Redirection

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- How can I search within files?
- How can I combine existing commands to do new things?

Objectives

- Employ the `grep` command to search for information within files.
- Print the results of a command to a file.
- Construct command pipelines with two or more stages.

Searching files

We discussed in a previous episode how to search within a file using `less`. We can also search within files without even opening them, using `grep`. `grep` is a command-line utility for searching plain-text files for lines matching a specific set of characters (sometimes called a string) or a particular pattern (which can be specified using something called regular expressions). We're not going to work with regular expressions in this lesson, and are instead going to specify the strings we are searching for. Let's give it a try!

Nucleotide abbreviations

The four nucleotides that appear in DNA are abbreviated `A`, `C`, `T` and `G`. Unknown nucleotides are represented with the letter `N`. An `N` appearing in a sequencing file represents a position where the sequencing machine was not able to confidently determine the nucleotide in that position. You can think of an `N` as a `NULL` value within a DNA sequence.

Suppose we want to see how many reads in our file have really bad segments containing 10 consecutive unknown nucleoties (Ns). Let's search for the string NNNNNNNNNN in the SRR098026 file.

Determining quality

In this lesson, we're going to be manually searching for strings of `N`s within our sequence results to illustrate some principles of file searching. It can be really useful to do this type of searching to get a feel for the quality of your sequencing results, however, in you research you will most likely use a bioinformatics tool that has a built-in program for filtering out low-quality reads. You'll learn how to use one such tool in a later lesson (http://www.datacarpentry.org/wrangling-genomics/00-readQC/).

```
$ grep NNNNNNNNNN SRR098026.fastq
```

This command returns a lot of output to the terminal. Each line in the SRR098026 file which contains at least 10 consecutive Ns is printed to the terminal. We may be interested not only in the actual sequence which contains this string, but in the name (or identifier) of that sequence. We discussed in a previous lesson that the identifier line immediately precedes the nucleotide sequence for each read in a FASTQ file. We may also want to inspect the quality scores associated with each of these reads. To get all of this information, we will return the line immediately before each match and the two lines immediately after each match.

We can use the `-B` argument for grep to return a specific number of lines before each match and the `-A` argument to return a specific number of lines after each matching line. Here we want the line before and the two lines after each matching line so we add `-B1 -A2` to our grep command.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq
```

One of the sets of lines returned by this command is:

```
@SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
+SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Exercise

1) Search for the sequence `GNATNACCACTTCC` in the `SRR098026.fastq` file. Have your search return all matching lines and the name (or identifier) for each sequence that contains a match.

2) Search for the sequence `AAGTT` in both FASTQ files. Have your search return all matching lines and the name (or identifier) for each sequence that contains a match.

Solution

1) `grep -B1 GNATNACCACTTCC SRR098026.fastq`
2) `grep -B1 AAGTT *.fastq`

Redirecting output

`grep` allowed us to identify sequences in our FASTQ files that match a particular pattern. But all of these sequences were printed to our terminal screen. In order to work with these sequences and perform other opperations on them, we will need to capture that output in some way.

We can do this with something called “redirection”. The idea is that we’re redirecting what was output to the terminal to another location. In our case, we want to print this information to a file, so that we can look at it later and do other analyses with this data.

The command for redirecting output to a file is `>`.

Let’s try out this command and copy all the records (including all four lines of each record) in our FASTQ files that contain ‘NNNNNNNNNN’ to another file called ‘bad_reads.txt’.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
```

File extensions

You might be confused about why we’re naming our output file with a `.txt` extension. After all, it will be holding FASTQ formatted data that we’re extracting from our FASTQ files. Won’t it also be a FASTQ file? The answer is, yes - it will be a FASTQ file and it would make sense to name it with a `.fastq` extension. However, using a `.fastq` extension will lead us to problems when we move to using wildcards later in this episode. We’ll point out where this becomes important. For now, it’s good that you’re thinking about file extensions!

The prompt should sit there a little bit, and then it should look like nothing happened. But type `ls`. You should see a new file called bad_reads.txt.

We can check the number of lines in our new file using a command called `wc`. `wc` stands for `word count`. This command counts the number of words, lines, and characters in a file.

```
$ wc bad_reads.txt
```

```
  537  1073 23217 bad_reads.txt
```

This will tell us the number of lines, words and characters in the file. If we want only the number of lines, we can use the `-l` flag for `lines`.

```
$ wc -l bad_reads.txt
```

```
537 bad_reads.txt
```

Because we asked `grep` for all four lines of each FASTQ record, we need to divide the output by four to get the number of sequences that match our search pattern.

Exercise

How many sequences in `SRR098026.fastq` contain at least 3 consecutive Ns?

Solution

```
$ grep NNN SRR098026.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
249
```

We might want to search multiple FASTQ files for sequences that match our search pattern. However, we need to be careful, because each time we use the `>` command to redirect output to a file, the new output will replace the output that was already present in the file. This is called "overwriting" and, just like you don't want to overwrite your video recording of your kid's first birthday party, you also want to avoid overwriting your data files.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
537 bad_reads.txt
```

```
$ grep -B1 -A2 NNNNNNNNNN SRR097977.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
0 bad_reads.txt
```

Here, the output of our second call to `wc` shows that we no longer have any lines in our bad_reads.txt file. This is because the second file we searched (`SRR097977.fastq`) does not contain any lines that match our search sequence. So our file was overwritten and is now empty.

We can avoid overwriting our files by using the command `>>`. `>>` is known as the "append redirect" and will append new output to the end of a file, rather than overwriting it.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
537 bad_reads.txt
```

```
$ grep -B1 -A2 NNNNNNNNNN SRR097977.fastq >> bad_reads.txt
$ wc -l bad_reads.txt
```

```
537 bad_reads.txt
```

The output of our second call to `wc` shows that we have not overwritten our original data.

We can also do this with a single line of code by using a wildcard.

```
$ grep -B1 -A2 NNNNNNNNNN *.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
537 bad_reads.txt
```

File extensions - part 2

This is where we would have trouble if we were naming our output file with a `.fastq` extension. If we already had a file called `bad_reads.fastq` (from our previous `grep` practice) and then ran the command above using a `.fastq` extension instead of a `.txt` extension, `grep` would give us a warning.

```
grep -B1 -A2 NNNNNNNNNN *.fastq > bad_reads.fastq
```

```
grep: input file 'bad_reads.fastq' is also the output
```

`grep` is letting you know that the output file `bad_reads.fastq` is also included in your `grep` call because it matches the `*.fastq` pattern. Be careful with this as it can lead to some surprising output.

So far we've searched for reads containing a long string of at least 10 unknown nucleotides. We might also be interested in finding any reads with at least two shorter strings of 5 unknown nucleotides, separated by any number of known nucleotides. Reads with more than one region of ambiguity like this might be poor enough to not pass our quality filter. We can search for these reads using a wildcard within our search string for `grep`.

Exercise

How many reads in the `SRR098026.fastq` file contain at least two regions of 5 unknown nucleotides in a row, separated by any number of known nucleotides?

Solution

```
$ grep "NNNNN*NNNNN" SRR098026.fastq > bad_reads_2.txt
$ wc -l bad_reads_2.txt
```

```
186 bad_reads_2.txt
```

We've now created two separate files to store the results of our search for reads matching particular criteria. Since we might have multiple different criteria we want to search for, creating a new output file each time has the potential to clutter up our workspace. We also so far haven't been interested in the actual contents of those files, only in the number of reads that we've found. We created the files to store the reads and then counted the lines in the file to see how many reads matched our criteria. There's a way to do this, however, that doesn't require us to create these intermediate files - the pipe command (|).

This is probably not a key on your keyboard you use very much, so let's all take a minute to find that key. What `|` does is take the output that is scrolling by on the terminal and uses that output as input to another command. When our output was scrolling by, we might have wished we could slow it down and look at it, like we can with `less`. Well it turns out that we can! We can redirect our output from our `grep` call through the `less` command.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq | less
```

We can now see the output from our `grep` call within the `less` interface. We can use the up and down arrows to scroll through the output and use `q` to exit `less`.

Redirecting output is often not intuitive, and can take some time to get used to. Once you're comfortable with redirection, however, you'll be able to combine any number of commands to do all sorts of exciting things with your data!

None of the command line programs we've been learning do anything all that impressive on their own, but when you start chaining them together, you can do some really powerful things very efficiently. Let's take a few minutes to practice.

Exercise

Now that we know about the pipe (|), write a single command to find the number of reads in the `SRR098026.fastq` file that contain at least two regions of 5 unknown nucleotides in a row, separated by any number of known nucleotides. Do this without creating a new file.

Solution

```
$ grep "NNNNN*NNNNN" SRR098026.fastq | wc -l
```

```
186
```

File manipulation and more practice with pipes

Let's use the tools we've added to our tool kit so far, along with a few new ones, to example our SRA metadata file. First, let's navigate to the correct directory.

```
$ cd /pool/genomics/username
$ cd dc_sample_data/sra_metadata
```

This file contains a lot of information about the samples that we submitted for sequencing. We took a look at this file in an earlier lesson. Here we're going to use the information in this file to answer some questions about our samples.

How many of the read libraries are paired end?

The samples that we submitted to the sequencing facility were a mix of single and paired end libraries. We know that we recorded information in our metadata table about which samples used which library preparation method, but we don't remember exactly where this data is recorded. Let's start by looking at our column headers to see which column might have this information. Our column headers are in the first row of our data table, so we can use `head` with a `-n` flag to look at just the first row of the file.

```
$ head -n 1 SraRunTable.txt
```

```
BioSample_s     InsertSize_l    LibraryLayout_s Library_Name_s  LoadDate_s      MBases_l        MBytes_l
        ReleaseDate_s Run_s SRA_Sample_s Sample_Name_s Assay_Type_s AssemblyName_s BioProject_s Center_Name
_s Consent_s Organism_Platform_s SRA_Study_s g1k_analysis_group_s g1k_pop_code_s source_s strain_s
```

That is only the first line of our file, but because there are a lot of columns, the output likely wraps around your terminal window and appears as multiple lines. Once we figure out which column our data is in, we can use a command called `cut` to extract the column of interest.

Because this is pretty hard to read, we can look at just a few column header names at a time by combining the `|` redirect and `cut`.

```
$ head -n 1 SraRunTable.txt | cut -f1-4
```

`cut` takes a `-f` flag, which stands for "field". This flag accepts a list of field numbers, in our case, column numbers. Here we are extracting the first four column names.

```
BioSample_s InsertSize_l      LibraryLayout_s   Library_Name_s
```

The LibraryLayout_s column looks like it should have the information we want. Let's look at some of the data from that column. We can use `cut` to extract only the 3rd column from the file and then use the `|` operator with `head` to look at just the first few lines of data in that column.

```
$ cut -f3 SraRunTable.txt | head -n 10
```

```
LibraryLayout_s
SINGLE
SINGLE
SINGLE
SINGLE
SINGLE
SINGLE
SINGLE
SINGLE
PAIRED
```

We can see that there are (at least) two categories, SINGLE and PAIRED. We want to search all entries in this column for just PAIRED and count the number of matches. For this, we will use the `|` operator twice to combine `cut` (to extract the column we want), `grep` (to find matches) and `wc` (to count matches).

```
$ cut -f3 SraRunTable.txt | grep PAIRED | wc -l
```

```
2
```

We can see from this that we have only two paired-end libraries in the samples we submitted for sequencing.

Exercise

How many single-end libraries are in our samples?

Solution

```
$ cut -f3 SraRunTable.txt | grep SINGLE | wc -l
```

```
35
```

How many of each class of library layout are there?

We can extract even more information from our metadata table if we add in some new tools: `sort` and `uniq`. The `sort` command will sort the lines of a text file and the `uniq` command will filter out repeated neighboring lines in a file. You might expect `uniq` to extract all of the unique lines in a file. This isn't what it does, however, for reasons involving computer memory and speed. If we want to extract all unique lines, we can do so by combining `uniq` with `sort`. We'll see how to do this soon.

For example, if we want to know how many samples of each library type are recorded in our table, we can extract the third column (with `cut`), and pipe that output into `sort`.

```
$ cut -f3 SraRunTable.txt | sort
```

If you look closely, you might see that we have one line that reads "LibraryLayout_s". This is the header of our column. We can discard this information using the `-v` flag in `grep`, which means return all the lines that **do not** match the search pattern.

```
$ cut -f3 SraRunTable.txt | grep -v LibraryLayout_s | sort
```

This command returns a sorted list (too long to show here) of PAIRED and SINGLE values. We can use the `uniq` command to see a list of all the different categories that are present. If we do this, we see that the only two types of libraries we have present are labelled PAIRED and SINGLE. There aren't any other types in our file.

```
$ cut -f3 SraRunTable.txt | grep -v LibraryLayout_s | sort | uniq
```

```
PAIRED
SINGLE
```

If we want to count how many of each we have, we can use the `-c` (count) flag for `uniq`.

```
$ cut -f3 SraRunTable.txt | grep -v LibraryLayout_s | sort | uniq -c
```

```
2 PAIRED
35 SINGLE
```

Exercise

1) How many different sample load dates are there?
2) How many samples were loaded on each date?

Solution

There are two different sample load dates.

```
cut -f5 SraRunTable.txt | grep -v LoadDate_s | sort | uniq
```

```
25-Jul-12
29-May-14
```

Six samples were loaded on one date and 31 were loaded on the other.

```
cut -f5 SraRunTable.txt | grep -v LoadDate_s | sort | uniq -c
```

```
 6 25-Jul-12
31 29-May-14
```

Can we sort the file by library layout and save that sorted information to a new file?

We might want to re-order our entire metadata table so that all of the paired-end samples appear together and all of the single-end samples appear together. We can use the `-k` (key) flag for `sort` to sort based on a particular column. This is similar to the `-f` flag for `cut`.

Let's sort based on the third column (`-k3`) and redirect our output to a new file.

```
$ sort -k3 SraRunTable.txt > SraRunTable_sorted_by_layout.txt
```

Can we extract only paired-end records into a new file?

We also might want to extract the information for all samples that meet a specific criterion (for example, are paired-end) and put those lines of our table in a new file. First, we need to check to make sure that the pattern we're searching for ("PAIRED") only appears in the column where we expect it to occur (column 3). We know from earlier that there are only two paired-end samples in the file, so we can `grep` for "PAIRED" and see how many results we get.

```
$ grep PAIRED SraRunTable.txt | wc -l
```

```
2
```

There are only two results, so we can use "PAIRED" as our search term to extract the paired-end samples to a new file.

```
$ grep PAIRED SraRunTable.txt > SraRunTable_only_paired_end.txt
```

Exercise

Sort samples by load date and export each of those sets to a new file (one new file per unique load date).

Solution

```
grep 25-Jul-12 SraRunTable.txt > SraRunTable_25-Jul-12.txt
grep 29-May-14 SraRunTable.txt > SraRunTable_29-May-14.txt
```

Key Points

- `grep` is a powerful search tool with many options for customization.
- `>`, `>>`, and `|` are different ways of redirecting output.
- `command > file` redirects a command's output to a file.
- `command >> file` redirects a command's output to a file without overwriting the existing contents of the file.
- `command_1 | command_2` redirects the output of the first command as input to the second command.

(../03-working-with-files/index.html)

(../05-writin script