Introduction to Scripting using bash

Scripting versus Programming (from COMP10120)

You may be wondering what the difference is between a 'script' and a 'program', or between the idea of 'scripting languages' or 'programming languages'. It's quite difficult to pin down exact meanings for these, since their use has shifted over time and different people use the terms to mean subtly different things. Scripting languages and programming languages both allow people to create sequences of instructions for a computer to execute. Generally speaking when people refer to scripts or scripting languages they are referring to mechanisms for automating tasks rather than for performing complex computations. So if you wrote something that once a month deleted any files that ended with **`.bak`** from your filestore, you would probably use a scripting language, and most likely think of it as a script. If you were to write a new user friendly desktop publishing software, you'd probably use a programming language and think of it as a program.

Regardless if work as scientist or an engineer in the future, in virtually any discipline computers and programs are used to carry out whatever task. This can involve quite a complex collection of different programs and software tools and scripts allow you to automate many task. This will boost your productivity (and eventually your salary), allows to reproduce tasks, repeat tasks and will help you to carry out your tasks with less errors. Consequently, most programs and tools used by scientists and engineers can be operated through scripts directly from a command line window, such as bash.

Bash is way more powerful than what is possible with the shell that comes with Windows (**Note:** for those of you that had problems with the **CS VM**, please look into **Cygwin)**.

The idea of this introduction is to give you some guidance into bash script programming. However, you should explore the tasks and commands on your own. Look into the man pages or search for online documentation to find out more! Take the challenge, some tasks can be more fun than solving a Sudoku!

Research

You are going to make use of "bash". In an xterm, type "**`man bash`**" and please spend a few minutes exploring the different sections of that documentation.
Inside the man-program, you will find a "**`:`**" at the bottom left of the output window. You can type commands here. Please try the following: (you don't type the ":")

`:h`	shows you the help, press q to leave help
`:-I<Return>`	Ignore CaSe when searching
`:/variable<Return>`	search for the word "Variable"
`:n`	jumps to the next search match
`:N`	jumps to the previous match

`:P` jumps to the beginning of the bash man page

`:/metacharacter<Return>`
search for characters that have special meaning

Try using google to find out about "bash" - e.g. look for an introduction or tutorial. (I think the simplest is v1.05 of the "**Linux Shell Scripting Tutorial**" by Vivek Gite - but where this refers to "vi", please use your preferred editor e.g. "nedit" or "gedit")

Task 1 - A few one-liners

For this task, consider you are a course leader and you have to manage a course with many students. Begin this task by creating a directory "**`scripting`**" in your home directory. Change into the new directory and download our student list (called Turing) from Blackboard.

Check our student list (without modifying the file) and you will see that we have very prominent students and you can find more about them on: http://en.wikipedia.org/wiki/Turing_Award
The first line is a comment (indicated by a "**#**" in the beginning of the line).

First, filter out the comment. Use the man page for grep to solve 'your task':

```
>cat Turing | grep 'your task!' | less
```

Tell us how many students are in the course list ("**`wc`**" is for word count):

```
>cat Turing | grep 'your task!' | wc -l
```

Because our "students" are very busy, only 11 showed up in the first lecture. Create a file "**`week1`**" that contains the first 11 students from "**`Turing`**". This file will serve as an attendance list for a further step.

```
>cat Turing | grep 'your task!' | head -n 11 > week1
```

As we need this later, repeat the process for the first 21 students for "**`week2`**".

Print the students that haven't shown up in week1:

```
>cat Turing week1| grep 'your task!'| sort | uniq -u
```

Build this command up one instruction after the other (before piping the output to the next instruction) and check the output! This is also the way to develop such one-liners.
Have you understood the trick? If not, check what "**`sort`**" and "**`uniq`**" do.

Check **`week1`** and print how many students attended the week 1 lecture.

```
>wc -l week1
```

The output of this command is "**`11 week1`**". So **`wc -l file`** displays also the input file name. There is no switch (another word for option) to turn this

behaviour off, but you won't see the filename, when the input comes from standard in (you remember the Linux introduction last week?). Try:

```
>cat week1 | wc -l
```

and (mind the "<"):

```
>wc -l <week1
```

We can store the output in a variable such that we can use it later without running the **wc** command again. Please execute the following:

```
>all_students=$(cat Turing | grep -v 'your task!' | wc -l)
>this_week_students=$(cat week1 | wc -l)
```

The syntax to store the output of a command in a variable is **variable=$(my_command)**, where the output of **my_command** will be stored in **variable**. Note that we used variables already in our Linux introduction. Remember that you can print your user name with:

```
>echo $USER
```

Or try this one:

```
>while sleep 1; do echo width:$COLUMNS height:$LINES; done
```

Now change the size of your shell window while the last line is executed. You can use this, for example to do some pretty printing for different window sizes. There are a few more variables: search for " Variables in Shell".

OK back to our course management. You can display both count values with:

```
>echo All: $all_students, This week: $this_week_students
```

Tip: for faster typing, you can use the tabulator key to extend known variable names: **>echo All: $all<TAB>, This week: $this<TAB>**

You can even compute and print how many students are missing.

```
>echo Missing: $[$all_students - $this_week_students]
```

So the variables inside the **$[...]** expression are actually treated as numbers (and not as text) and the expression is then arithmetically evaluated.

If you want, you can assign the result of the expression to a variable:

```
>missing_students=$[$all_students - $this_week_students]
>echo Missing: $missing_students
```

Instead of variables, you could even use directly commands (between two "**`your command`**"):

```
>echo Missing: $[`wc -l < Turing` - `wc -l < week1` -1]
```

Hopefully all this wasn't too fast. If you find things too easy, play around with the commands and expressions you just learned.

Take a breath (may be a small break) and try to understand all the magic you just did.

Task 2 - more one-liners

We will now explore our Turing Award winners a bit more.
Try the following:

```
>cat Turing | grep "199[1,2,3]"
>cat Turing | grep "199[1-3]"
>cat Turing | grep "[1991,1992,1993]"
```

does not work because we have to consider individual characters/digits.
But this will do the job:

```
>cat Turing | egrep "1991|1992|1993"
```

Try not to get confused with the **`egrep`** command. It changes the behaviour of grep such that we can use **`egrep "pattern1|pattern2|..."`**. Where we get the line printed if **`pattern1`** OR **`pattern2`** OR... occurs in the actual line.
In all our examples, you could have used egrep instead of grep without seeing a difference.

The Turing Award winners are ordered by the year of the award. Use **`sort`** to change the order from youngest first to oldest first:

```
>cat Turing | grep -v "#" | sort
```

Figure out (and try) how to tell **`sort`** to sort in reverse order!

Now try the following lines and compare the output:

```
>cat Turing | grep -v "#" | sort -k 1
>cat Turing | grep -v "#" | sort -k 2
>cat Turing | grep -v "#" | sort -k 3
>cat Turing | grep -v "#" | sort -k 3 -k 2
```

Note that all commands will deliver something different!
You should see that **`sort`** interprets the file in several columns that are separated by white space symbols (here the blank " "). and with "**`-k`**" you tell **`sort`** which column you want to sort.

You will find for the last **`grep`** command that "(1981)Codd, Edgar" is not sorted as expected. Think about what could cause this and use **`nedit`** to verify this in the file "**`Turing`**".

Now we want to sort by the family name, but sort sees not only the family name but a concatenated field consisting of the year and the family name.
If you haven't recognised this, rerun the for commands (use the up arrow key to pick the commands from the history)

So we have to split date and family name. This can be done by replacing the ")" with a blank " " with the help of the translate command **`tr`**:

```
>cat Turing | grep -v "#" | tr ")" " "
```

With this, you should be able to sort correctly by family name and first name on your own.

Our last command will be sed, which is the *stream line editor*. This command is incredible complex and powerful and it can take years to become a master in this command. So don't get scared. Look in the following examples and see what sed is doing. Manipulate and play with the examples and try to understand what is going on:

```
>cat Turing | sed 's/ /test:/g'
```

The **sed** commands works as follows: the **s** in the beginning of the parameter string is for *substitute* operation and will replace the pattern encapsulated by the first two "**/**" with the second pattern (here " " will be replaced with "**test:**". The final **g** tells **sed** to do this *globally* for all occurrences of the first search pattern. Use **1**, **2**, (and so on) instead of **g** and observe the difference. This is quite cool, isn't it?

The first pattern can be actually an expression. We learned this last time for grep. For example we could delete the date by replacing anything (which is given by "**.***") from the beginning of the line until the ")" by nothing:

```
>cat Turing | sed 's/.*)//g'
```

Now try this:

```
>cat Turing | sed 's/,/)/g' > test
>cat test | sed 's/.*)//g'
>cat test | sed 's/.*(....)//g'
```

We see that "**.*)**" will replace the largest possible pattern, so everything from the beginning including the last "**)**" will get replaced in test. However, by giving a more precise pattern (the four dots "**....**" actually wildcard the four characters/digits of the year), we can delete the year in the file test correctly.

So far, we haven't thought about how to find a pattern that starts at the beginning and for more complex inputs our simple approach might fail. However, you can use the "**^**" symbol to match the beginning of a line:

```
>cat Turing | sed 's/^/Beginning:/g'
>cat Turing | sed 's/^.*(....)//g'
```

There is virtually no string manipulation you cannot do with sed and with the few examples you learned so far, you can already perform quite a lot of tricky text manipulation tasks.

For most common tasks, you will find good examples in the Internet. For example, search for: "sed how to match on the end of a line". Typically the answers on stackoverflow are of very good quality.

Or search for "sed how to do an arithmetic substitution". This would be more difficult, but with the tricks you learn today, you are already able to, lets say, add 42 on the year of each of our students.

Task 3 - our first bash script

Write a bash script in the **scripting** directory that automates the student management that is needed for monitoring attendance each week. Name the script "attendance". We assume that we will get for each week an attendance file (week1, week2, and so forth) exactly as we have used this in Task 1.

Please note that you have to set the execute flag for the attendance file:

>chmod 700 attendance

The three digits are actually an octal number (a number that can have the digits 0-7 instead of 0-9, which is decimal). The digits set the access privileges for yourself (left digit), your group - which are all FY students (middle digit) and for everybody else (right digit). Octal numbers are used to group three binary digits together. This is similar to how you sometimes group three digits together in a long decimal number to groups the thousands, millions, billions, etc.
For executing a file add 1 to the digit, for writing 2 and for reading 4. So the 700 means that only you can execute, write, and read the file (1+2+4=7).

You should call attendance with the specific weekly attendance file name as a parameter, like for example:

>attendance week1

The result should be a report file with the given file name as a prefix (ideally week1.report) and a formatted attendance list (week1.students) with the student names, but without the date and the whole output sorted by family name followed by the first name.

Start with this example for attendance and try it out:

```
#!/bin/bash
echo This is the report for $1 > $1.report
echo ---------------------------- >> $1.report
# this is a comment and will not be executed
echo                               >> $1.report
echo Total number of students:     >> $1.report
echo                               >> $1.report
echo Students attending:           >> $1.report
echo                               >> $1.report
echo Students missing:             >> $1.report
echo                               >> $1.report
echo Unknown students:             >> $1.report
```

The fist line (**#!/bin/bash**) is something special and ensures that the bash is used for executing our script. Follow the link for more information:
http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_02_02.html
(this one is linked from "Bash Guide for Beginners" by Mendel Cooper)

The variable $1 gives you access to the first parameter that you passed to our script (which is, for example, "week1"). You could have more parameters that you can access with $2, $3, ... in the order they are specified.

Pick what you need from the previous two tasks and complete the attendance script that will generate the two output files.

The last line in the example script should print the number of unknown students. Those are students that are in our weekly list, but not in our global student list (i.e. **`Turing`**).

If you are done with your script and have tested it, please call one of our GTAs to show your work and get marked.

Fun Tasks

The attendance script is the mandatory delivery for today. However If you are done, early, you could consider:

- Write a script that computes the statistics of how often each individual student showed up over the whole course unit.
  Hint: you can use the con**`cat`**enate and **`sort`** trick and use **`uniq`** to compute a histogram.
- Write a bash script that prints the file Turing line-by-line in a loop
- Based on this, try to add 42 on each year in the file
- Consider the following example: **`sort -k 1 -r`**
  The command uses two options whereof one consists of two parts "-k 1" and where one is a simple switch "-r". Write a bash script that allows you to optionally specify such switches in any order.
- Write a bash program that runs in one shell that generates a directory with a sensible name (e.g. year_month_date_hours_minutes_seconds) and that stores a copy of our attendance script whenever the attendance script is saved. (so you are basically creating a very simple log over all changes you committed when saving the script.