Teacher's Assistants

Robert Kaplow
Wednesday, 9:30 to 11:30
T.A. room, Trottier, 3rd floor

Jun Wang
Friday, 14h00 to 16h00
T.A. room, Trottier, 3rd floor

Michael Hawker
Monday, 14h30 to 16h30
McConnell, Room 322
Command line text editors allow you to create/edit files at the command line. Several text editors are available.

- **vi** is one of the original text editor available on Unix. It's very difficult to use and learn. However, its very powerful and available on every Unix machines.
- **pico** is a simple text editor based on the **pine** mail client. It's very easy to use, and is available on most Unix machines.
- **emacs** is a very popular and powerful. Considering the number of features it has, it should be considered a heavy weight client.

You can also use graphical text editors, such as bluefish, gedit or jedit.

As a long term investment, I highly suggest you learn **vi**.
Example of Text Editors

Text based, console
- Vi
- Emacs
- Pico
- Ed
- JStar / Jove
- Edit (dos)

Graphic based, GUI
- Xemacs
- Bluefish
- Gedit (Gnome)
- Kate (KDE)
- Jedit (java)
- Notepad (windows)
Several Unix commands and editors allow you to search on text patterns.

These text patterns are known as regular expressions (regex).

Examples of regular expressions include:

- Text starting with the letter “a” and finishing with the letter “z”.
- Text with at least one number, but not starting with the letter “a” or “b”.
- Text with a letter repeated three times in a row.
- Text contains the string “abc” exactly three times.
Take a look at the Regex Syntax quick sheet.

- Literal characters are combination to represent special characters.
- Character classes are combination to represent groups of characters.
- Repetition indicate how often a character should be appear to be a match.
- Anchors determine where the matching string must be found.
grep, sed and awk

- grep [options] string file
  - search for occurrences of the string.

- sed [options] file
  - stream editor for editing files.

- awk [options] file
  - scan for patterns in a file and process the results.
**grep** is used to search for the occurrence of a regular expression in files.

Regular expressions, are best specified in apostrophes (or single quotes) when use with grep.

Some common options include:

- `-i`: ignore case
- `-c`: report only a count of the number of lines containing matches
- `-v`: invert the search, displaying only lines that do not match
- `-n`: display the line number along with the line on which a match was found
- `-l`: list filenames, but not lines, in which matches were found
Examples of grep

Consider the following text file:

Alex
Marc
Micheal
Ting
Juan
Jeremy
Jessica
Yannick
Nicolas
Jean-Sebastien
Nadeem
Examples of grep (cont.)

- Grep for a specific string . . .

  [adenau][rogue][~/cs206] grep 'Je' demo.txt
  Jeremy
  Jessica
  Jean-Sebastien

  [adenau][rogue][~/cs206] grep -n 'Je' demo.txt
  6:Jeremy
  7:Jessica
  10:Jean-Sebastien

  [adenau][rogue][~/cs206] grep -c 'Je' demo.txt
  3
Examples of grep (cont.)

- Grep for vowels...

  ```bash
  [adenau][rogue][~/cs206] grep -i '^[aeiouy]' demo.txt
  Alex
  Yannick
  [adenau][rogue][~/cs206] grep -i '[aeiouy]$' demo.txt
  Jeremy
  Jessica
  [adenau][rogue][~/cs206] grep -i '[aeiouy]\{2\}' demo.txt
  Micheal
  Juan
  Yannick
  Jean-Sebastien
  Nadeem
  ```
Examples of grep (cont.)

- **Grep for specific characters** . . .
  
  [adenau][rogue][~/cs206] grep -i '^.e' demo.txt
  Jeremy
  Jessica
  Jean-Sebastien

  [adenau][rogue][~/cs206] grep -i '^.e\|a.$' demo.txt
  Micheal
  Juan
  Jeremy
  Jessica
  Nicolas
  Jean-Sebastien
When to use grep

- Grep is a useful tool to find specific strings.
  - Outlining all the errors in a log file.
  - Finding a specific string in a collection of source files.

- It becomes an even more powerful tool when combined to other utilities.

```bash
[adenau][rogue][~/cs206] ps -e | grep 'java'
14256 pts/1 00:18:30 java
21395 ? 00:00:08 java
11218 pts/4 00:03:51 java
```
A shell program (or script) containing a series of shell commands.

- The first line of the script should start with `#!` which indicates to the kernel that the script is directly executable.
- You immediately follow this with the name of the shell, or program (spaces are allowed), to execute, using the full path name.

Different languages can be used to script (sh, bash, perl, python, ruby, etc).

To set up a Bourne shell script the first line would be:
```
#! /bin/sh
```

You also need to specify that the script is executable by setting the proper permissions on the file.
```
% chmod +x shell_script
```
There are three kinds of variables in a shell script:

- Environment Variable: these variables are used to customize the operating system and the shell to your needs.
- User-created: these variables are created by the script itself.
- Positional Parameters: these variables store the parameter used to start the script.
Positional Variables

- `#$`: number of arguments on the command line
- `#$-`: options supplied to the shell
- `#$?`: exit value of the last command executed
- `#$!`: process number of the last command done in background
- `#$n`: argument on the command line, where n is from 1 through 9, reading left to right
  - `#$0`: the name of the current shell or program
  - `#$*`: all arguments on the command line ("$1 $2 ... $9")
  - `#$@`: all arguments on the command line, each separately quoted ("$1" "$2" . . . "$9")
The following script will print out the positional variables:

```bash
#!/bin/sh
echo "#$:" $#
echo ' $-:' $-
echo ' $?:' $?
echo ' $$:' $$
echo ' $!:' $!
echo ' $3:' $3
echo ' $0:' $0
echo ' $*:' $*
echo ' $@:' $@
```
Shell Scripts

- A shell script runs from top to bottom.
- If statements and loop can be used to alter the control flow.
- You can also create functions.
- The # character is usually used to denote a comment.
- The #! at the start of the script indicates which program should execute/interpret the script.
- Unlike other programming languages, scripts are sometime sensitive to extra spaces.
The following script gathers information about the system and stores it in a file specified at the command line.

```
#!/bin/sh
uname -a > $1
date >> $1
who >> $1
```

The output was as follows:

```
[adenau][rogue][~/cs206] ./info.sh output.txt
[adenau][rogue][~/cs206] cat output.txt
Linux rogue 2.6.12-gentoo-r4 #1 SMP ...
Thu Aug 10 10:57:38 EDT 2006
adenau pts/0 Aug 10 08:04 (dz2.cs.mcgill.ca)
```
The `read` command allows you to read a string from STDIN.
That string is then stored in the specified variable.

```bash
#!/bin/sh

echo "What is your name?"
read name
echo "Your name is $name."
```
Arithmetic Operations

- The shell was never designed for numerical work.
- To do mathematical (integer) operations, you can use the `expr` command.
- The following example script adds two numbers passed at the command line and outputs the answer to `STDOUT`.

```sh
#!/bin/sh

sum=`expr $1 + $2`

echo $sum
```
Before discussing control statements (if, for, etc), we need to check out the `test` command.

The test command is used to evaluate an expression, or in our case, a condition.

- Although shells do contain operators to test a condition, they are not as versatile and universal as test.

The test command can evaluate condition at the file, string or integer level.
File Tests

- **-r**: true if it exists and is readable
- **-w**: true if it exists and is writable
- **-x**: true if it exists and is executable
- **-f**: true if it exists and is a regular file
- **-d**: true if it exists and is a directory
- **-h** or **-L**: true if it exists and is a symbolic link
- and many more . . .
-z string : true if the string length is zero
-n string : true if the string length is non-zero
string1 = string2 : true if string1 is identical to string2
string1 != string2 : true if string1 is non identical to string2
string : true if string is not NULL
Integer Tests

- `n1 -eq n2`: true if integers n1 and n2 are equal
- `n1 -ne n2`: true if integers n1 and n2 are not equal
- `n1 -gt n2`: true if integer n1 is greater than integer n2
- `n1 -ge n2`: true if integer n1 is greater than or equal to integer n2
- `n1 -lt n2`: true if integer n1 is less than integer n2
- `n1 -le n2`: true if integer n1 is less than or equal to integer n2
Logical Operators for Tests

- !: negation (unary)
- -a: and (binary)
- -o: or (binary)
- (): expressions within the () are grouped together. You may need to quote the () to prevent the shell from interpreting them.