12 Building Bioinformatics Pipelines I

12.1 Overview

Many types of analyses, especially those involving genomic data, require the investigator to carry out a large number of sequential steps. For example, given a set of uncharacterized genes in your organism of interest you might want to find out as much as you can about the structure and function of the proteins they encode, search for related proteins in other organisms, and try to identify pathways that they might be involved in. If you had only a single gene of interest you might apply each of the appropriate software tools by hand to carry out such an analysis. However, when the number of genes of interest grows beyond a small number (say 10-15) doing such an analysis by hand starts to become tedious and error prone. A bioinformatics pipeline can help to automate this process, will make the analysis easier to replicate or apply to new sets of genes, and can be modified to include additional tasks. Writing out a series of analysis steps as a pipeline also helps us to achieve the goal of 'reproducible research' in the same way that knitr helps you to do so in R.

During the next two class sessions we're going to build a series of analysis pipelines, starting with simple shell scripts and eventually building up to a complex series of analyses integrating Python code, several command line programs, and web queries to NCBI.

12.2 The Unix philosophy

Doug McIlroy who invented the concept of the Unix 'pipe' (discussed below) summarized the Unix philosphy as follows:

> "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

This is not some rigid set of specifications, but rather an approach to writing simple programs that can be tied together in useful ways to accomplish larger, more complex tasks. Most of the standard Unix commands are written with this philosophy in mind. The scripts you will develop over the next two class session will follow the same philosophy (and take advantage of other software tools that also use the Unix approach).

12.3 The Unix command line

A Unix/Linux command line environment is the de-facto standard for building bioinformatics pipelines. While the command line may not be particularly user friendly, various aspects of how Unix is designed make it very powerful for constructing analysis pipelines. We'll review some of those design aspects here.

Basic Unix commands

You should familiarize yourself with the basic unix commands covered in the UNIX Tutorial for Beginners (see link on class website). Here are some of the more common ones you'll need to navigate around your file system:

- `ls` – list the content of a directory e.g. `ls /home/`
- `cd` – change directory. e.g. `cd /home/pmagwene/tmp`
- `pwd` – display the name of the present working directory.
- `mv` – move a file. e.g. `mv myfile newfile`
- `rm` – remove (delete) a file. Be careful with this one! e.g. `rm tmpfile`
- `find` – find files that match a given patern. e.g. `find . -name "*.txt"` (matches all files in the current directory that end with '.txt').
- `man` – show the manual pages for a command. e.g. `man ls`
- `less` – show the contents of a file, displaying one page at a time. e.g. `less somefile.txt` (use the space bar to advance, `b` to go back, `q` to quit)

The bash shell

The bash shell is the default shell on most Linux systems, Cygwin, and recent versions of OS X. The shell itself provides a useful framework for interacting with the operating system. Shell scripts can be written to make the shell environment even more powerful. We'll explore how to do this in today's exercises.

Here's a few efficiency tips to keep in mind when working in the bash shell:

- Scroll backwards and forwards through your command history using the up and down arrow keys
- Use the tab key to invoke command and file-name completion to keep your typing to a minimum
- Use `<ctrl-r>` and then start typing a word or phrase to search on; this invokes the history search mode to do a reverse incremental search of previous commands

- Once you've found the command you were searching for hit `<Enter>` to execute it or `<ctrl-j>` to retrieve the command for further editing.
- Use `<ctrl-g>` or `<ctrl-c>` to cancel the history search mode

Everything is a file or a process

One of the aspects of Unix that makes it easy to tie programs together is that the operating systems treats pretty much everything as either a *file* or a *process*. There are three categories of files in Unix: plain files (e.g. text files, image files, word documents, video files, the code for a program, etc.), directories (e.g. your home directory, the root directory), and devices (e.g. the keyboard, a printer, a display screen, etc). The same basic set of commands can be applied to all three types of files.

A *process* is an instance of a running program. Everytime you start a program the operating system creates a process ID (PID) that is associated with that process. Processes typically operate on data in the form of files (of any of the three types) and return data that is sent to a file (again, any of the three file types). Any given process can start multiple subprocesses (also called child processes), however a process can only have one parent. For example, when you logon to a Unix system you are typically working in a shell process (common shells include bash, tcsh, csh, etc.). When you type a command like `ls` this creates a child process. The parent process is temporarily suspended until the child process returns its output. The `ls` process takes a file as input (the current directory by default), and returns it's output to the display associated with the shell (represented by a device file).

Redirection and Pipes

Because Unix treats everything as a file or process, it's easy to change the source of input and the destination for output. There are several special operators that allow one to change the source/destination of input and output from a process. These are:

- > (redirect output operator)
- >> (append output operator)
- < (redirect input operator)
- | (pipe)

We'll give a few example of redirection and pipes using the commands `ls` (list directory contents), and `grep` (find lines matching a pattern):

```
$ ls -l
total 4184
drwx------+   43 pmagwene  staff     1462 Nov 13 18:14 Desktop
drwx------+   20 pmagwene  staff      680 Sep 23 12:32 Documents
... output truncated ...
$ ls -l > ex1.out # redirect output to file
```

[bash]

In the example above we redirected the output of the `ls -l` command to a file named `ex1.out`. Open the file to confirm this. Now we'll 'pipe' the output of `ls` to the `grep` command.

```
$ ls -l | grep 'Nov'
drwx------+   43 pmagwene  staff     1462 Nov 13 18:14 Desktop
drwx------+ 1285 pmagwene  staff    43690 Nov 13 18:48 Downloads
drwx------+   14 pmagwene  staff      476 Nov 10 11:51 Pictures
-rw-r--r--     1 pmagwene  staff     1427 Nov 14 14:13 ex1.out
-rw-r--r--     1 pmagwene  staff   604976 Nov  1 12:21 rolland-etal-2000-cAMP.pdf
```

In this example we used grep to show all the lines of the `ls` output that have the string 'Nov' in them.

Now we'll combine those two commands and redirect the output to another file.

```
$ ls -l | grep 'Nov' > ex2.out
```

Finally, let's use the append output operator to append to our file lines with 'Oct' as well.

```
$ ls -l | grep 'Oct' >> ex2.out
```

If we had used the redirection operator rather than append than it would have overwritten the previous contents of the file rather than adding the output to what was already there. Open `ex2.out` to confirm that your commands worked as expected.

Review chapter 3 of the UNIX Tutorial for Beginners (see link on class website) for more examples illustrating the use of redirection and pipes. We'll be using pipes and redirection throughout these hands-on exercises.

12.3.1 Using curl to retrieve files from the net

`curl` is a command line tool for transfering data to or from a server using a variety of different protocols including FTP, HTTP, SCP, etc. `curl` is available by default in recent versions of OS X. If you're running Cygwin on Windows you may have to install it from the "Net" subdirectory in the Cygwin installer.

Using `curl` is relatively straightforward. Here, we'll use it to download a file that we're going to use for today's exercises:

```
$ curl -O http://downloads.yeastgenome.org/curation/chromosomal_feature/
    saccharomyces_cerevisiae.gff
```

Type `man curl` or check out an online version of the manual for more information on using curl: http://curl.haxx.se/docs/manual.html.

12.3.2 The GFF3 File Format

GFF3 (GENERIC FEATURE FORMAT VERSION 3) is a text-based format for representing genomic features. It is widely used by the genomics research community for representing sequence features associated with genome projects. All of the major genome

databases provide data in GFF3 format and most of the software tools used by the reserach community can parse GFF3 formatted files.

You can read the details of the GFF3 format here: http://www.sequenceontology.org/gff3.shtml. Notice that a GFF3 file consists 9 columns, separated by tabs. Read the above web page to understand what each of these 9 columns represents.

12.4 Tools for manipulating text

Many types of data, including GFF3 files, are structured text files. Because of this it's useful to have a handle on some of the major tools that Unix provides for manipulating such files.

12.4.1 head and tail

`head` and `tail` respectively show the first n and last n lines of a file (default $n = 10$). These can be useful for quickly checking out what's in a file. `tail` is especially useful for looking at log files to see the last few entries entered in a log.

```
$ head saccharomyces_cerevisiae.gff
... < output truncated > ...
$ tail saccharomyces_cerevisiae.gff
... < output truncated > ...
```

Use the `-n` argument to specify the number of lines you'd like to see:

```
$ head -n 3 saccharomyces_cerevisiae.gff
##gff-version 3
#date Mon Nov 15 19:50:13 2010
#
```

12.4.2 less

`less` is 'pager' program that allows you to scroll through a file (or standard input) page by page.

```
$ less saccharomyces_cerevisiae.gff
```

From within `less` you can scroll forward by hitting the space bar, or the 'f' key, backward by typing 'b'. To search for a particular word or pattern in the file type '/' followed by the word of interest and then hit return. All the instances of that word / pattern will be highlighted. For example, from within `less` type `/gene<RET>`, where `<RET>` means hit the enter or return key, to find all instances of the word 'gene' in the file. Type `q` to quit `less`.

12.4.3 echo

`echo` simply writes it's string argument to standard output (i.e. it echos what you type).

```
$ echo "hello, world"
hello, world
$ echo "These are the times that try men's souls"
These are the times that try men's souls
```

12.4.4 cat

`cat`, short for 'concatenate', is a utility for concatenating and printing text. Here are some examples of it's use:

```
$ echo "some text here" > file1.txt
$ echo "some more text" > file2.txt
$ cat file1.txt file2.txt > file1plus2.txt
$ cat file1plus2.txt
some text here
some more text
```

12.4.5 wc

`wc` is a program that counts the number of words, lines, and characters in a file. You can also specify you only want one of those counts using options like `-l` (count only lines).

```
$ wc saccharomyces_cerevisiae.gff
  168490  299825 18871650 saccharomyces_cerevisiae.gff
$ wc -l saccharomyces_cerevisiae.gff
  168490 saccharomyces_cerevisiae.gff
```

12.4.6 cut

`cut` is a utility for subsetting words, bytes or columns of a text file. For example:

```
$ cut -f1-3 saccharomyces_cerevisiae.gff | less
```

In the above we use cut to show the first three fields of the file, and then we pipe it to `less` to examine one page of text at a time . The default field delimiter in `cut` is a tab (`\t`), but you can specify other delimiters with the `-d` option. You don't have to use adjacent columns with `cut`. For example,

```
$ cut -f1,3-5,7 saccharomyces_cerevisiae.gff | less
```

This allows us to look at the first column, and columns 3-5 and 7, corresponding to the seqid (=chromosome), the feature type, the feature start and stop coordinates (1-based), and the strand on which the feature is defined.

Notice how in addition to the fields, `cut` also gave use the header information at the beginning of the file. We can use the `-s` option to suppress lines that don't have the field delimiter character:

```
$ cut -s -f1,3-5,7 saccharomyces_cerevisiae.gff > out.txt
```

Notice this time we redirected the output of the command to a file, `out.txt`.

12.4.7 sort

The `sort` utility sorts lines of text. By default `sort` interprets an entire line of text as the key for sorting and sorts in dictionary order. For example, to see the default sorting:

```
$ sort out.txt | less
```

We can use the `-k` option to specify the field to sort on. For example, this is how we can sort on the second column of `out.txt`:

```
$ sort -k2 out.txt | less
```

Another useful option to sort is `-u` which tells sort to output only the first instance of a set of identical keys. Try and figure out what the following command does before running it:

```
$ cut -s -f3 saccharomyces_cerevisiae.gff | sort -u
```

12.4.8 grep

`grep` is a tool for doing regular expression matching on lines of a file. Regular expressions are a way to specificy search patterns in strings. The simplest type of regular expression is to just search for a specific word, as illustrated here:

```
$ grep "gene" out.txt | less
```

The above command simply returns all the lines in `out.txt` that have the word "gene" in them. Let's use this in a slightly different way to count instances of different features in the file:

```
$ grep "gene" out.txt | wc -l
    6720
$ grep "pseudogene" out.txt | wc -l
      21
$ grep "telomere" out.txt | wc -l
      32
```

NOTE: the numbers of matches may change somewhat between releases of the curated yeast genome. If the numbers you get above or below are *slightly* different from what is shown here don't worry.

We can get a little fancier if we use the "extended" `grep` syntax (specified using the `-E` option). Here's how we can search for lines that match on any of a set of terms (the vertical bar `|` indicates an "OR" operator):

```
$ grep -E "tRNA|rRNA|snRNA|snoRNA" out.txt | wc -l
     409
```

Note that we have to be careful about what `grep` matches, for example:

```
$ grep "chr01" out.txt | grep "gene" | wc -l
    121
```

Note how we piped two `grep` commands together to get the equivalent of AND ("chr01" AND "gene"). However, there's a very subtle problem with this command as constructed. We search on the word "gene" but "gene" is also a substring of "psuedogene" and hence "pseudogene" features also generate matches. What we really want is whole word matches. We can do that as follows:

```
$ grep "\<chr01\>" out.txt | grep "\<gene\>" | wc -l
     117
```

This uses what are called "POSIX character classes" to match possible sets of characters. A list of the POSIX character classes is linked to on the course wiki. Here's the equivalent call for counting genes on chromosome IV:

```
$ grep "\<chr04\>" out.txt | grep "\<gene\>" | wc -l
     836
```

We've only just scratched the surface of regular expressions. Regular expressions are a very powerful tool and there are whole books on the topic. I'll post a number of links on the course wiki to online tutorials on `grep` and regular expressions.

12.4.9 tr

`tr` is a utility for translating characters within a text stream. `tr` can be useful for converting delimiters from one file type to another. For example, let's say we wanted to analyze the file `out.txt` in a program that expected comma separated values (csv) instead of tab-delimited fields. `tr` makes that conversion easy:

```
$ cat out.txt | tr "\t" "," > out.csv
```

Note that `tr` only reads from standard input so we used the `cat` program to feed the lines of text to `tr`.

12.5 Awk

`awk` is a programming language designed for processing structured text files. You can use it to write short one liners or to write full blown programs. It turns out that some form of text file manipulation is often a necessary first step in most bioinformatics analyses, so `awk` often comes in very handy. We'll use `awk` to illustrate how you might transition from simple command line usage into slightly more complicated scripts.

One simple thing we can do with awk is to use it to re-order fields in a structured data file:

```
awk '{print $2, $1, $4, $5}' out.txt | less
```

In the command above the the dollar signs followed by numbers refer to the fields of the file. With it's default setting `awk` operates line by line, so you can interpret the above statement as saying: "for each line, print the fields 2, 1, 4 and 5".

12.5.1 The pattern {action} syntax of Awk

The basic syntax of awk is often depicted in the form pattern {action}. In the example below the pattern can be read as – "if the 3rd field is 'chromosome'". For all lines that match that pattern the corresponding action is applied; in this case "print fields 1 and 5" (the chromosome name and its length):

```
awk '$3=="chromosome" {print $1, $5}' saccharomyces_cerevisiae.gff
```

Here's another pattern {action} pair that shows how we could find all gene features with length less than 300:

```
$ awk '$3 == "gene" && ($5 - $4) < 300 {print $0 }'
    saccharomyces_cerevisiae.gff | wc -l
```

&& is the AND operator. The pattern for this example translates as "if the 3rd fields is 'gene' AND the the 5th field minus the 4th field is less than 300"; the corresponding action is "print the whole line." Notice how we piped the output of awk to the utility wc to count the number of lines returned.

Let's add one more condition to the previous example – we'll look for the word 'Dubious' in the 9th field.

```
$ awk '$3 == "gene" && ($5 - $4) < 300 && match($9, "Dubious") {print $0 }'
     saccharomyces_cerevisiae.gff | wc -l
```

Comparing the output of the previous example to this one you'll see that a significant proportion of small genes are classified as 'Dubious'.

12.5.2 Writing an Awk script

There's lots of powerful things you can do with awk one-liners, but writing short scripts often makes things easier to understand. You can think of an awk scripts as a series of pattern {action} statements. Our script will create a table giving both the length of each chromosome and the number of genes on that chromosome. Save the following script in a file called gcount.awk.

```
# gcount.awk
# length of each chromosome
$3 == "chromosome" {
    clen[$1] = $5
}

# increment the gene count for the given chromosome
$3 == "gene" {
    ngenes[$1] += 1
}

# END only gets carried out once
# we've processed all the records
END {
print "Chrom\tLength\t# Genes"
```

```
for (chr in clen) {
    print chr "\t" clen[chr] "\t" ngenes[chr]
    }
}
```

In this example we create two arrays – `clen` and `ngenes` – to keep track of the chromosome lengths and number of genes on each chromosome. Arrays can be indexed by either integers or strings; when they are indexed by strings we can think of them like Python dictionaries. We have two patterns – whether the 3rd field equals 1) "chromosome" or 2) "gene". The final pattern, labeled `END`, says what to do once we've processed all the lines in the file. Run this script as follows:

```
$ awk -f gcount.awk saccharomyces_cerevisiae.gff
```

The `-f` option says to use the pattern/action pairs contained in the specified file. One possible shortcoming (at least on my system) is that the output wasn't sorted. That's easy to solve by piping the results to the sort command:

```
$ awk -f gcount.awk saccharomyces_cerevisiae.gff | sort
```

12.5.3 A more flexible Awk script

Our `gcount.awk` script works pretty well, but what if we wanted to count pseudogenes rather than genes, or tRNA features? In its current form the feature type is hardcoded into the script. Let's see how we can get rid of that constraint. Save the following awk script as `fcount.awk`.

```
# fcount.awk
BEGIN {
# if var ftype has NOT been defined, assign it a default value
if (!ftype)
  ftype = "gene"
}

# length of each chromosome
$3 == "chromosome" {
    clen[$1] = $5
}

# increment the feature count for the given chromosome
$3 == ftype {
    ngenes[$1] += 1
}

END {
for (chr in clen) {
    print chr "\t" clen[chr] "\t" ngenes[chr]
    }
}
```

Here we introduced the `BEGIN` pattern. This pattern is carried out before any lines of the file are processed. By default, this new script will count genes like our previous script did, but if you specify the variable `ftype` using the `-v` option on the command line it will count the specified feature type:

```
# count pseudogenes
$ awk -f fcount.awk  -v ftype="pseudogene" saccharomyces_cerevisiae.gff

# counts ARS sequences (origins of replication)
$ awk -f fcount.awk  -v ftype="ARS" saccharomyces_cerevisiae.gff

# count tRNA genes
$ awk -f fcount.awk  -v ftype="tRNA" saccharomyces_cerevisiae.gff
```

12.5.4 String substitution in Awk

As a final example of using `awk`, let's see how we can use string substitution to create more human friendly output when we output our GFF file in awk.

Here's a now familiar example of using the `pattern {action}` syntax to find all the pseudogenes in a GFF file:

```
$ awk '$3 == "pseudogene" { print $0 }' saccharomyces_cerevisiae.gff  |
    less
```

This works, but the output is a bit ugly because of how the attribute field is specified in GFF format. Let's write a simple awk `function` that nicely formats the output. Save the following script as `attribs.awk`.

```
# attribs.awk
# parse the attributes field of a GFF file

NF >= 9 {
    # print some useful fields
    print "Chromosome =", $1
    print "Type = ", $3
    print "Start =", $4
    print "End =", $5
    print "Strand =", $7
    #print $1, $2, $3, $4, $5, $6, $7, $8

    # break the attributes field up into individual attributes
    n = split($9, attributes, ";")
    for (i = 1; i <= n; i++){
        tstr = attributes[i]
        gsub(/%20/," ", tstr ) # spaces
        gsub(/%2C/, ",", tstr) # commas
        gsub(/%3B/, ";", tstr) # semi-colons
        gsub(/%2F/, "/", tstr) # forward slash
        gsub("=","  = ", tstr) # add spaces around equal signs
```

```
        print tstr
        }
    print "\n"
}
```

The awk function `gsub()` globally substitutes one string for another. In this case it's replacing HTML type encoding of spaces, commas, semi-colons, etc. with more human friendly versions of the same. We can use our `attribs.awk` script as follows:

```
$ awk '$3 == "pseudogene" { print $0 }' saccharomyces_cerevisiae.gff  | awk
    -f attribs.awk | less
```

This produces output that is much nicer for a human reader to interpret, though perhaps less easy to parse computationally.

The GFF3 format is used by many organism specific genome projects besides yeast. If we take care to write our scripts to operate on GFF3 files generically then we can apply scripts we write for one organism easily to another organism. Let's test this out by downloading the X-chromosome GFF3 file for *Drosophila melanogaster* from FlyBase:

```
$ curl -O ftp://ftp.flybase.net/genomes/dmel/current/gff/dmel-X-r5.54.gff.
    gz
$ gunzip dmel-X-r5.54.gff.gz  # unzip the compressed file
```

Now let's test our `attribs.awk` script with this new GFF file by generating a report on pseudogenes on the *Drosophila* X-chromosome:

```
$ awk '$3 == "pseudogene"  {print $0}' dmel-X-r5.54.gff | awk -f attribs.
    awk > fly-X-pseudogenes.txt
```

Use `less` or a text editor to view your report.

12.6 Shell Scripting

To this point all of our examples have involved single command lines or scripts, occasionally tied together with pipes. This works well for quick analyses, but what if you wanted to run an analysis over and over again, say on a monthly basis as a genome project was updated, or as you generated new data as part of your research? In that context a shell script might be useful. A shell script is a small program written for the command line interpreter of an operating system. A shell script is convenient for tying together a series of commands that you might otherwise type by hand into a repeatable and documented set of operations. For these exercises we will assume that you use the "bash shell", which is the default command line interface on OSX, Cygwin, and most Linux based systems. You can confirm that your default shell is bash by doing something like:

```
$ sh --version
GNU bash, version 3.2.48(1)-release (x86_64-apple-darwin10.0)
Copyright (C) 2007 Free Software Foundation, Inc.
```

Assuming, that you've got bash working on your system, enter the following code into your text-editor and save it with the filename `genome_reporter.sh` in the same directory where you've saved `fcount.awk` that we created earlier. Be careful that you enter the text as shown as `bash` is particularly picky about extra spaces around the equal sign (=) in variable assignment so if you get error messages when you try and run this script (see below), that's the first thing to check.

```
#!/bin/bash

URL='http://downloads.yeastgenome.org/curation/chromosomal_feature/
    saccharomyces_cerevisiae.gff'
BASEFILE='saccharomyces_cerevisiae.gff'

# get today's date
TODAY=$(date -u +%Y-%m-%d)

# create filename, prepended w/today's date
FILENAME="$TODAY-$BASEFILE"
REPORT="report-$FILENAME"

# if the GFF file does not already exist then
# use curl to download the file and save if with the name above
if [ ! -e $FILENAME ]
then
    curl -o $FILENAME $URL
fi

# create report with a series of awk calls
echo -e "Genome Report\nPrepared: $TODAY\n" > $REPORT

echo "Total genes: " >> $REPORT
awk '$3 == "gene" {print $0}' $FILENAME | wc -l >> $REPORT

echo -e "\nDubious ORFs: " >> $REPORT
awk '$3 == "gene"  && match($9, "Dubious") {print $0 }' $FILENAME | wc -l
    >> $REPORT

echo -e "\nPseudogenes: " >> $REPORT
awk -f fcount.awk  -v ftype="pseudogene" $FILENAME | wc -l >> $REPORT

echo -e "\nChromosome, length, genes per chromosome: " >> $REPORT
awk -f fcount.awk $FILENAME | sort >> $REPORT

echo "Report written to: $REPORT"
```

Note that the line `#!/bin/bash` needs to be the first line in the file. This tells the operating system to run this script using the bash shell. This line is sometimes referred to as the 'she-bang' line by Unix programmers. We'll see next week how to set this for a Python program.

Having entered and saved that script, make the script executable by typing:

```
$ chmod +x genome_reporter.sh
```

from the command line. Once you've done that you can run the script, from the same directory, by typing:

```
$ ./genome_reporter.sh
```

Assuming you don't have any errors the script will download the GFF file from the Saccharomyces Genome Database, save it with the date prefixed to the file name, and then generate a short report listing some useful summaries generated from the file.

Most of the bottom half of the script should be easy to understand; it simply shows a bunch of `echo` and `awk` commands that you might have typed at the command line. In the top portion of the script we create a set of variables to hold the names of the files we'll be using. One new feature we haven't seen before is the use of the dollar sign ($) to dereference variable names. For example, the variable `FILENAME` is constructed by creating a string by joining together the strings held in the variables `TODAY` and `BASEFILE` (and separated by a dash -). Depending on the date on which the script is run it generates a different set of file names, as specified by the variables `TODAY`, `BASEFILE`, and `REPORT`. The bottom half of the script is setup to generate the appropriate output given those changing variables. One other feature to take note of is the `if-then-fi` conditional statement. The portion in the square brackets (`[ ! -e $FILENAME ]`) asks whether the file for that date already exists. If so it doesn't bother downloading the file again, for efficiency reasons.