Portfolio Report 4: Linux

Jake Spiteri

2020

The Unix Shell

Pipes and Filters

Pipes and filters allow us to combine multiple programs and functions. This allows us to perform a sequence of actions and then output the result rather than reading, editing, and writing to a file. This saves a lot of time and allows us to develop more powerful programs.

We will look at using pipes with data from the directory `data-shell/molecules` which contains files describing simple organic molecules. This dataset can be downloaded here.

We can quickly view all files in this rectory by using the `ls` command. We can use the flag `-F` to make the output a little more readable. Below you see that folders now have a trailing `/` indicating that this is a directory, and all files have their extensions displayed.

```
ls -F
```

```
## creatures/
## data/
## get_lines.sh
## middle.sh
## molecules/
## north-pacific-gyre/
## notes.txt
## pizza.cfg
## solar.pdf
## writing/
```

We can also use `ls` to print the contents of directories other than our current working directory. Below we print information about files in the sub-directory `molecules`.

```
ls molecules -F
```

```
## cat
## cubane.pdb
## ethane.pdb
## --help
## lengths.txt
## methane.pdb
## octane.pdb
## pentane.pdb
## propane.pdb
## sort
## sorted-lengths.txt
## testfile01.txt
```

We can also use `cd` to move the working directory into the specified directory. We will use `cat` to print out a file. The function `cat` gets its name from concatenate — when given multiple files it will combine them one after the other, but in this case we only supply one file. Let's look at `cubane.pdb`.

```
cd molecules
cat cubane.pdb
```

```
## COMPND      CUBANE
## AUTHOR      DAVE WOODCOCK  95 12 06
## ATOM      1  C           1       0.789  -0.852   0.504  1.00  0.00
## ATOM      2  C           1      -0.161  -1.104  -0.624  1.00  0.00
## ATOM      3  C           1      -1.262  -0.440   0.160  1.00  0.00
## ATOM      4  C           1      -0.289  -0.202   1.284  1.00  0.00
## ATOM      5  C           1       1.203   0.513  -0.094  1.00  0.00
## ATOM      6  C           1       0.099   1.184   0.694  1.00  0.00
## ATOM      7  C           1      -0.885   0.959  -0.460  1.00  0.00
## ATOM      8  C           1       0.236   0.283  -1.269  1.00  0.00
## ATOM      9  H           1       1.410  -1.631   0.942  1.00  0.00
## ATOM     10  H           1      -0.262  -2.112  -1.024  1.00  0.00
## ATOM     11  H           1      -2.224  -0.925   0.328  1.00  0.00
## ATOM     12  H           1      -0.468  -0.501   2.315  1.00  0.00
## ATOM     13  H           1       2.224   0.892  -0.134  1.00  0.00
## ATOM     14  H           1       0.240   2.112   1.251  1.00  0.00
## ATOM     15  H           1      -1.565   1.730  -0.831  1.00  0.00
## ATOM     16  H           1       0.472   0.494  -2.315  1.00  0.00
## TER      17              1
## END
```

In order to avoid printing an entire file which can be quite an expensive operation, we can use `head` or `tail` to print the beginning or end of the file respectively. These functions also allow us to specify how many lines we want to print via `-n`.

```
cd molecules
head cubane.pdb -n 2
tail cubane.pdb -n 4
```

```
## COMPND      CUBANE
## AUTHOR      DAVE WOODCOCK  95 12 06
## ATOM     15  H           1      -1.565   1.730  -0.831  1.00  0.00
## ATOM     16  H           1       0.472   0.494  -2.315  1.00  0.00
## TER      17              1
## END
```

Let's use the word count command `wc` to compute the number of words in a document. We could also use the `-l` attachment to *only* report the number of lines in the document. By default, the `wc` command outputs the newline, word, and byte counts for each file provided; this information can easily be found by typing `wc --help` into the terminal.

```
cd molecules
wc cubane.pdb
wc -l cubane.pdb
```

```
##   20  156 1158 cubane.pdb
## 20 cubane.pdb
```

We can use regular expressions to incorporate flexibility into our commands. Running the command ls `*.pdb` will return all file names which end in `.pdb`. In a similar fashion, we could print all files in all subdirectories by running `ls *`. This is quite a powerful technique which allows us to select all files within a directory of a

particular file type.

```
cd molecules
wc *.pdb
```

```
##   20  156 1158 cubane.pdb
##   12   84  622 ethane.pdb
##    9   57  422 methane.pdb
##   30  246 1828 octane.pdb
##   21  165 1226 pentane.pdb
##   15  111  825 propane.pdb
##  107  819 6081 total
```

```
ls *
```

```
## get_lines.sh
## middle.sh
## notes.txt
## pizza.cfg
## solar.pdf
##
## creatures:
## basilisk.dat
## minotaur.dat
## original-basilisk.dat
## original-minotaur.dat
## original-original-basilisk.dat
## original-original-minotaur.dat
## original-original-original-basilisk.dat
## original-original-original-minotaur.dat
## original-original-original-original-basilisk.dat
## original-original-original-original-minotaur.dat
## original-original-original-original-unicorn.dat
## original-original-original-unicorn.dat
## original-original-unicorn.dat
## original-unicorn.dat
## unicorn.dat
##
## data:
## amino-acids.txt
## animal-counts
## animals.txt
## elements
## morse.txt
## pdb
## planets.txt
## salmon.txt
## sunspot.txt
##
## molecules:
## cat
## cubane.pdb
## ethane.pdb
## --help
## lengths.txt
## methane.pdb
```

```
## octane.pdb
## pentane.pdb
## propane.pdb
## sort
## sorted-lengths.txt
## testfile01.txt
##
## north-pacific-gyre:
## 2012-07-03
##
## writing:
## data
## haiku.txt
## thesis
## tools
```

We can combine the commands we have seen thus far using *pipes*. For example, we may be interested in printing the number of lines in each document in a subdirectory and redirecting this information to a text file.

```
cd molecules
wc -l *.pdb > lengths.txt
```

We print the `lengths.txt` file.

```
cd molecules
cat lengths.txt
```

```
##   20 cubane.pdb
##   12 ethane.pdb
##    9 methane.pdb
##   30 octane.pdb
##   21 pentane.pdb
##   15 propane.pdb
##  107 total
```

We may be interested in sorting this document, which can again be done easily in the UNIX shell. By default the `sort` filter sorts data alphabetically, but we can implement a numeric sort by specifying `-n`. Once again we can use pipes to save the sorted data.

```
cd molecules
sort -n lengths.txt > sorted-lengths.txt
```

```
cd molecules
cat sorted-lengths.txt
```

```
##    9 methane.pdb
##   12 ethane.pdb
##   15 propane.pdb
##   20 cubane.pdb
##   21 pentane.pdb
##   30 octane.pdb
##  107 total
```

Interestingly, we could combine all of the above operations into a single line using *pipes*. The pipe `|` tells the shell that we want to use the output of the program on the left as an input to the program on the right.

```
cd molecules

# combine word count and sort
```

```
wc -l *.pdb | sort -n

# combine word count and sort, and save in a text file
wc -l *.pdb | sort -n > lengths.txt
```

```
##    9 methane.pdb
##   12 ethane.pdb
##   15 propane.pdb
##   20 cubane.pdb
##   21 pentane.pdb
##   30 octane.pdb
##  107 total
```

We could combine the above operations to print out the largest or smallest file (in terms of line count) with a single line of code.

```
cd molecules

# smallest
wc -l *.pdb | sort -n | head -n 1

# largest
wc -l *.pdb | sort -n | tail -n 2 | head -n 1
```

```
##    9 methane.pdb
##   30 octane.pdb
```

We now demonstrate the use of the `cut` filter which separates lines of a file using delimiters. For this example we will use the `animals.txt` file.

```
cd data
cat animals.txt
```

```
## 2012-11-05,deer
## 2012-11-05,rabbit
## 2012-11-05,raccoon
## 2012-11-06,rabbit
## 2012-11-06,deer
## 2012-11-06,fox
## 2012-11-07,rabbit
## 2012-11-07,bear
```

We want to separate the dates from the animals. They are separated by a comma delimiter which we must provide to `cut`. We must also specify one and only one of `-b`, `-c`, or `-f`, to indicate which information to retain. In this case we will keep the second field i.e. the data after the delimiter.

```
cd data
cat animals.txt | cut -d , -f 2
```

```
## deer
## rabbit
## raccoon
## rabbit
## deer
## fox
## rabbit
## bear
```

We can combine these simple filters to provide advanced functionality. Let's print which days animals were recorded.

```
cd data
cat animals.txt | cut -d , -f 1 | uniq
```

```
## 2012-11-05
## 2012-11-06
## 2012-11-07
```

We could also print which animals were seen.

```
cd data
cat animals.txt | cut -d , -f 2 | sort -u
```

```
## bear
## deer
## fox
## rabbit
## raccoon
```

Loops

Loops allow us to repeat a command a given number of times, or for each item in a list. Using a loop can greatly reduce the amount of code needed for repetitive tasks.

In this example we will use the `creatures` directory. Within the directory are three files: `basilisk.dat`, `minotaur.dat`, and `unicorn.dat`. Let's print the head of each files.

```
cd creatures
head -n 5 basilisk.dat minotaur.dat unicorn.dat
```

```
## ==> basilisk.dat <==
## COMMON NAME: basilisk
## CLASSIFICATION: basiliscus vulgaris
## UPDATED: 1745-05-02
## CCCCAACGAG
## GAAACAGATC
##
## ==> minotaur.dat <==
## COMMON NAME: minotaur
## CLASSIFICATION: bos hominus
## UPDATED: 1765-02-17
## CCCGAAGGAC
## CGACATCTCT
##
## ==> unicorn.dat <==
## COMMON NAME: unicorn
## CLASSIFICATION: equus monoceros
## UPDATED: 1738-11-24
## AGCCGGGTCG
## CTTTACCTTA
```

Let's use the filters we discussed above and a loop to extract the classification for each file, which is contained in the second row. We see that this can be done with minimal effort.

```
cd creatures
for filename in basilisk.dat minotaur.dat unicorn.dat
```

```
do head -n 2 $filename | tail -n 1
done
```

```
## CLASSIFICATION: basiliscus vulgaris
## CLASSIFICATION: bos hominus
## CLASSIFICATION: equus monoceros
```

The general syntax for a for loop in the UNIX shell is

```
for thing in list_of_things
do filter $thing
done
```

To make life easier we can combine for loops with regular expressions. For example instead of manually writing out the three `.dat` files, we can simply list them using a regular expression `*.dat`.

```
cd creatures
for file in *.dat
do
    echo $file
    head -n 2 $file | tail -n 1
done
```

```
## basilisk.dat
## CLASSIFICATION: basiliscus vulgaris
## minotaur.dat
## CLASSIFICATION: bos hominus
## original-basilisk.dat
## CLASSIFICATION: basiliscus vulgaris
## original-minotaur.dat
## CLASSIFICATION: bos hominus
## original-original-basilisk.dat
## CLASSIFICATION: basiliscus vulgaris
## original-original-minotaur.dat
## CLASSIFICATION: bos hominus
## original-original-original-basilisk.dat
## CLASSIFICATION: basiliscus vulgaris
## original-original-original-minotaur.dat
## CLASSIFICATION: bos hominus
## original-original-original-original-basilisk.dat
## CLASSIFICATION: basiliscus vulgaris
## original-original-original-original-minotaur.dat
## CLASSIFICATION: bos hominus
## original-original-original-original-unicorn.dat
## CLASSIFICATION: equus monoceros
## original-original-original-unicorn.dat
## CLASSIFICATION: equus monoceros
## original-original-unicorn.dat
## CLASSIFICATION: equus monoceros
## original-unicorn.dat
## CLASSIFICATION: equus monoceros
## unicorn.dat
## CLASSIFICATION: equus monoceros
```

Let's use a for loop to backup our files so we can safely modify them.

```
cd creatures
for file in *.dat
do
    cp $file original-$file
done
```

Shell Scripts

Now we can explore the real power of the UNIX shell, which come in the form of scripts. You may often find yourself reusing the same set of commands frequently in order to update a set of files or push them to a server. In this scenario it may be work writing a script which will repeat these commands for you.

We will write a very basic script to extract the second to fifth line of any document supplied to it. The script is below and saved as `middle.sh`.

```
# make script
echo "
head -n 5 \$1 | tail -n 4
" > middle.sh

# test script
cd molecules
bash ../middle.sh octane.pdb
```

```
## AUTHOR      DAVE WOODCOCK  96 01 05
## ATOM      1  C           1      -4.397   0.370  -0.255  1.00  0.00
## ATOM      2  C           1      -3.113  -0.447  -0.421  1.00  0.00
## ATOM      3  C           1      -1.896   0.386  -0.007  1.00  0.00
```

We can increase the flexibility of our scripts by requiring arguments. Below we will write a script to extract lines from a file. We should be able to call the script in the following way `bash get_lines.sh input from_line to_line`. Below we extract lines 1 to 10.

```
# make script
echo "
diff=\$(expr \$3 - \$2)
head -n \"\$3\" \"\$1\" | tail -n \$diff
" > get_lines.sh

# test script
cd molecules
bash ../get_lines.sh octane.pdb 1 10
```

```
## AUTHOR      DAVE WOODCOCK  96 01 05
## ATOM      1  C           1      -4.397   0.370  -0.255  1.00  0.00
## ATOM      2  C           1      -3.113  -0.447  -0.421  1.00  0.00
## ATOM      3  C           1      -1.896   0.386  -0.007  1.00  0.00
## ATOM      4  C           1      -0.611  -0.426  -0.198  1.00  0.00
## ATOM      5  C           1       0.608   0.405   0.216  1.00  0.00
## ATOM      6  C           1       1.892  -0.400   0.001  1.00  0.00
## ATOM      7  C           1       3.113   0.429   0.414  1.00  0.00
## ATOM      8  C           1       4.397  -0.374   0.199  1.00  0.00
```

We can obtain more functionality if we replace `$1` in the script with `$@` which allows us to pass multiple files into the function.

It may be useful to know that you can debug bash scripts by executing them with the option `-x`.

Finding Things

In this section we will show how you might search for files using the UNIX shell, and search for information within files. The main function we will be using is `grep` which stands for global/regular expression/print. Let's look at a simple example.

```
cd writing
cat haiku.txt
```

```
## The Tao that is seen
## Is not the true Tao, until
## You bring fresh toner.
##
## With searching comes loss
## and the presence of absence:
## "My Thesis" not found.
##
## Yesterday it worked
## Today it is not working
## Software is like that.
```

`grep` allows us to search for and print lines in files which match a given pattern. Let's extract a couple of lines from the file `haiku.txt` using `grep`.

```
cd writing

# search for any sentences containing searching
grep searching haiku.txt
```

```
## With searching comes loss
```

```
cd writing

# search for any sentences containing it
grep it haiku.txt
```

```
## With searching comes loss
## Yesterday it worked
## Today it is not working
```

When searching for "it", `grep` also extracted lines which included words with "it" in them such as "with". If we only want to search for sentences with the *word* "it", we can use the option `-w`.

```
cd writing

# search for any sentences containing it
grep -w it haiku.txt
```

```
## Yesterday it worked
## Today it is not working
```

We can expand this further and search for specific phrases using quotation marks.

```
cd writing
```

```
# search for any sentences containing it
grep -w "it worked" haiku.txt
```

```
## Yesterday it worked
```

`grep` offers many other options which can be seen using `grep --help`. These include `-n` which numbers the lines which match, `-i` which makes our search case-insensitive, and `-v` which inverts the search (i.e. we find all lines which do *not* contain a phrase).

Let's look for files using `find`. We begin with `find .` which simply tells the shell to find everything in the current directory.

```
cd writing
find .
```

```
## .
## ./tools
## ./tools/old
## ./tools/old/oldtool
## ./tools/format
## ./tools/stats
## ./data
## ./data/one.txt
## ./data/two.txt
## ./data/LittleWomen.txt
## ./thesis
## ./thesis/empty-draft.md
## ./haiku.txt
```

We can specify the type of thing we are looking for using `-type`. For example, let's look for all subdirectories. Similarly, we could use `-type f` to find all files within the subdirectory.

```
cd writing
find . -type d
```

```
## .
## ./tools
## ./tools/old
## ./data
## ./thesis
```

You may wonder how to combine filters such as `wc` with files selected using `find` and `grep`. The most obvious approach of using filters does not work as you may expect, as demonstrated below.

```
cd molecules
find . -name "*.pdb" | wc -l
```

```
## 6
```

This is because `wc` is counting the number of files found. Instead we must provide the file names to `wc` as arguments. One way to do this is as follows

```
cd molecules
wc -l $(find . -name "*.pdb")
```

```
##   21 ./pentane.pdb
##   12 ./ethane.pdb
##   15 ./propane.pdb
##   20 ./cubane.pdb
##   30 ./octane.pdb
```

```
##    9 ./methane.pdb
##  107 total
```

An alternative approach might be to use a for loop.

```
cd molecules
for filename in $(find . -name "*.pdb")
do
    wc -l $filename
done
```

```
## 21 ./pentane.pdb
## 12 ./ethane.pdb
## 15 ./propane.pdb
## 20 ./cubane.pdb
## 30 ./octane.pdb
## 9 ./methane.pdb
```

In both cases above we have used `$()`. This ensures that the shell runs the code in the `$()` first.