Unix exercises

Here are some tasks that will introduce how the commands from the previous chapter can be combined to get things done. Sometimes you can get exactly what you need with command-line tools and never need to write a program. Here I show you output on my system, and I encourage you to type (do not copy and paste!) the commands on your system to compare. Note that we are likely to be using different versions of a Unix-like OS, so the implementations of commands like `ls` or `sed` might differ by output or arguments. Be sure to consult your manpage and versions to understand any differences you see.

Find the number of unique users on a shared system

We know that `w` will tell us the users logged in. Try it now on a system that has many users (i.e., not your laptop) and see the output. Likely there are dozens of users, so we'll connect the output of `w` to `head` using a pipe `|` so that we only see the first five lines:

```
[hpc:login2@~]$ w | head -5
 09:39:27 up 65 days, 20:05, 10 users,  load average: 0.72, 0.75, 0.78
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
kyclark  pts/2    gatekeeper.hpc.a 09:38    0.00s  0.05s  0.02s w
emsenhub pts/0    gatekeeper.hpc.a 04:05   14.00s  0.87s  0.87s -bash
joneska  pts/3    gatekeeper.hpc.a 08:25    1:12m  0.16s  0.12s vim results_x2r
```

Really we want to see the first five *users*, not the first five *lines* of output. To skip the first two lines of headers from `w`, we can pipe `w` into `awk` and tell it we only want to see output when the Number of Records (`NR`) is greater than 2:

```
[hpc:login2@~]$ w | awk 'NR>2' | head -5
kyclark  pts/2    gatekeeper.hpc.a 09:38    0.00s  0.07s  0.03s w
emsenhub pts/0    gatekeeper.hpc.a 04:05   26.00s  0.87s  0.87s -bash
joneska  pts/3    gatekeeper.hpc.a 08:25    1:13m  0.16s  0.12s vim results_x2r
shawtaro pts/4    gatekeeper.hpc.a 08:06   58:34   0.17s  0.17s -bash
darrenc  pts/5    gatekeeper.hpc.a 07:58   51:07   0.14s  0.07s qsub -I -N pipe
```

`awk` takes a PREDICATE and a CODE BLOCK (contained within curly brackets `{}`). Without a PREDICATE, `awk` prints the whole line. I only want to see the first column, so I can tell `awk` to `print` just column `$1`:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | head -5
kyclark
emsenhub
joneska
shawtaro
```

```
darrenc
```

We can see that the some users like "joneska" are logged in multiple times:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}'
kyclark
emsenhub
joneska
shawtaro
darrenc
guven
guven
guven
joneska
dmarrone
```

Let's `uniq` that output:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | uniq
kyclark
emsenhub
joneska
shawtaro
darrenc
guven
joneska
dmarrone
```

Hmm, that's not right – "joneska" is listed twice, and that is not unique. Remember that `uniq` only works *on sorted input*? So let's sort those names first:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | sort | uniq
darrenc
dmarrone
emsenhub
guven
joneska
kyclark
shawtaro
```

To count how many unique users are logged in, we can use the `wc` (word count) program with the `-l` (lines) flag to count just the lines from the previous command

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | sort | uniq | wc -l
7
```

So what you see is that we're connecting small, well-defined programs together using pipes to connect the "standard input" (STDIN) and "standard output (STDOUT) streams. There's a third basic file handle in Unix called"standard

error" (STDERR) that we'll come across later. It's a way for programs to report problems without simply dying. You can redirect errors into a file like so:

```
$ program 2>err
$ program 1>out 2>err
```

The first example puts STDERR into a file called "err" and lets STDOUT print to the terminal. The second example captures STDOUT into a file called "out" while STDERR goes to "err."

NB: Sometimes a program will complain about things that you cannot fix, e.g., `find` may complain about file permissions that you don't care about. In those cases, you can redirect STDERR to a special filehandle called `/dev/null` where they are forgotten forever – kind of like the "memory hole" in 1984.

```
$ find / -name my-file.txt 2>/dev/null
```

Count "oo" words

On almost every Unix system, you can find `/usr/share/dict/words`. Let's use `grep` to find how many have the "oo" vowel combination. It's a long list, so I'll pipe it into "head" to see just the first five:

```
$ grep 'oo' /usr/share/dict/words | head -5
abloom
aboon
aboveproof
abrood
abrook
```

Yes, that works, so redirect those words into a file and count them. Notice the use of `!$` (bang-dollar) to reference the last argument of the previous line so that I don't have to type it again (really useful if it's a long path):

```
$ grep 'oo' /usr/share/dict/words > oo-words
$ wc -l !$
10460 oo-words
```

Let's count them directly out of `grep`:

```
$ grep 'oo' /usr/share/dict/words | wc -l
10460
```

Do any of those words additionally contain the "ow" sequence?

```
$ grep 'oo' /usr/share/dict/words | grep 'ow' | head -5
arrowroot
arrowwood
balloonflower
```

```
bloodflower
blowproof
```

How many are there?

```
$ grep 'oo' /usr/share/dict/words | grep 'ow' | wc -l
158
```

How many *do not* contain the "ow" sequence? Use `grep -v` to invert the match:

```
$ grep 'oo' /usr/share/dict/words | grep -v 'ow' | wc -l
10302
```

Do those numbers add up?

```
$ bc <<< 158+10302
10460
```

Find unclustered protein sequences

A labmate wants help finding the sequences of proteins that failed to cluster.
Here is the setup:

```
$ wget ftp://ftp.imicrobe.us/biosys-analytics/exercises/unclustered-proteins.tgz
$ tar xvf unclustered-proteins.tgz
$ cd unclustered-proteins
```

The "README" contains our instructions:

```
The file "cdhit60.3+.clstr" contains all of the GI numbers for
proteins that were clustered and put into hmm profiles.  The file
"proteins.fa" contains all proteins (the header is only the GI
number).  Extract the proteins from the "proteins.fa" file that were
not clustered.
```

If we look at the IDs in the proteins file, we'll see they are integers:

```
$ grep '>' proteins.fa | head -5
>388548806
>388548807
>388548808
>388548809
>388548810
```

Where can we find those protein IDs in the "cdhit60.3+.clstr" file?

```
$ head -5 cdhit60.3+.clstr
>Cluster_5086
0    358aa, >gi|317183610|gb|ADV... at 66.76%
1    361aa, >gi|315661179|gb|ADU... at 70.36%
```

```
2    118aa, >gi|375968555|gb|AFB... at 70.34%
3    208aa, >gi|194307477|gb|ACF... at 61.54%
```

The format of the file is similar to a FASTA file where the ">" sign at the left-most column identifies a cluster with the following lines showing the IDs of the sequences in the cluster. To extract just the clustered IDs, we cannot just do `grep '>'` as we'll get both the cluster IDs and the protein IDs.

```
$ grep '>' cdhit60.3+.clstr | head -5
>Cluster_5086
0    358aa, >gi|317183610|gb|ADV... at 66.76%
1    361aa, >gi|315661179|gb|ADU... at 70.36%
2    118aa, >gi|375968555|gb|AFB... at 70.34%
3    208aa, >gi|194307477|gb|ACF... at 61.54%
```

We'll need to use a regular expression (the `-e` for "extended" on most greps, but sometimes not required) to say that we are looking at the beginning of a line `^` for a `>`:

```
$ grep -e '^>' cdhit60.3+.clstr | head -5
>Cluster_5086
>Cluster_10030
>Cluster_8374
>Cluster_13356
>Cluster_7732
```

and then invert that with "-v":

```
$ grep -v '^>' cdhit60.3+.clstr | head -5
0    358aa, >gi|317183610|gb|ADV... at 66.76%
1    361aa, >gi|315661179|gb|ADU... at 70.36%
2    118aa, >gi|375968555|gb|AFB... at 70.34%
3    208aa, >gi|194307477|gb|ACF... at 61.54%
4    358aa, >gi|291292536|gb|ADD... at 68.99%
```

The integer protein IDs we want are in the third column of this output when split on whitespace. The tool `awk` is perfect for this, and whitespace is the default split character (as opposed to `cut` which uses tabs):

```
$ grep -ve '^>' cdhit60.3+.clstr | awk '{print $3}' | head -5
>gi|317183610|gb|ADV...
>gi|315661179|gb|ADU...
>gi|375968555|gb|AFB...
>gi|194307477|gb|ACF...
>gi|291292536|gb|ADD...
```

The protein ID is still nestled there in the second field when splitting on the vertical bar (pipe). Again, `awk` is perfect, but we need to tell it to split on something other than the default by using the "-F" flag:

```
$ grep -ve '^>' cdhit60.3+.clstr | awk '{print $3}' | \
```

```
  awk -F'|' '{print $2}' | head -5
317183610
315661179
375968555
194307477
291292536
```

These are the protein IDs for those that were successfully clustered, so we need to capture these to a file which we can do with a redirect `>`. Since each protein might have been clustered more than once, so I should `sort | uniq` the list:

```
$ grep -ve '^>' cdhit60.3+.clstr | awk '{print $3}' | \
  awk -F"|" '{print $2}' | sort | uniq > clustered-ids.o
```

The "proteins.fa" is actually a little problematic. Some of the IDs have extra information. If you `grep '^>' proteins.fa`, you will see 220K IDs scroll by, not all of which are just integers. Let's isolate those that do not look like integers.

First we can remove the leading ">" from the FASTA header lines with this:

```
$ grep '^>' proteins.fa | sed "s/^>//"
```

If I can find a regular expression that matches what I want, then I can use `grep -v` to invert it to find the complement. `^\d+$` will do the trick. Let's break down that regex:

```
^ \d + $
1 2  3 4
```

1. start of the line
2. a digit (0-9)
3. one or more
4. end of the line

This particular regex uses extensions introduced by the Perl programming language, so we need to use the `-P` flag. Add the `-v` to invert it:

```
$ grep -e '^>' proteins.fa | sed "s/^>//" | grep -v -P '^\d+$' | head -5
26788002|emb|CAD19173.1| putative RNA helicase, partial [Agaricus bisporus virus X]
26788000|emb|CAD19172.1| putative RNA helicase, partial [Agaricus bisporus virus X]
985757046|ref|YP_009222010.1| hypothetical protein [Alternaria brassicicola fusarivirus 1]
985757045|ref|YP_009222011.1| hypothetical protein [Alternaria brassicicola fusarivirus 1]
985757044|ref|YP_009222009.1| polyprotein [Alternaria brassicicola fusarivirus 1]
```

Looking at the above output, we can see that it would be pretty easy to get rid of everything starting with the vertical bar, and `sed` is perfect for this. Note that we can tell `sed` to do more than one action by separating them with semicolons. Lastly, we need to ensure the IDs are sorted for the next step:

```
$ grep -e '^>' proteins.fa | sed "s/^>//; s/|.*//" | sort > protein-ids.o
```

To find the lines in "protein-ids.o" that are not in "clustered-ids.o", I can use the `comm` (common) command:

```
$ comm -23 protein-ids.o clustered-ids.o > unclustered-ids.o
```

Did we get a reasonable answer?

```
$ wc -l clustered-ids.o unclustered-ids.o
  16257 clustered-ids.o
 204263 unclustered-ids.o
 220520 total
$ wc -l protein-ids.o
220520 protein-ids.o
```

Gapminder

For this exercise, look in the `biosys-analytics/data/gapminder` directory.

How many "txt" files are in the directory?

```
$ ls *.txt | wc -l
```

How many lines are in each/all of the files?

```
$ wc -l *.txt
```

You can use `cat` to spew at the entire contents of a file into your shell, but if you'd just like to see the top of a file, you can use:

```
$ head Trinidad_and_Tobago.cc.txt
```

If you only want to see 5 lines, use `-n 5` or `-5` .

For our exercise, we'd like to combine all the files into one file we can analyze. That's easy enough with:

```
$ cat *.cc.txt > all.txt
```

Let's use `head` to look at the top of file:

```
$ head -5 all.txt
Afghanistan    1997    22227415    Asia    41.763    635.341351
Afghanistan    2002    25268405    Asia    42.129    726.7340548
Afghanistan    2007    31889923    Asia    43.828    974.5803384
Afghanistan    1952    8425333    Asia    28.801    779.4453145
Afghanistan    1957    9240934    Asia    30.332    820.8530296
```

Hmm, there are no column headers. Let's fix that. There's one file that's pretty different in content (it has only one line) and name ("country.cc.txt"):

```
$ cat country.cc.txt
country    year    pop    continent    lifeExp    gdpPercap
```

Those are the headers that you can combine to all the other files to get named columns, something very important if you want to look at the data in Excel and R/Python data frames.

```
$ rm all.txt
$ mv country.cc.txt headers
$ cat headers *.txt > all.txt
$ head -5 all.txt | column -t
country      year  pop       continent  lifeExp  gdpPercap
Afghanistan  1997  22227415  Asia       41.763   635.341351
Afghanistan  2002  25268405  Asia       42.129   726.7340548
Afghanistan  2007  31889923  Asia       43.828   974.5803384
Afghanistan  1952  8425333   Asia       28.801   779.4453145
```

Yes, that looks much better. Double-check that the number of lines in the `all.txt` match the number of lines of input:

```
$ wc -l *.cc.txt headers
$ wc -l all.txt
```

How many observations do we have for 1952? For this, we need to find all the rows where the second field is equal to "1952," and `awk` will let us do just that. Normally `awk` splits on whitespace, but we have tab-delimited so we need to use `-F"\t"`. Recipes in `awk` take the form of a CONDITIONAL and an ACTION. If the CONDITIONAL is missing, then the ACTION is applied to all lines. If the ACTION is missing, then the default is to print the entire line. Here we just provide the CONDITIONAL and then count the results:

```
$ awk -F"\t" '$2 == "1952"' all.txt | wc -l
```

How many observations for each year?

```
$ awk -F"\t" '{print $2}' all.txt | sort | uniq -c
```

How many observations are present for Africa (the fourth field is continent)?

```
$ awk -F"\t" '$4 == "Africa"' all.txt | wc -l
```

And what are the countries in Africa?

```
$ awk -F"\t" '$4 == "Africa" {print $1}' all.txt | sort | uniq
```

How many observations for for each continent?

```
$ awk -F"\t" 'NR>1 {print $4}' all.txt | sort | uniq -c
```

What was the world population in 1952? To answer this, we need to get the third column when the second column is "1952":

```
$ awk -F"\t" '$2 == "1952" {print $3}' all.txt
```

There's a problem because one of the numbers is in scientific notation:

```
$ awk -F"\t" '$2 == "1952" {print $3}' all.txt | grep [a-z]
3.72e+08
```

Let's just remove that using `grep -v` (the `-v` reverses the match), then use the `paste` command to put a "+" in between all the numbers:

```
$ awk -F"\t" '$2 == "1952" {print $3}' *.cc.txt | grep -v [a-z] | paste -sd+ -
```

and then we pipe that to the `bc` calculator:

```
$ awk -F"\t" '$2 == "1952" {print $3}' *.cc.txt | grep -v [a-z] | paste -sd+ - | bc
2034957150.999989
```

It bothers me that it's not an integer, so I'm going to use `printf` in the `awk` command to trim that:

```
$ awk -F"\t" '$2 == "1952" {printf "%d\n", $3}' *.cc.txt | grep -v [a-z] | paste -sd+ - | bc
2406957150
```

I know that's all a bit crude and absurd, but I thought you might be curious just how far you can take this.

How many observations where the life expectancy ("lifeExp," field #5) is greater than 40?

```
$ awk -F"\t" '$5 > 40' all.txt | wc -l
```

How many of those are from Africa?

```
$ awk -F"\t" '$5 > 40 && $4 == "Africa"' all.txt
```

How many countries had a life expectancy greater than 70, grouped by year?

```
$ awk -F"\t" '$5 > 70 { print $2 }' all.txt | sort | uniq -c
   5 1952
   9 1957
  16 1962
  25 1967
  30 1972
  38 1977
  44 1982
  49 1987
  54 1992
  65 1997
  75 2002
  83 2007
```

How could we add continent to this?

```
$ awk -F"\t" '$5 > 70 { print $2 ":" $4 }' all.txt | sort | uniq -c
```

As you look at the data and want to ask more complicated questions like how does `gdpPercap` affect `lifeExp`, you'll find you need more advanced tools like

Python or R. Now that the data has been collated and the columns named, that will be much easier.

What if we want to add headers to each of the files?

```
$ mkdir wheaders
$ for file in *.txt; do cat headers $file > wheaders/$file; done
$ wc -l wheaders/* | head -5
      13 wheaders/Afghanistan.cc.txt
      13 wheaders/Albania.cc.txt
      13 wheaders/Algeria.cc.txt
      13 wheaders/Angola.cc.txt
      13 wheaders/Argentina.cc.txt
$ head wheaders/Vietnam.cc.txt
country    year    pop    continent    lifeExp    gdpPercap
Vietnam    1952    26246839    Asia    40.412    605.0664917
Vietnam    1957    28998543    Asia    42.887    676.2854478
Vietnam    1962    33796140    Asia    45.363    772.0491602
Vietnam    1967    39463910    Asia    47.838    637.1232887
Vietnam    1972    44655014    Asia    50.254    699.5016441
Vietnam    1977    50533506    Asia    55.764    713.5371196
Vietnam    1982    56142181    Asia    58.816    707.2357863
Vietnam    1987    62826491    Asia    62.82    820.7994449
Vietnam    1992    69940728    Asia    67.662    989.0231487
```

Exercises

Exercise: head.sh

Write a bash script called `head.sh` that mimics the `head` utility where it will print the first few lines of a file. The script should expect one required argument (the file) and a second optional argument of the number of lines, defaulting to 3. If are no arguments, it should print a "Usage" and exit *with an error code* Your program will expect to receive an argument in `$1` and maybe a second in `$2`. If the first argument is not a file, it should notify the user and exit *with an error code*. If the second argument is missing, use the value "3". Print the number of lines requested by the user by iterating over the lines in the file and exiting the loop appropriately. Do not use the actual `head` command!

```
$ ./head.sh
Usage: head.sh FILE NUM
$ ./head.sh files/issa.txt
Selected Haiku by Issa

Don't worry, spiders,
```

```
$ ./head.sh files/issa.txt 5
Selected Haiku by Issa

Don't worry, spiders,
I keep house
casually.
```

Exercise: cat_n.sh

Write a bash program called `cat_n.sh` that mimics the behavior of `cat -n` where it will print the line number and line of an input file. If there are no arguments, it should print a "Usage" and exit *with an error code*. Your program will expect to receive an argument in `$1`. If the argument is not a file, it should notify the user and exit *with an error code*. It will iterate over the lines in the file and print the line number, a space, and the line of the file. Your output will differ from regular `cat -n` as I won't expect you to right-align the numbers.

```
$ ./cat_n.sh
Usage: cat-n.sh FILE
$ ./cat_n.sh foo
foo is not a file
$ ./cat_n.sh files/sonnet-29.txt
1 Sonnet 29
2 William Shakespeare
3
4 When, in disgrace with fortune and men's eyes,
5 I all alone beweep my outcast state,
6 And trouble deaf heaven with my bootless cries,
7 And look upon myself and curse my fate,
8 Wishing me like to one more rich in hope,
9 Featured like him, like him with friends possessed,
10 Desiring this man's art and that man's scope,
11 With what I most enjoy contented least;
12 Yet in these thoughts myself almost despising,
13 Haply I think on thee, and then my state,
14 (Like to the lark at break of day arising
15 From sullen earth) sings hymns at heaven's gate;
16 For thy sweet love remembered such wealth brings
17 That then I scorn to change my state with kings.
```

Exercise: hello.sh

Create a bash script called `hello.sh` that accepts one or two arguments. If there are no arguments, it should print a "Usage" and exit *with an error code*.

Your program will expect to receive a "greeting" in `$1` and possibly a name in `$2`; if there is no second argument, use "Human" as the default. If there are more than two arguments, print a "Usage" and exit *with an error code*. Print the greeting, a comma and space, the name, and an exclamation point.

```
$ ./hello.sh
Usage: hello.sh GREETING [NAME]
$ ./hello.sh That\'ll do pig
Usage: hello.sh GREETING [NAME]
$ ./hello.sh "That'll do" pig
That'll do, pig!
$ ./hello.sh "Top o' the morning"
Top o' the morning, Human!
$ ./hello.sh "Greetings" "Earthling"
Greetings, Earthling!
```

Exercise: gap.sh

Write a bash script called `gap.sh` that will print out the files in the `gapminder` directory. Note that to be portable for testing purposes, you will need to use a **relative** path from the directory where the script lives (hint: start with `$PWD`). If there are no arguments, print out all the *basenames* of the files in sorted order. If there is an argument, treat it like a regular expression and find files where the basename matches at the beginning of the string in a case-insensitive manner and print them in sorted order. If no files are found, print a message telling the user.

```
$ ./gap.sh | head -5
     1  Afghanistan
     2  Albania
     3  Algeria
     4  Angola
     5  Argentina
$ ./gap.sh l
     1  Lebanon
     2  Lesotho
     3  Liberia
     4  Libya
$ ./gap.sh [w-z]
     1  West_Bank_and_Gaza
     2  Yemen_Rep
     3  Zambia
     4  Zimbabwe
$ ./gap.sh x
There are no countries starting with "x"
```