Shell Exercise Book

Based on material http://software-carpentry.org
Used with permission

Introduction to the shell

`[user@machine] [directory] $` is the shell prompt

- Means the shell is ready to receive our instructions
- Do not type the **`$`** in these examples, only what follows
- Shell stars in our home directory:

```
$ /c/Users/<username>   # Windows

$ /Users/<username>     # Mac

$ /home/<username>      # Linux/Cygwin
```

`#` is a comment. Everything until end of line is ignored by the shell

whoami and **pwd**

```
$ whoami

$ pwd       # print working directory
```

- **/** (slash) is the root of file system
- Subsequent slashes delimit directories on the file path
- **/Users/john** means: “From the root of the file system, go into the Users directory, then john/”
- Unlike Windows, where each drive has its root (C:\ and D:\), in UNIX **/** the single highest root of the file system

ls

```
$ ls          # list directory contents
```

- Without arguments, lists current directory

```
$ ls /
```

- **/** is an argument: what to process

```
$ ls -F     # classify listing: / to dirs, * to executables
```

- **-F** is an option: how to process

Options and Arguments

- By convention, options start with a dash (**-**), followed by single letter
- GNU also has long names: double dash (**--**) followed by word(s)

```
$ ls --classify  # same as -F

$ ls -F /        # combines options and arguments

$ ls --color -F  # more than one option, mixed style
```

- Not all commands follow this convention
- Nor are all options available on all platforms

mkdir

```
$ mkdir tutorial
```

- The shell searches for a program called **mkdir**
- Passes the argument **tutorial** to **mkdir**
- **mkdir** does not print any output

```
$ ls -t   # shows by most recent modification time first

$ ls -r   # reverse order

$ ls -lt  # most recent last
```

cd

- Change the directory the shell is in
- Two special directories

  `.` current directory

  `..` parent directory

```
$ cd tutorial/

$ cd ..

$ cd              # without arguments, navigates to home
```

Exercises

1) Navigate to the root of the file system
2) List the directory's contents of root
3) What happens if we **`cd ..`** from root? What does this mean?
4) Navigate back to your home directory
5) List the directory's contents by reverse modification time
6) Create a directory called **exercises** for the subsequent exercises this tutorial
7) Navigate inside the newly created **exercises**
8) Verify your work by printing the current working directory
9) List the contents of the **tutorial** directory from within **exercises**
10) Return to the **tutorial** directory from **exercises** with a single command

tutorial/fish.txt

```
Date,Species,Count
2015-08-29,marlin,3
2015-08-29,shark,1
2015-08-29,tuna,15
2015-08-30,shark,2
2015-08-30,flying fish,1
2015-08-30,cod,4
2015-08-31,turtle,1
2015-08-31,marlin,2
2015-08-31,shark,1
```

- Create a text file called fish.txt inside the **tutorial**
- Paste the content on the left into **fish.txt**
- We will be working with the file for the rest of this tutorial

wc

```
$ wc fish.txt   # wordcount: lines, words, characters
```

- Delimits on spaces

  -l: lines

  -w: words

  -c: characters (bytes)

Omitting a filename

```
$ wc      # seems stuck, but we can actually type:

this is a test

^D

    1   4   15
```

- Expects input from STDIN (standard input), typically the keyboard

^D: End-of-file (EOF) signal. ^ means Ctrl key

^C: Interrupt signal. Aborts process

Examining file content without an editor

```
$ head [-n] fish.txt    # list first n lines

    -n: 10 by default

$ tail [-/+n] fish.txt # list last n lines

    +n: start listing from offset

$ cat fish.txt          # concatenate file(s)

$ cat fish.txt fish.txt
```

Redirection and piping

```
$ head -8 fish.txt > temp.txt
```

- Send the output of the **head** command to **temp.txt** instead of terminal

```
$ head -8 fish.txt | tail -4
```

- Send the output of the **head** command as input to **tail**

```
$ head -8 fish.txt | tail -4 > | wc > temp.txt
```

- Redirection and pipelining can be combined

Exercises

Save the text on the right into a file called ***hello.cpp*** in ***exercises/***

1) **WITHOUT** opening an editor, extract the entire function *say_hello()* into a new file, *say_hello.txt*
2) How many words and characters are there in say_hello.txt?
3) **WITHOUT** opening an editor or creating a temporary file, concatenate the file *fish.txt* with *say_hello.txt* onto the screen
4) How many lines and characters are there in this concatenated output?
5) Copy contents of *say_hello.txt* to a new file, *function.txt* (**WITHOUT** using **cp**)

```
#include <iostream>

void say_hello()
{
  std::cout << "hello";
  std::cout << std::endl;
}

int main()
{
  say_hello();
  return 0;
}
```

cut

```
$ cut -d , -f 1 fish.txt   # extract columns from data file
```

-d: delimiter (separator) between columns

-f: field number(s) to return

```
$ cut -d , -f 1,3
```

- Selects multiple columns, which needn’t be adjacent

```
$ cut -d , -f 1-3
```

- Selects a range of columns

sort and **uniq**

```
$ cut -d , -f 2 fish | sort
```

-n: numeric (default is lexicographical)

-r: reverse

-V: version sort

```
$ cut -d , -f 2 fish | sort | uniq -c
```

-c: count

history and **rm**

```
$ history

$ history > history.txt

$ rm history.txt # remove a file

    -i: interactive, prompt before deleting

    -f: force, do not prompt
```

- No trash bin with shell. **rm** is permanent

Exercises

Save the text on the right into a file called ***animals.txt*** in ***exercises/***

1) Create a pipeline that extracts the animal names and counts the number of occurrences of each animal
2) Using the same pipeline as above, what do the options **`-u`** and **`-d`** do in **`uniq`**?
3) Create a pipeline that extracts the day field from the date column. Display how many entries we have on each of the days (the output should be: 3 on the 5th day, 3 on the 6th day, 2 on the 7th day)

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,racoon
2012-11-06,rabbit
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

grep

```
$ grep marlin fish.txt     # print lines matching pattern
```

- Global Regular Expression Print

  -i: case insensitive

  -v: inverted search (lines that)

  -h: suppress headers (filenames)

  -e: expression (escape ‘-’ in pattern)

diff

```
$ diff file1 file2
```

1,6c1,10 lines 1 to 6 in the first file changed into lines 1 to 10 in the second

< lines in the first file that are not in the second

--- end of first file, start of second

> lines in the second file that are not in the first

Looping

- Programming constructs to repeat operations multiple times

```
$ for data in dmitri_2015.txt nicole_2015.txt
> do
> wc $data
> done
```

- Prompt changed from **$** to **>**
- Means the shell expects rest of command on next line(s)

Anatomy of a for-loop

`for`	introduces the loop
`variable-name`	a variable to iterate over a collection
`in`	introduces start of the collection
`do` and **`done`**	designate start and end of the loop
`$variable-name`	references to the current item in the iteration

- Called a ***foreach*** in some languages

Some keyboard shortcuts

Ctrl+A	Move cursor to start of line
Ctrl+E	Move cursor to end of line
Alt+F	Move one word forward. Esc instead of Alt on Mac
Alt+B	Move one word backwards. Esc instead of Alt on Mac
Alt+D	Delete one word forward
Alt+Backspace	Delete one word backwards