Workshop Command-Line Beginner II

Keyboard Shortcuts

- *Ctrl-U*: remove everything before the cursor
- *Ctrl-K*: remove everything after the cursor
- *Ctrl-W*: remove word before cursor
- *Ctrl-Y*: paste the last thing to be cut
- *Ctrl-A*: move cursor to beginning of line
- *Ctrl-E*: move cursor to end of line
- *Ctrl-L*: clear terminal screen
- *Ctrl-R*: reverse search
  - search history for last command that corresponds with input
  - useful if you want to quickly redo a previous command

Ctrl-R Example

```
$ echo foo
foo
$ echo bar
bar
(reverse-i-search)`e': echo bar`
(reverse-i-search)`echo f': echo foo
```

Stream operations

`wc`

Print the number of characters/words/lines

```
$ echo "a random sentence with words" | wc -m # number of characters
29
$ echo "a random sentence with words" | wc -w # number of words
5
$ echo "a random sentence with words" | wc -l # number of lines
1
```

`head`

Print the first `n` bytes or lines of the input

```
$ echo -e "1\n2\n3\n4\n5" | head -n 2 # -e option for \n as newline
# only take the first two lines
1
```

```
2
$ echo -e "1\n2\n3\n4\n5" | head -n -2 # drops the last two lines
1
2
3
$ echo hello | head -c 3
hel
$ head -c 9 /dev/urandom | base64
# some random characters
```

cut

Removes some sections from each line

```
$ echo "hello world" | cut -c 1-3
hel
$ echo "hello world" | cut -c 1,3
hl
$ echo "hello world" | cut -d " " -f 2
world
```

sort

Sort all lines

```
$ echo -e "c\nb\na"
c
b
a
$ echo -e "c\nb\na" | sort
a
b
c
$ echo -e "c\nb\na" | sort -r # reverse order
c
b
a
$ echo -e "2\n11 | sort
11
2
$ echo -e "2\n11 | sort -n # sort numbers
2
11
```

`-h` option will sort numbers in human readable format like 2M (for megabyte) and 1G (for gigabyte)

```
$ du -hs * | sort -h
# files and directories in current directory sorted on human readable size
```

`uniq`

Remove repeated lines, but only if they are right next to each other. So you might have to sort first.

```
$ echo -e "a\nb\na"
a
b
a
$ echo -e "a\nb\na" | uniq
a
b
a
$ echo -e "a\nb\na" | sort | uniq
a
b
```

`-c` makes `uniq` count occurences of each unique value

```
$ echo -e "a\nb\na" | sort | uniq -c
2 a
1 b
```

Stream editing with `sed`

```
$ echo "hello world" | sed "s/hello/goodbye/"
goodbye world
```

Replace first occurence of hello with goodbye You can use regular expressions

```
$ echo "hello hello" | sed "s/hello/bye/g"
bye bye
```

`g` tells `sed` to replace *every* occurence

```
$ echo "hello Hello" | sed "s/hello/bye/gi"
bye bye
```

`i` tells `sed` to search case-insensitive

Editing files with `sed`

```
$ sed -e "/^#.*/d" -e "s/#.*//g" -i file-with-comments
# file now has no comments
```

- `/regex/d` removes all lines that match
- `^` means start of the line
- `-e` allows you to do multiple operations with one command
- `-i` tells sed to edit the file *inplace*

Command substitution

```
$( command )
```

This expression will be replaced by the output of the command.

```
$ touch $(hostname | sed "s/o/a/g")
```

Hostname print the current machine's name, so this will create a file whith the name of the current machine. But with the o's replaced with a's.

```
$ echo $(cat file-with-spaces-and-newlines)
a b c
$ echo "$(cat file-with-spaces-and-newlines)"
a   b
c
```

If output contains whitespace, each word is interpreted as seperate argument. Use quotes to preserve whitespace, everything between is interpreted as single argument.

Command substitution as file

```
<( command )
```

Represents a fake file containing the command's output.

Boring example:

```
$ cat <(echo hello)
hello
```

Cool example:

- `curl` prints content of website
- `vim` text editor, opens file

```
$ vim <(curl michiel.ulyssis.be)
```

Doesn't work with nano :(

Finding files with `find`

```
find [path] [expression]
```

Prints files and directories. If path is omitted, the current working directory is used.

Tests

- `-type f` or `d` for only files/directories respectively
- `-name name`
  - can include wildcards: eg finding all txt files `find -name "*.txt"` quotes are required, else bash will interpet * als wildcard
- `-user name` all files owned by a certain user
- many others, check the man page
- `find` gives alot of `Permission denied` errors when you are not allowed to read certain files. Use `find [path] [expression] 2> /dev/null` to filter those out

Making `find` do things

- `-delete` instead of print, delete the files
- `-exec command {} \;` execute command, with `{}` substituted with the found file, for every found file.

  ```
  $ mkdir somedir
  $ find -name "*.txt" -exec cp {} somedir \;
  ```

  Copies all txt files to the somedir directory.

  ```
  $ find somedir -name "*.txt" -delete
  ```

  Deletes all txt files in somedir.

Combining `find` and `xargs`

Recap `xargs`

Executes a given command with with arguments comming from stdin.

```
$ echo "world everyone" | xargs echo hello
hello world everyone
$ echo "world everyone" | xargs -n 1 echo hello
# -n specifies how many args per command
hello world
hello everyone
```

- There are also other options like `-l` to execute the command for each line.
- `-s` for number of characters per command.

- Personal favorite: `-P` for how many processes executed in parallel for if the command takes a while, to speed things up on SMP systems

Combining find and xargs

Little bit more flexible than `-exec`

```
touch filename\ with\ spaces.txt
$ find -name "*.txt" | xargs rm -f
rm: cannot remove './filename': No such file or directory
rm: cannot remove 'with': No such file or directory
rm: cannot remove 'spaces.txt': No such file or directory
```

Files with spaces might cause problems, seen as seperate argument The solution is to seperate args with a null byte.

```
$ find -name "*.txt" -print0 | xargs -0 rm -f
```

Connectors

Return codes

When commands end, they return a return code (number). When the command failed, that code will be non-zero. Zero means everything went OK.

&&

These are actually logic operators

```
$ command1 && command2
```

The second command will only be executed if the first succeeds.

||

```
$ command1 || command2
```

The second command will only be executed if the first one fails.

;

Always executes both, used to put multiple commands on one line.

Asynchronous jobs

```
$ command&
```

`&` behind the command will cause the command to be executed asynchronously. During the commands execution, you will keep control over the shell and be able to do multiple things at the same time.

```
$ sleep 5&
$ sleep 10& # you can do this on the same line
# now we play the waiting game
```

It will report when it's done, and it will keep printing to stdout. You can use `jobs` to see what's in the background, and `fg` to bring the last job to the foreground and take control over it.

```
$ sleep 500&
$ jobs
[1]+  Running                 sleep 500 &
$ fg
#Ctrl-C, now it's closed
```

Ctrl-Z will suspend the current process and put it in the background, The process's progress will be paused though, if you want it to continue it's work in the background type `bg`.

```
$ sleep 5
# Ctrl-Z
$ jobs
[1]+  Stopped                 sleep 5
$ bg
[1]+ sleep 5 &
[1]+  Done                    sleep 5
```

This is useful for putting editors in the background and you need to use the shell but don't want to close your editor just yet.

```
nano -z # z option allows suspending
# type some stuff and do Ctrl-z
Use "fg" to return to nano.

[1]+  Stopped                 nano -z
fg
# etc etc
```

Killing a job can also be done using `kill %1` with 1 a job id reported by `jobs`.

`disown` will detach a job from the shell, you can than close the shell and the job will keep running

Basics of bash scripting

Variables

Getting the content of a variable is done by putting a `$` infront of it.

Setting variables:

```
$ a="some string" # no spaces between varname, = and content
$ echo "$a" # best use quotes
some string
$ username=$(whoami)
$ echo $username
username
```

Variable names can only be alphanumeric, so anything else will mean the rest of the string

```
touch /tmp/$username/newfile # this works
touch /tmp/$username1 # wont work
touch /tmp/${username}1 # curly brackets fixes this
```

Your shell already have several variables set, these are called environment variables.

```
$ echo $HOME
/home/username
$ echo $PWD # $(pwd) has the same result
/path/to/current/directory
$ echo $USER
username
```

You can request all of them using `env`

Some special variables

```
$ echo $RANDOM
# random number

$ echo $? # return/exit code of previous command
0
$ false
$ echo $?
1

$ echo $$
# pid of current shell
```

```
$ echo $!
# pid of last job in background
sleep 5&
[1] 7012
$ echo $!
7012
```

Killing a command that has been running for to long

```
some_command="sleep 10000000000"
timeout=5
$some_command & sleep $timeout; kill -9 $! # use eval or not?
# if command takes to long, kill it
```

for loops

```
for item in $list
  do_something_with $item

for item in $list
do
  multiple
  commands
  with $item
done
```

`$list` should be a list with each item seperated with whitespace (newlines work too)

```
for char in a b c
do
  echo character: $char  # each char on a seperate
done
```

Don't do the next example, when using quotes, the list will be one item, containing the whitespace

```
for char in "a b c"
do
  echo characters: $char  # only one line
done
```

Example: getting contents of all files in current directory

```
for file in *; do
  Contents of $file
  echo "--------------"
  cat $file
  echo
```

```
done
```

Want all files starting with a dot too?

```
for file in $(ls -A); do
  echo "Contents of $file"
  echo "--------------"
  cat $file
  echo
done
```

Iterating over numbers

```
$ echo {1..9}
1 2 3 4 5 6 7 8 9
```

```
for i in {1..9}; do
  touch file-$i
done
```

if statements

```
if command; then
  do_this_command
else               # can be omitted
  do_other_command #
fi
```

If the command's return code is 0 execute what is below the `then`, else, below the `else`

The next example: get all files with a certain content in the current directory.

```
echo "some content" > some_file
for file in *; do
  if grep "some content" &> /dev/null
  then
    echo $file
  fi
done
```

test

Testing arbitrary things, like equality of strings, is done with `test` (= `[[ = [`) Check the man page for everything you can test.

```
if [[ 5 = 5 ]]; then
  echo "Equal :)"
```

```
else
  echo "Not equal :("
fi
# Equal

if [[ 5 = 05 ]]; then
  echo "Equal :)"
else
  echo "Not equal :("
fi
# Not Equal

if [[ 5 -eq 05 ]]; then
  echo "Equal :)"
else
  echo "Not equal :("
fi
# Equal

for file in *; do
  if [[ -d $file ]];then
    echo $file is a directory
  elif [[ -f $file ]];then
    echo $file is a normal file
  else
    echo $file is something special
  fi
done
```