Lab #2

Physics 91SI Spring 2013

Objective: Some more experience with advanced UNIX concepts, such as redirecting and piping. You will also explore the usefulness of Mercurial version control and how to write short "programs" using UNIX commands in the form of shell scripts.

As will become usual for the labs, you will ssh into a Corn machine, where you will do these exercises. You do not need X11-forwarding (`ssh -X`), but it can be nice to have a GUI text editor, and it can't hurt if you have it. Please cd to your `Physics 91 SI` directory, but do not create a `lab2` subdirectory yet - see below.

Part 0: Getting Started

There's one bit of administrivia that needs to be done before using Mercurial. Using a text editor (`gedit` or `nano`, or your choice), open the file `~/.hgrc` (*note the similarity to the ~/.cshrc file from Lab 1*). Paste the following text (replacing with your own username), and save the file:

```
[ui]
editor = gedit
username = John Doe <spamandeggs@stanford.edu>
merge = emacs

[merge-tools]
emacs.args = -q --eval "(ediff-merge-with-ancestor \"$local\" \"$other\"
\"$base\" nil \"$output\")"
```

Also, if you didn't get to the "Aliases and Links" section of Lab 1, we suggest that you follow these instructions to create environment variables for the course directory and your working directory. First, open, ~/.cshrc and find the section titled "Environmental Variables" (*In gedit, just use Ctrl-f*). Add the following lines (making sure the "`si`" in the first line is lowercase):

```
setenv SICDIR "/afs/ir.stanford.edu/class/physics91si"
setenv SI "~/physics91si"
```

Save the file, and when you next log in you'll be able to access the course directory and your working directory using these environment variables (i.e. `cd $SICDIR` or `ls $SI/lab1/`).

Part 1: Using Mercurial

survey.py

`cd` into your `Physics 91 SI` directory; you're going to create your `lab2` directory by *cloning* the Mercurial repository located in `/afs/ir/class/physics91si/src/lab2`. Do this by running the command

```
hg clone /afs/ir/class/physics91si/src/lab2 lab2
```

Or, if you created an environment variable last class, you can do something like

```
hg clone $SICDIR/src/lab2 lab2 .
```

Go into `lab2` and run `ls` and `hg log` to see what files were saved in the directory and when they were added/edited. You can also use `hg status` to see which files Mercurial is tracking.

The first file you're going to interact with is `survey.py`, a little Python program to ask you some quick survey questions. Go ahead a run it by typing `python survey.py` and answer its questions. Now, from the `hg log`, figure out what *changeset* you're in right now and which ones survey.py was added/edited in.

How can you figure out what edits were made previously? You could revert to each version and look at the file to try to find the difference, but Mercurial has a nice command, `hg diff`, that makes this a lot easier. To find out how to use it, use Mercurial's own help system: `hg help <command>`

After you've figured out what each of the two edits were, revert back to the original survey.py and see if you can get away with "lying" to the survey. (This does involve understanding some Python code, but I promise, the code that matters should be pretty straightforward to figure out). You should also try creating a file with the contents "no <enter> yes" (2 lines) and save it (say as answers.txt), and run

```
python survey.py < answers.txt
```

as an example of redirecting a file to `stdin`.

define_ions.py

Now we're going to play with `define_ions.py`. Go ahead and run it, note that it takes 5 numerical arguments, so type something along the lines of

```
python define_ions.py 5 4 6 7 8
```

and go ahead and look at the error message that gets printed out. Oh no, somebody gave you a program that doesn't work (or maybe you were editing something a while ago and accidentally did something and now it doesn't work anymore)! But they're sure that it was working before; perhaps a working version was saved in one of the older changesets, before the "accident."

Find a way to get back to the working version, and also find a way to search for the accidental change that "broke" the program. *(Hint: you want to use `hg update` here. Type `hg help update` for more information)* Think about when it's okay to just revert back to an old version and commit and when it's better to search for the accidental change, fix it, and then commit. Maybe their last edit included some important changes, on top of the one accidental change? (doesn't matter for this example, but important to remember for your own use) The `hg diff` command may be useful here.

Go back to the "tip", fix the mistake, and use `hg commit` to save a new "snapshot" of the files, and include a helpful commit message, briefly explaining what happened and what you did.

Run the program again now and see what happens, but this time run it with these arguments:

```
python define_ions.py 8 5 9 3 1000
```

Wow! That's a lot of text! You can't even scroll up to the top in your terminal! If only there was a way to *redirect* all of that output to a file that you can just open up in a text editor to view...try that out. (Alternatively, try piping the output straight to `less`)

Once you've done that, go ahead and add the output file to your Mercurial repository and make another commit (remember to give a concise, useful commit message!)

Part 2: Piping and Command Substitution

So let's start off with a simple example for you to see right on your own screen. Try `ls | grep .py` and `ls | grep .py -v` (the -v in the second one tells grep to *exclude* all of the lines that match the pattern). You can also try `ls | grep .pdb` and then run `rm` `` `ls | grep .pdb` `` (careful! those are backquote characters - the key should be near Esc), which will remove all of the files that match the pattern .pdb.

Now you're going to work through a more complicated example, stringing together a few different commands to eventually find the name and size of the largest file in your home (~) directory, so cd over there.

Start off with `du -aS` (*remember, UNIX is case-sensitive!*), which will print out the name and size of all of the files in your home directory and its sub-directories in a way that you'll want (if you want, go ahead and check out the man page for `du` to learn more about the `-a` and `-S` options).

Now, using pipes (" | ") , use the following commands to parse the output of `du -aS`:

```
sort -nr
cut -f2
head -5
```

Add them one at a time to build up a chain, and see how the output changes. What does each of these commands do? If you have time, check the `man` pages for each to see what each option flag does.

Now add the `tail` command to your string of piped commands and fiddle around with the options given to `head` and `tail` so that your output is just the location of the largest file (not directory, files (usually) have a suffix like .doc). Your output might also contain an error message talking about not having access to something.

That error message is annoying, so run your long command again, but this time add a redirect to a file to the end of it. View the output file and notice how the error message wasn't redirected to the file, just the location of your largest file. This is an example of the difference between `stdout` and `stderr`; they may both print to the terminal under normal conditions, but when you use a redirect, it by default only redirects `stdout` to the file.

Now, for the finishing touch, try `du -h` `` `cat output.txt` `` (again, these are backquotes!) to see the size of your largest file in a "human readable" format.

Part 3: Shell Scripting Challenge

We hinted at this with the "Challenge" in Lab #1, and now you'll get a chance to do something a little more useful. Shell scripting is best for when the commands that you need to type out are long and complicated OR they're something you'll do a lot and don't want to write every time. So let's work on an example that serves both: submitting your lab directories when you're done.

Write a shell script, called submit.sh, that will run `hg status`, an `echo` command to remind you to check that you've got everything in the repository, `hg commit` (don't use `-m` option), and then, using `whoami, pwd`, along with `echo` and `cut`, create a file that will contain, on a single line, the full path of your submission directory. Then add a line that uses `hg push DEST` (where `DEST` is the directory name in your file) to submit your work. Finally, add lines that use `rm` to delete any temporary files you created, so that the submission process doesn't leave any side effects.

Your submission directory has the format

```
/afs/ir/class/physics91SI/submissions/jchaves/lab2
```

NOTE: You will need to use redirection to eventually create a file with the above address, but you will need to know the difference between > and >>. Also, there will be some annoyances

with newlines when redirecting to this file, so look into `echo -n` and `tr -d '\n' < FILE > newFILE` (this latter command makes a copy of a file, stripping out newline characters).