file* takes *file* as input for *command*
- *command > file* writes the output of *command* into *file*
- *command < file1 > file2* takes *file1* as input for *command* and writes the output into *file2*
- *command >> file* appends the output of *command* to *file*
- *command1 | command2* passes the output of *command1* to *command2*
- (bashism) *command &>> file* appends both the normal and error output of *command* to *file*
- *command* > /dev/null redirects output to basically a void, discarding it (can also discard error output by appending 2> /dev/null)

Variables

Setting and using variables

- *myVar=value* initializes a variable called *myVar* with value *value* (note: you cannot have any spaces around the = sign)
- $*myVar* expands *myVar* to the value that is bound to it (in this case, *value*)
- export *myVar* makes *myVar* accessible in child processes

Built-in variables

- $? is the exit status of the last command
- $# is the number of arguments
- $@ is all arguments, where each argument is separated by one space
- $*number* is the *number*'th argument (note: argument 0 is the name of the command/shell script/function itself, so the first argument provided to it would be 1)

More on variable expansions

- ${*myVar*} expands *myVar* to the value that is bound to it (preferred to omitting the curly braces since this removes ambiguity in some situations)
- (bashism) ${*myVar*:*lowerIndex*:*upperIndex*} gets a substring of *myVar* starting from *lowerIndex* until but not including *upperIndex* (note: :*upperIndex* can be omitted, in which case the substring will include up to the end of *myVar*.)
- *myVar*=$(*stuff*) sets *myVar* to the output of *stuff* (ex: *stuff* is $ echo hello)
- *myVar*=$((*mathematicalExpression*)) sets *myVar* to the result of *mathematicalExpression*; you can do arithmetic inside $(()).

Quoting

- Single quotes (i.e ‘’) keeps every character between the single quotes literally as is

- Double quotes (i.e “”) keeps every character between the double quotes literally as is except for variable expansions (i.e $someVariable), which it expands
- Commands can be enclosed in back ticks (i.e ``), and will expand to the result of the commands. ex: `$ for i in `ls`; do...`

Control flow

Evaluating conditional expressions

- test *expression* returns true or false based on the truth of the expression (ex: 5 -lt 3)
- [*expression*] is equivalent to test *expression* (note: you need a space between *expression* and the brackets)
- You can chain [expression]’s with && and || operators (ex: [expression1] && [expression2]
- ! [*expression*] negates [*expression*] (note the space between ! and [)
- (bashism) [[*expression*]] is like [*expression*] but it allows for more string operators, such as < and > for string comparison
- (bashism) ((*expression*)) is like [*expression*] but it allows for more number operators, such as < and > for number comparison

If statements

```
if test-commands; then
      commands
elif test-commands2; then
      commands2
else
      alt-commands
fi
```

While loops

1. ```
   while test-commands; do
         commands
   done
   ```
2. ```
   until test-commands; do
         commands
   done
   ```

For loops (where *list* is a space-delimited sequence of tokens)

```
for var in list; do
      commands
done
```

List can be $(seq 1 10) or any other whitespace-delimited list

Functions (bashism)

```
function-name () {
      commands
```

```
}
```

Functions can then be called as so: $ *function-name arg1 arg2* (remember, you can extract the values of *arg1* and *arg2* in the function body using $1 and $2 respectively)

There is also a case statement.

Executables and shell scripts

About shell scripts

Shell scripts contain commands that are run when the script is run.

Before the commands, shell scripts should contain:

- #!/bin/bash
    - This tells your terminal that when running the script, it should use bash to execute it. Replace bash with a different shell such as zsh if you want to use that instead.
- set -Eeuo pipefail
    - This is bash-specific. It makes the script's exit status the exit status of the first failing command if it encounters an error while executing.

Executing shell scripts

chmod +x *file* makes *file* executable

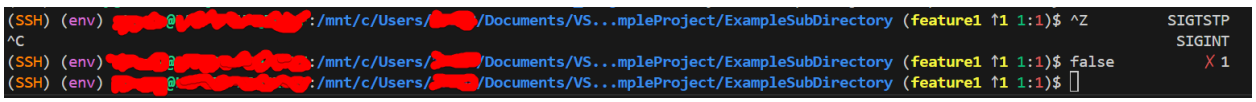
./file runs *file* if it's in the current directory and executable

Miscellaneous information

- You can use ctrl+arrow keys to move your cursor back or forward one word in the terminal
- You can use ctrl+l to clear the terminal
- You may be able to use the tab key to autocomplete a command
- You can use regular expression notation such as * and \w for and it will expand to all matches. ex: `$ rm *.txt`
- `$ du -h` lists how much storage different directories use

Shell configuration

I configured my shell to a certain extent. You can see more in the Github repository linked below.



Github repo for my shell: https://github.com/Racekid16/bashrc