Chapter 3: Expansions, Environment, and Functions

3.7 Shell Expansions

As it turns out, what we enter into the shell vs what actually gets run can be quite different. The shell undergoes a number of expansions. What that means is that the shell takes the command that we've entered, and then *expands* it if there are any available. A simple example of an expansion can be a wild card expansion. Suppose we're in a directory where there are a number of files of a specific type. You only want files that match a specific pattern (like, say files that have a specific file extension, or files that follow the same naming schema). You could list the entire directory, like:

```
ls
```

> file1 file2 file3 file4 someFileName.java vimFile

If you wanted to only grab the files that look like `file#`, you could use a wild card (`*`) in the spot where those files differ (here they differ only in one spot, the number):

```
ls file*
```

> file1 file2 file3 file4

The above is an example of *filename* expansion, which is just one of many expansions. Think of an expansion as a shortcut to access significantly more information.

History Expansions

A history expansion is one that ultimately allows for you to refer to your historical commands with just a few short keys. When you make a call `history`, you can see a log of all of your previous commands:

```
history
```

> ... 959 cp takeforever.sh 960 cp takeforever.sh failforever.sh 961 vim failforever.sh 962 ./failforever.sh 963 ./takeforever.sh & 964 ./failforever.sh & 965 ./takeforever.sh & 966 fg 3 967 jobs 968 kill -9 %+ 969 jobs 970 ls 971 cat coolName 972 vim vimfile 973 vim coolName < vimfile 974 ll 975 vim coolName << vimfile 976 vim coolName < vimfile 977 mv vimfile vimCommands 978 vim coolName < vimCommands 979 vim coolName 980 ls 981 cd someDirectory/ 982 ls 983 cd .. 984 cd testDirectory/ 985 ls 986 getmedonuts 987 which get 988 which getmedonuts 989 ls

There are a number of ways to use your `history`. First and foremost, you can keep it at a log of your commands and redirect the output to a file of your choosing. You can access commands by their historical *sequence number*, by prefixes, and even access specific arguments. The following are some common history expansions elaborated on from the book (pp 94):

Accessing Historical Commands:

- `!n`: Accessing a command by its *sequence number*:

```
1 !988
```

> which getmedonuts ~/someDirectory/getmedonuts

- `!-n`: The n^{th} previous command:

```
1 !-4
```

> getmedonuts Get your own donuts, you monster!

- `!!`: Repeat the most recently executed command:

```
1 !!
```

> getmedonuts Get your own donuts, you monster!

- `!somPrefix`: Executes the most recent command that matches the prefix

```
1 !getme
```

> getmedonuts Get your own donuts, you monster!

- `^matchers^replacers`: Takes the most previously executed argument, and replaces characters that match in *matchers* with the new characters in `replacers`:

```
1 ^ge^hi
```

> hitmedonuts -bash: hitmedonuts: command not found

Accessing Historical Arguments:

It's not entirely out of the question that you'll need to access the arguments of a previous command (maybe to send the same arguments to a similar, but marginally different script?). First, let's create a script that will take in 4 arguments. Open up a file editor to create a new script `takesArguments.sh`

```
#!/bin/bash

echo arg0 $0
echo arg1 $1
echo arg2 $2
echo arg3 $3
echo arg4 $4
```

This is a very brief introduction to how a shell script can read in arguments. Ultimately speaking, any command line argument passed into a shell script will be assigned a value (accessible through `$n` where `n` is its place in line). We will cover more of this in later sections and chapters.

If we then execute the script (make sure to `chmod` it to allow execution):

```
./takesArguments.sh helllo there general kenobi
```

> arg0 ./takesArguments.sh arg1 helllo arg2 there arg3 general arg4 kenobi

- `!*`: Accesses all arguments of the previous command:

  ```
  echo !*
  ```

  > echo helllo there general kenobi helllo there general kenobi

- `!$`: Accesses the last argument of the previous command

  ```
  echo !$
  ```

  > echo kenobi kenobi

- `!^`: Accesses the first argument of the previous command (re-run the `./takesArguments.sh` command above so it is your previous command)

  ```
  echo !^
  ```

  > echo helllo helllo

- `!:n` Accesses the n^{th} argument of the previous command (re-run the `./takesArguments.sh` command above so it is your previous command)

  ```
  echo !:2
  ```

  > echo there there

Alias Expansions

Up until now, we've been using predefined system calls (other than adding a rudimentary bash script to our path). It's likely that throughout your time working in the shell, you'll wish to use some shorthand notation for making a call. Aliasing allows for that! The format for aliasing is:

```
alias newAliasName=aliasValue
```

Aliasing allows for us to have user defined notation to make our own lives easier. I personally dislike typing out `ls -la` to see all of the files of a given directory in a nice format, so instead, let's alias it to a much simpler command, `ll`:

```
alias ll="ls -la --color=auto"
```

Now after entering that into the command line, you can now use `ll` to expand to `ls -la --color=auto`. You can ultimately alias pretty much anything, however, it should be noted that you should consider aliasing something for making your life easier with shorthand notation.

If you want to view all of your systems current aliases, execute `alias` with no arguments:

```
alias
```

> alias egrep='egrep --color=auto' alias fgrep='fgrep --color=auto' alias grep='grep --color=auto' alias l.='ls -d .* --color=auto' alias ll='ls -la --color=auto' alias ls='ls --color=auto' alias pico='nano -z' alias vi='vim' alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'

The above are my aliases, you may have some pre-existing from the system, you may have none (other than `ll`).

If you want to remove an alias, use the `unalias` command:

```
unalias aliasName
```

such as:

```
unalias pico
alias
```

> alias egrep='egrep --color=auto' alias fgrep='fgrep --color=auto' alias grep='grep --color=auto' alias l.='ls -d .* --color=auto' alias ll='ls -la --color=auto' alias ls='ls --color=auto' alias vi='vim' alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'

Note: aliases only last for as long as your session from within the system. If you wish to have an alias last, enter the command in your `.bash_profile`.

Brace Expansions

Brace expansions use curly braces to expand among the options. For example:

```
echo {one,two,three,four}
```

> one two three four

The above just expands among the curly braces, but if you add a prefix (or a suffix) to that command:

```
echo prefix-{one,two,three,four}-suffix
```

> prefix-one-suffix prefix-two-suffix prefix-three-suffix prefix-four-suffix

You can also use these expansions for sequential data so long as you tack a `..` between the start and stop values, such as **numbers**

```
echo {-1..23}
```

> -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

```
echo {23..-1}
```

> 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1

Or letters:

```
echo {d..m}
```

> d e f g h i j k l m

```
echo {m..d}
```

> m l k j i h g f e d

What is the point of this? Suppose you wanted to create multiple files with same suffix, such as a file extension:

```
touch {file1,file2}.sh
ls
```

> file1.sh file2.sh

Or create multiple directories for a course?

```
mkdir CS2750_{notes,projects,homework}
ls
```

> CS2750_homework CS2750_notes CS2750_projects

Tilde Expansions

We've seen this before already! The tilde character, `~` expands to become the path to the user's home directory. You can navigate from your home directory with:

```
cd ~/path/to/desired/directory/from/home
```

Additionally, you can also navigate backwards in your command history (not backward like `cd ..`) with:

```
cd -
```

So:

```
pwd
```

> /home/mjlny2/2750Materials/ch3

```
cd ..
pwd
```

> /home/mjlny2/2750Materials

```
cd -
pwd
```

> /home/mjlny2/2750Materials/ch3

Variable Expansions

The shell allows for us to create and use variables. We've already used a variable (and overwrote that variable): `PATH`.

Assignment

A variable can be anything from a numerical value, a string wrapped in quotations to an entire path. When variables are assigned, there must not be any whitespace in the assignment:

```
VARIABLE=neat
echo $VARIABLE
```

> neat

If you were to have whitespace in the assignment, you may receive a response, similar to:

```
OTHERVARIABLE = COOL!
```

> -bash: OTHERVARIABLE: command not found

Or

```
BADASSIGNMENT= something
```

> -bash: something: command not found

Working with Variables

As seen above variables are accessed by prefixing the variable name with a `$`:

```
VARIABLE=neat
echo $VARIABLE
```

> neat

However, suppose you wanted to have your variable nested inside of something else, like a string? What if we wanted to turn `neat` into `neatomosquito`:

```
echo $VARIABLEomosquito
```

>

The above prints nothing because the shell things we have a variable named `VARIABLEomosquito`. There are a number of methods to get around this:

```
echo "$VARIABLE"omosquito
```

> neatomosquito

Or using curly braces to expand the value:

```
echo ${VARIABLE}omoqsuito
```

> neatomosquito

Or also using the `\` to break the variable name:

```
echo $VARIABLE\omosquito
```

> neatomosquito

Math

In the event that you have numbers as variables, you can do math with them by using doubled parentheses preceded by a `$`:

```
TWO=2
THREE=3
FOUR=4
echo $((TWO + THREE * FOUR))
```

> 14

It should be noted that math in the shell follows the exact same orders of operations as math. The shell also has increment `++` and modulo `%` operators (these will come in handy when we start writing loops).

Removing Variables

Unless you start a new session or overwrite our variables, they won't be going away. To remove a variable, you can get rid of it by using `unset`:

```
unset TWO
echo $TWO
```

>

Since TWO no longer stores any value, an empty string is printed. If you find yourself curious to see what variables and functions (covered in 3.15) are set within your environment use the commands `set` or `declare` with no arguments.

Command Expansions

It is possible to expand the output of a command into a string so that it can worked with, or stored into a variable. This is done by wrapping the command (with its arguments) in a single layer of parenthes with a `$` as a prefix.

```
ls -la
```

> total 12 drwxr-xr-x 2 mjlny2 401451 45 Jun 15 01:24 . drwxr-xr-x 16 mjlny2 401451 4096 Jun 15 01:46 .. -rwx------ 1 mjlny2 401451 102 Jun 15 01:21 failFast.sh -rwx------ 1 mjlny2 401451 115 Jun 14 23:52 takeforever.sh

```
OUTPUT=$(ls -la)
echo $OUTPUT
```

> total 12 drwxr-xr-x 2 mjlny2 401451 45 Jun 15 01:24 . drwxr-xr-x 16 mjlny2 401451 4096 Jun 15 01:46 .. -rwx------ 1 mjlny2 401451 102 Jun 15 01:21 failFast.sh -rwx------ 1 mjlny2 401451 115 Jun 14 23:52 takeforever.sh

The output of the `$(ls -la)` did not render new lines, so the saved data are a single lined string. A useful case of this could be in saving your working directory for easy navigation back (in the event you would be navigating through a system you don't know entirely well):

```
myWorkingDir=$(pwd)
cd ~ && pwd
```

> /home/mjlny2

```
cd $myWorkingDir && pwd
```

> /home/mjlny2/2750Materials/ch3

Process Expansions

Process expansion allows for us to treat command outputs as if they were plain text. Take, for example, the `diff` command. `diff` typically is used to compare files line by line to pull out their differences. This can be done with commands as well by wrapping the commands and their arguments in parentheses preceded by a `<`. To see how this works, copy the following bash to create two similar, but mildly different directories:

```
mkdir processExpansions && cd processExpansions
mkdir directory_{1,2}
touch directory_{1,2}/file{1,2,3}
touch directory_2/file{a..d}
```

The above command creates two separate directories, `directory_1` and `directory_2` using brace expansions. Both have the same files, `file1`, `file2`, and `file3`, however, `directory_2` has additional files, `filea` through `filed`.

```
diff <(ls directory_1) <(ls directory_2)
```

> `3a4,7` `> filea` `> fileb` `> filec` `> filed`

In the above, using the process expansions, we were able to treat the `ls` commands as if they were files. You could also do this with a text editor:

```
vim <(ls -la)
```

This will then create a file for you to edit based off of the output of `ls -la`.

Filename Expansions

As noted in the introduction to this section, the filename expansions are those which use pattern matching to find a specific file. In a directory with multiple files named `file1`, `file2` and so on, you could match the pattern with the wild card `*` to list all the files that match the expansion:

```
ls file*
```

> file1 file10 file2 file3 file4

However, the pattern matching above is not the only pattern matching at all:

- `*`: The Wildcard: Matches any string *Example above*
- `?`: Matches on a single character

  ```
  ls file?
  ```

  > file1 file2 file3 file4

  Notice that `file10` did not show up?
- `[...]`: Matches on any one of the characters between the brackets:

```
ls fi[lnr]e
```

> fine fire

The above did not find the files with the `file*` pattern because we did not specifically say to match on anything after the `e`. However, to do so would look like:

```
 ls fi[lnr]e*
```

> file1 file10 file2 file3 file4 fine fire

- `[0-9]`: Matches on numerical values:

  ```
  ls file[0-9]
  ```

  > file1 file2 file3 file4

  It should be noted that this counts only for one digit:

  ```
  ls file[0-20]
  ```

  > file1 file2

  To be safe, use one digit for now.

- `[^..]`: Matches on any character not following the carrot:

  ```
  ls fi[^ln]e
  ```

  > fire

- `[[:className:]]`: Matches on a particular class of things. Classes can be: `alnum`, `alpha`, `digit`, `lower`, and `upper`

Note: File extensions typically do not work with hidden files (i.e. files in the file system prefixed by a `.`). That is intentional. It is possible to change this option (see Mastering Modern Linux pp 100).

3.8 Built-In Bash Commands:

The shell has built in commands. To view all of these, type:

```
help
```

To elaborate on any one of the commands within the list, type:

```
help commandName
```

We have already used a number of these commands, and we'll continue to learn more throughout the semester, however, the one that may be the most helpful for now is the `source` command.

Sourcing Files

Often times, you'll find you want to alias a command, or possibly set up a startup script or some function. When you write these into the `.bash_profile`, these commands have yet to exist since your session hasn't restarted. To gain access to those aliases/variables/whatever was written into the .bash_profile, you need to source it:

```
source .bash_profile
```

Any file can be sourced, not just `.` files. However, it should be noted that if the files are not the `.bash_profile` the sourced aliases / variables / etc. may not exist in your next session unless you source them again.

3.9 Shell Variables

There are a number of preset shell variables. You can see this by typing:

```
set
```

Some shell variables of note are:

- `OLDPWD`: Your previous working directory.
- `HISTFILE`: The location of your history
- `HISFILESIZE`: The amount of lines within your histfile
- `PWD`: Your current working directory
- `SHELL`: Your current shell.

More shell variables will be discussed throughout the coming chapters.

3.10 Environment

The `environment` of a program can be thought of as the shell in which the process is running. That process is running in an "environment", and it has access to all of the "environment variables" of that "environment". Those "environment variables" are really just variables that are part of the shell. Create a basic shell script `basicScript.sh` (don't forget to change the mode with `chmod`):

```
#!/bin/bash
echo In basic script, I really want $COOLSTUFF
```

Now, if you try to run `basicScript.sh`, the output will be:

```
./basicScript.sh
```

> In basic script, I really want

Nothing was printed out for cool stuff. That's because we neither defined it inside of the file, nor did we defined it outside as a variable in the shell (the environment). Let's try that again:

```
export COOLSTUFF="Pizza, my favorite food"
./basicScript.sh
```

> In basic script, I really want Pizza, my favorite food

The script `basicScript.sh` read `$COOLSTUFF` out of the shell's environment.

Inline Environment Variables

It's not always the case that you'll be comfortable exporting environment variables all the time. There has to be a better way to do this, and there is! In the same line you run your command, you add the environment variables beforehand. First, let's unset `COOLSTUFF`

```
unset COOLSTUFF
echo $COOLSTUFF
```

Now:

```
COOLSTUFF="Pizza, my favorite food" ./basicScript.sh
```

> In basic script, I really want Pizza, my favorite food

With this notation, you don't need to worry about setting too many environment variables. You can do it all within one line.

3.12 Default File Permissions

It is possible to specify the file permissions of a file upon its creation with `umask`. The ways in which the bits are flipped, however, act in the exact opposite order as those of `chmod`, in that, if the bit is flipped to 1, then those permissions are by default denied to the user group.

```
umask
```

> 0022

The above octal numbers are:

`000010010`, or more easily read: `000` `010` `010`. These bits specify that both group and others may not have write permissions as write is the second bit of the octal number (`rwx`).

3.13 Shell Startup/Initialization

When a user logs into a system, it selects the users password file entry which contains:

- Login Name
- Password
- UserID
- GroupID
- Shell Program
- etc.

Since we're starting our programs up with bash, after logging in, the shell then loads the system wide `profile`, and then the user specific `.bash_profile`. Depending on the Linux distribution, there may also exist a `.bashrc` (rc stands for run command). In your .bash_profile, you likely have the conditional:

```
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

This code sources the `.bashrc` if it exists (i.e. the `.` before a file can also be used to source).

3.14 Quotes and Special Characters

There are a number of special characters in bash. It's not always the case that your files won't have special characters in their names, however. This can be due to any number of factors, but there will definitely come a time when you need to access a file with a `*` or a `$`. To access these, you need to use the escape character `\`.

Suppose you had a file named `2018$Revenue`. If you were to attempt to open that file with:

```
vim 2018$Revenue
```

Your vim would instead open a file named `2018` . The shell thinks that `$revenue` is a variable because of the special character. To escape that, you need to write:

```
vim 2018\$Revenue
```

As for quotations, they follow the rule where quotes either have to be escaped if you have all double quotes:

```
echo "You say \"tomato\" and I say \"tomato\""
```

Or you can wrap your strings with single quotes:

```
echo 'You say "tomato" and I say "tomato"'
```

The escape character works the same for single quotes as well:

```
echo 'Heghlu\'meH QaQ jajvam is Klingon for It\'s a good day to die, a saying Klingons say before going into battle.'
```

3.15 Functions

Up until now, we've see bash variables, expansions, and even a brief introduction to regular expressions. All fo these can be incredibly powerful, but one of the most powerful tools a computer has is its ability to abstract and use functions! Bash can do the same thing every other language can as well!

The basic function syntax is:

```
function funcName () {
  command
  ...
  command
}
```

Note: You don't necessarily need to declare the function with the **function** keyword. You can simply type:

```
funcName () { ... }
```

Suppose you wanted a function to immediately log all of the files in a particular directory after you entered it:

```
function changeAndLog () {
  echo Changing directory to $1
  cd $1 # $1 is the first argument passed to the function
  ls -la
}
```

When using the function, you simply call it like any other command:

```
changeAndLog someDirectory/
```

> Changing directory to someDirectory/ total 8 drwxr-xr-x 2 mjlny2 401451 24 Jun 15 02:07 . drwxr-xr-x 18 mjlny2 students 4096 Jun 17 00:32 .. -rwx------ 1 mjlny2 401451 54 Jun 15 02:06 getmedonuts

We will discuss functions in great detail in the following chapters.