Advanced command structures

shell expansion

All the words you type after the prompt to form a command are interpreted by the shell. The command line interpretor or shell in ubuntu is `bash`, which stands for `Bourne Again SHell`.

The command line will be cut in pieces everytime the interpreter sees one or more `consecutive spaces` or `tabs` and this forms arguments (the spaces and tabs will be removed). The first argument is the `command` and all the other arguments are given as values to this command. This mechanism is called `shell expansion`. This is the reason that consecutive spaces will be trimmed with the echo command:

```bash
student@linux-ess:~$ echo I like to game
I like to game
student@linux-ess:~$ echo     I    like            to                    g
I like to game
student@linux-ess:~$ echo I    like     to     game
I like to game
student@linux-ess:~$ echo     I             like        to
I like to game
student@linux-ess:~$ echo   I             like          to
I like to game
```

Single and double quotes

If we want to retain the spaces we have three options. The `first option` is to use `double quotes`. With this option the spaces will be retained and variables

will still be interpreted.

```bash
student@linux-ess:~$ action="play       games"
student@linux-ess:~$ echo    I             like      to
I like to play games
student@linux-ess:~$ echo "    I             like      to
    I             like      to                  play       games
```

The `second option` is to use `single quotes` . With this option the spaces will be retained but the text will not be interpreted. So variables won't be changed to their value.

```bash
student@linux-ess:~$ action='play       games'
student@linux-ess:~$ echo    I             like      to
I like to play games
student@linux-ess:~$ echo '    I             like      to
    I             like      to                  $action
```

The `third option` is to escape every space. Don't use this mechanism within the value of a variable because it won't work.

```bash
student@linux-ess:~$ action='play       games'
student@linux-ess:~$ echo \ \  I \ \ \ \ \ \ \ like \ \ to \ \ \ \ \ \ \
    I       like   to              play       games
```

File globbing

When specifying filenames, we can get the shell to generate the filenames dynamically by giving a certain pattern. For example: we might want to find all the files starting with `temp` followed by whatever text or extention. The concept where we generate file names dynamically is called *file globbing*. There are a

couple of special characters that we can use as seen in the example below:

bash

```
student@linux-ess:~/globbing$ ls
afilea  file1  file2  file3  File4  File5  filea  fileabc  FileABC  fileb
student@linux-ess:~/globbing$ ls file*
file1  file2  file3  filea  fileabc  fileb  filebc
student@linux-ess:~/globbing$ ls *a
afilea  filea
student@linux-ess:~/globbing$ ls *a*
afilea  filea fileabc
student@linux-ess:~/globbing$ ls F*
File4  File5  FileABC  Filec
student@linux-ess:~/globbing$ ls F*ile*
File4  File5  FileABC  Filec
```

an asterisk (`*`) in *file globbing* means zero, one or more characters can be whatever they want. This is often called a wildcard that we can use one or multiple times in a filename. Another option would be a question mark (`?`) which is interpreted as exactly *one character* can be what they want as seen in the following example:

bash

```
student@linux-ess:~/globbing$ ls
afilea  file1  file2  file3  File4  File5  filea  fileabc  FileABC  fileb
student@linux-ess:~/globbing$ ls File?
File4  File5  Filec
student@linux-ess:~/globbing$ ls file??
filebc
student@linux-ess:~/globbing$ ls ?fi*
afilea
```

Lastly we can also use square brackets (`[ ]`) which usually contain one or more characters in between the brackets. The brackets define one character that matches one of the characters between the brackets:

```bash
student@linux-ess:~/globbing$ ls
afilea  file1  file2  file3  File4  File5  filea  fileabc  FileABC  fileb
student@linux-ess:~/globbing$ ls file[12]
file1  file2
student@linux-ess:~/globbing$ ls file[a]
filea
student@linux-ess:~/globbing$ ls file[1ac]
file1  filea
```

When using brackets we also can define ranges:

```bash
student@linux-ess:~/globbing$ ls
afilea  file1  file2  file3  File4  File5  filea  fileabc  FileABC  fileb
student@linux-ess:~/globbing$ ls file[a-z]
filea  fileb
student@linux-ess:~/globbing$ ls File[A-Z]*
FileABC
student@linux-ess:~/globbing$ ls File[a-zA-Z]*
FileABC  Filec
student@linux-ess:~/globbing$ ls file[0-9]
file1  file2  file3
```

When using brackets we also can exclude the specified range by specifying a caret (^) or an exclamation mark (!) at the beginning:

```bash
student@linux-ess:~/globbing$ ls
afilea  file1  file2  file3  File4  File5  filea  fileabc  FileABC  fileb
student@linux-ess:~/globbing$ ls file[a-z]*
filea  fileabc  fileb  filebc
student@linux-ess:~/globbing$ ls file[^a-z]*
file1  file2  file3
student@linux-ess:~/globbing$ ls file[!a-z]*
file1  file2  file3
```

Prevent file globbing

We can prevent file globbing by *escaping* the special characters in our command. Escaping can be done by placing a `\` in front of the character. This tells the shell to interpret the next character as a regular symbol rather than the special operation:

```bash
student@linux-ess:~/globbing$ ls
 File4   File5   FileABC   Filec   afilea  'file*'   file1   file2   file
student@linux-ess:~/globbing$ ls file*
'file*'   file1   file2   file3   filea   fileb   filebc
student@linux-ess:~/globbing$ ls file\*
'file*'
student@linux-ess:~/globbing$ echo **** TITLE ****
File4   File5   FileABC   Filec   afilea  'file*'   file1   file2   file3
student@linux-ess:~/globbing$ echo \*\*\*\* TITLE \*\*\*\*
**** TITLE ****
```

> You can imagine what would happen if we would delete the file *file** without using the escaping.

Aliases

Aliases are a way to give a simple name to a rather complex command as seen below:

```bash
student@linux-ess:~$ alias show='ls -lah'
student@linux-ess:~$ show
total 16K
drwxr-xr-x 1 student student  512 Jun  4 22:19 .
drwxr-xr-x 1 root  root       512 Mar  7 17:09 ..
-rw------- 1 student student 1.1K May 22 22:41 .bash_history
-rw-r--r-- 1 student student  220 Mar  7 17:09 .bash_logout
```

```
-rw-r--r-- 1 student student 3.7K Jun  4 21:25 .bashrc
...
```

Aliases are often used for implementing an extra layer of security:

bash

```
student@linux-ess:~$ alias rm='rm -i'
student@linux-ess:~$ rm jokes.txt
rm: remove regular file 'jokes.txt'? y
```

You already used aliases. For example the ls command we use prints colored text in its output. This is because we use the alias. If we put a `\` in front of a command it will use the command instead of the alias.

bash

```
student@linux-ess:~$ alias ls
alias ls='ls --color=auto'
```

If we want to remove an alias we can use the unalias command:

bash

```
student@linux-ess:~$ unalias rm
```

If you want to keep an aliase for future use (open a new shell, reboot, ...) you can add it to a (new) hidden file in your homefolder named `.bash_aliases`

bash

```
student@linux-ess:~$ cat .bash_aliases
alias memory='free --giga -h'
student@linux-ess:~$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal
alias egrep='egrep --color=auto'
```

```
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias memory='free --giga -h'
student@linux-ess:~$ memory
               total        used        free      shared  buff/cache   av
Mem:            3.9G        301M        3.1G        1.0M        492M
Swap:             0B          0B          0B
student@linux-ess:~$
```

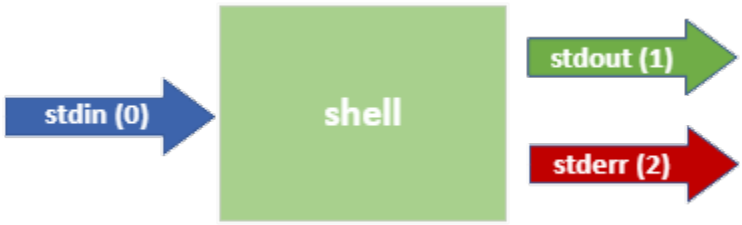
The order in which the shell checks for commands in the shell:

- Aliases. Names set by the alias command that represent a particular command and a set of options.
- Built-in command. This is a command built into the shell.
- Filesystem command. This command is stored in and executed from the computer's filesystem. (These are the commands that are indicated by the value of the PATH variable).

I/O redirection

I/O Streams

When working with the shell we actually work with something called streams. There are 3 basic streams available when using a shell:



The most basic example is called `stdin`. This is the stream that we use to

input data into the shell using our keyboard.

The output that gets generated from running commands is split up in two seperate streams:

- `stdout` gives us all the regular command output. By default this output gets printed on our screen.
- `stderr` gives us all the error messages a command generates. By default these error messages get printed on our screen. Because both `stdout` and `stderr` get printed on our screen, we don't notice the difference between the two.

Every stream has its own identifier (=number) as seen in the image above. These identifiers are important in the next paragraphs.

Stream redirection

We can redirect any of these streams to make the output go *somewhere else*. Often 'somewhere else' means to a file. This means we can seperate regular output (stdout) and errors (stderr) to get saved into seperate files. Below we begin with an example on how you can save the regular output of a command to a file:

bash

```
student@linux-ess:~$ ls / 1> listrootfolder
student@linux-ess:~$ head -6 listrootfolder
bin
boot
dev
etc
home
lib
```

The `1>` means we redirect stream `1` to the file `listrootfolder`. Stream `1` refers to the `stdout` stream. Note that in this case the number `1` is optional, so the command below will work as well:

bash

```
student@linux-ess:~$ ls / > listrootfolder
```

If the command also generates errors than these will still be printed on our screen.

> You might recognize this syntax as we've used it before in chapter 5. We used the command `echo hello world > ourfile` to write the string `hello world` to the file `ourfile`.

If we want to redirect `stderr` we can use the same concept as follows:

bash

```
student@linux-ess:~$ find / 2> /dev/null
```

In this example every file that's found (which is regular output) will be printed on our screen. The errors that are generated (eg. not able to dive in a certain directory to look for files because of lack of privileges) will not be shown on the screen because they are redirected to the black hole. This path `/dev/null` is often referred to as `the void` or `the black hole` because we can throw in as much "garbage" as we want.

And we could even combine redirecting both streams to seperate files in one command:

bash

```
student@linux-ess:~$ find / > results.txt 2> errors.txt
```

If we want to redirect both `stderr` and `stdout` to the same file we can use the `&>` operator as follows:

bash

```
student@linux-ess:~$ find / &> results_and_errors.txt
```

If you want to redirect to a file and add the contents to that file you need to use two `>>`. This is because one `>` will always empty the file before adding the content:

bash

```
student@linux-ess:~$ echo "text 1" > testfile
student@linux-ess:~$ cat testfile
text 1
student@linux-ess:~$ echo "text 2" > testfile
student@linux-ess:~$ cat testfile
text 2
student@linux-ess:~$ echo "text 3" >> testfile
student@linux-ess:~$ cat testfile
text 2
text 3
student@linux-ess:~$
```

Control operators

Seperating commands

We can use a `;` (semicolon) character to seperate multiple commands on one line. Each command can have its own options and arguments and they will be ran sequentially. The shell will wait for a command to finish before starting the next one:

bash

```
student@linux-ess:~$ echo hello ; echo pxl ; pwd
hello
pxl
/home/student
```

Logical operators

Next up are some control operators that we can use that you might know from other use cases or environments:

- logical *AND* operator (`&&`): The second command will only execute if the first command scuceeds

bash

```
student@linux-ess:~$ echo first && echo second
first
second
student@linux-ess:~$ zecho first && echo second
Command 'zecho' not found, did you mean:
command 'echo' from deb coreutils (8.30-3ubuntu2)
Try: sudo apt install <deb name>
```

- logical *OR* operator (`||`): The second command is only executed when the first command fails

bash

```
student@linux-ess:~$ echo first || echo second
first
student@linux-ess:~$ zecho first || echo second
Command 'zecho' not found, did you mean:
command 'echo' from deb coreutils (8.30-3ubuntu2)
Try: sudo apt install <deb name>
second
```

We can combine both operators as well to simulate an *if-then-else* like structure as follows:

bash

```
student@linux-ess:~$ touch testfile
student@linux-ess:~$ ls test*
testfile
student@linux-ess:~$ rm testfile && echo file deleted || echo failed to d
file deleted
student@linux-ess:~$ ls test*
```

```
ls: cannot access 'test*': No such file or directory
student@linux-ess:~$ rm testfile && echo file deleted || echo failed to d
rm: cannot remove 'testfile': No such file or directory
failed to delete
```

In this example it would be best to redirect our errors to the *void* as well, because we generate our own fault message now. Use `/dev/null` as seen earlier to do this in combination with the redirection of the stderr as follows:

bash

```
student@linux-ess:~$ touch testfile
student@linux-ess:~$ ls test*
testfile
student@linux-ess:~$ rm 2> /dev/null testfile && echo file deleted || ech
file deleted
student@linux-ess:~$ ls test*
ls: cannot access 'test*': No such file or directory
student@linux-ess:~$ rm testfile 2>/dev/null && echo file deleted || echo
failed to delete
```

Previous
6 Software & packages

Next
Assignment - file globbing