98-172 : Great Practical Ideas for Computer Scientists

My Skills R Bourne Again

Grab Some Files

Throughout this lab-itation, we will use some files, which you should grab using the `wget` command:

- `wget http://www.andrew.cmu.edu/course/98-172/examples/gpi_roster`
- `wget http://www.andrew.cmu.edu/course/98-172/examples/permute.py`
- `wget http://www.andrew.cmu.edu/course/98-172/staff.html`

`wget` will get files from the internet and put them in the directory you are currently in which can be useful.

What's a Pipe?

A pipe "|" takes the output of the command before it and feeds it to the input of the command after it. A simple example would be to `cat` the lines of the `roster` and feed the result into `sort`, like this:

```
cat gpi_roster | sort
```

The roster was mostly already sorted; so, instead, let's use `permute.py` first, to demonstrate what `sort` is really about:

```
cat gpi_roster | python permute.py | sort
```

Now (as in lecture), let's introduce `uniq` into the picture, to remove all the phony duplicates:

```
cat gpi_roster | python permute.py | sort | uniq
```

`uniq` can take some useful options. If you use `uniq -d`, then it will only show the duplicates (in this case `a` and `z`). If you use `uniq -u`, then it will only show the unique elements. Here, we want the actual roster; so, use `uniq -u` to filter out `a` and `z`.

Redirection Reminder

Recall that we can use <, >, >>, <<, 2>, etc. to redirect input, output, and standard error. In this case, we might want to save our corrected roster into a file; so, run the commands:

```
cat gpi_roster | python permute.py | sort | uniq -u > corrected_roster
cat corrected_roster
```

Now, we've saved the fixed roster into the `corrected_roster` file. Note that, instead, we could have run the command:

```
python permute.py < gpi_roster | sort | uniq -u > corrected_roster
```

This would have done the same thing, except via standard input rather than `cat`!

A Little More Greping

Some of you got an e-mail that said you weren't coming to lab-itation. The way Adam was able to figure that out was by testing which shell was your default. At this point, you are almost ready to do that audit yourself. The machines in Gates can be directly `ssh`ed to; so, start this section by `ssh`ing to a machine of the form `ghcXX.ghc.andrew.cmu.edu`, where `XX` are two digits (they only go up to around 50). Once you have done that, try using the finger command:

```
finger $USER
```

In the second line of the input, it gives you the home directory of the user you `finger`ed and their default shell. Grep for this line in the output by doing:

```
finger $USER | grep "Shell:"
```

Grep is a much more powerful tool than it might seem at first, and we will go into more detail about it later in this lab-itation.

Bash For Loops

Knowing how to do a for loop in your shell can be really useful, because you often want to do something "for all files" or "for every line in a file" and `bash` makes this really easy to do. First, the syntax of a `for` loop in bash:

```
for VARIABLE_NAME in SOMETHING; do <whatever you want to do>; done
```

Now, a few examples. Suppose we wanted to `cat` every file in my current directory:

```
for file in *; do cat $file; done
```

It is probably worthy of note that we can do the same thing using `cat *`
Now, what if we wanted to only print the first line of every file in my current directory:

```
for file in *; do head --lines=1 $file; done
```

This time, doing `head --lines=1 *` will give me something different!

Lastly, what if we wanted to `echo` each line in my `corrected_roster`:

```
for line in `cat corrected_roster`; do echo $line; done
```

The only new and tricky part of that command is the backticks (the non-shifted version of the tilde key). They tell `bash` to replace whatever is inside them with what it evaluates to, which we needed in this case to get all the lines in the file.

Figuring Out Everybody's Default Shell

Now you have everything you need to figure out the default shell of everyone in the class. Replace the `echo` in the for loop with the `finger` and `grep` from before, and you should be done. Just as a sanity check, if you add "`| grep "bash" | wc -l`" to the end of your command, it should spit out a number around 77 instead, because of the 91 students in the class, around 77 of them are using `bash`.

Using Grep for Real

In this section, we will use grep a little more:

Start by doing `cat staff.html | grep "td"`
Note how we get a bunch of lines that don't start with "td"; this is because we didn't tell grep to look for only for lines starting with that string–it searched for lines containing "td". To search for lines starting with "td", do:

```
cat staff.html | grep "^td"
```

(there actually aren't any in `staff.html`!)

Analogously, we might want lines that **end** in a certain string (like "br>", since this is an html file). To do this, we would use:

```
cat staff.html | grep "br>$"
```

Suppose, instead, we want only the lines that do not start with "<". Then, we would do:

```
cat staff.html | grep -v "^<"
```

We could get rid of lines with "var" as well by chaining greps:

```
cat staff.html | grep -v "^<" | grep -v "var"
```

Character classes (using the [and] characters) allow us to search for particular types of characters. For instance, to search for lines containing at least one number, we could do:

```
cat staff.html | grep "[0-9]"
```

We could be more specific and search for lines which set a width to a number by doing:

```
cat staff.html | grep "width=\"[0-9]"
```

Note how the " had to be escaped here. We could be a little more advanced and make sure the width is a bunch of numbers ending in a quote by doing:

```
cat staff.html | grep "width=\"[0-9]*\""
```

(The * matches any number of the thing right before it; so, in this case, the * means "as many 0,1,2,3,4,5,6,7,8,or 9's as you can find.)

We could search instead for `cat staff.html | grep "width=[\"]*[0-9]*[\"]*"` which would allow us to not have the quotes (since * can match zero occurrences of something.

Lastly, one more useful things that grep supports is ".", which will match exactly one occurrence of any (non-newline) character.

Everybody Uses A Little Bit of `sed`

`sed` is a stream editor which uses regular expressions (like `grep`). While the purpose of `grep` is to find things, `sed` will do string processing and **replace** pieces of them.

The basic syntax is as follows: `sed -e "s/<regular expression to find>/<what to replace it with>"`

For instance, suppose we wanted to search for the e-mail addresses of everyone on course staff. We could use `grep` to do `cat staff.html | grep "Email"` However, this isn't exactly what we wanted. There's a few things wrong with it (theres a `<br>` on the end, "at andrew" isn't what we want, etc.). So, we can use `sed` to fix these problems:
`cat staff.html | grep "Email" | sed -e "s/<br>//"` (replace `<br>` with nothing)
`cat staff.html | grep "Email" | sed -e "s/<br>//" | sed -e "s/ at andrew/@andrew.cmu.edu/"`

If you wanted to replace something everywhere it appears in the line, you would add a "g" after the last slash.

To show one more powerful feature of `grep`, let's get the names of all the people on course staff, and rearrange them so that they are in the form "Last, First". First, we note that all of the lines with names of course staff have "`span class="black"`" in them. So, grep for that by using:

```
cat staff.html | grep "span class=\"black\""
```

Then, we want to get rid of everything but the name; so, use what we already know about `sed`:

```
cat staff.html | grep "span class=\"black\"" | sed -e "s/<span class=\"black\"><b>//"
               | sed -e "s/<\/b><br>//"
```

Note that here, we had to escape the "/", because it is a special character for `sed`.

Finally, we get to the special feature. You can use "groups" in `sed` to match pieces of lines and do interesting things with them. So, here, we want to match everything up to the first space and everything after that. To match a group, use `\(<thing to match>\)`. Then, to reference it later, use `\<number of group>`.

So, to make it look how we wanted, we would use:

```
cat staff.html | grep "span class=\"black\"" | sed -e "s/<span class=\"black\"><b>//"
               | sed -e "s/<\/b><br>//" | sed -e "s/\(.*\) \(.*\)/\2, \1/"
```

I know this command looks really complicated, but (1) you should be able to disect every piece of it now and (2) the syntax gets easier to write with practice.

Bash If Statements

Rather than repeating the large list of if parameters, we point you to: `http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html`

You probably won't need bash if statements all that often though.