‹ (../03-working-with-files/index.html) Introduction to the Command Line for Genomics (../) › (../05-writin script

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 being aNy nucleotide at that position in the DNA sequence.

We'll search for strings inside of our fastq files. Let's first make sure we are in the correct directory:

Bash

```
$ cd ~/shell_data/untrimmed_fastq
```

Suppose we want to see how many reads in our file have really bad segments containing 10 consecutive unknown nucleotides (Ns).

📌 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 your 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 (https://datacarpentry.org/wrangling-genomics/02-quality-control/index.html).

Let's search for the string NNNNNNNNNN in the SRR098026 file:

Bash

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

This command returns a lot of output to the terminal. Every single line in the SRR098026 file that contains at least 10 consecutive Ns is printed to the terminal, regardless of how long or short the file is. 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. The `-A` argument returns 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:

Bash

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

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

Output

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

```
@SRR098026.245 HWUSI-EAS1599_1:2:1:1:2:801 length=35
GNATNACCACTTCCAGTGCTGANNNNNNNGGGATG
```

2. `grep -B1 AAGTT *.fastq`

```
SRR097977.fastq-@SRR097977.11 209DTAAXX_Lenski2_1_7:8:3:247:351 length=36
SRR097977.fastq:GATTGCTTTAATGAAAAAGTCATATAAGTTGCCATG
--
SRR097977.fastq-@SRR097977.67 209DTAAXX_Lenski2_1_7:8:3:544:566 length=36
SRR097977.fastq:TTGTCCACGCTTTTCTATGTAAAGTTTATTTGCTTT
--
SRR097977.fastq-@SRR097977.68 209DTAAXX_Lenski2_1_7:8:3:724:110 length=36
SRR097977.fastq:TGAAGCCTGCTTTTTTATACTAAGTTTGCATTATAA
--
SRR097977.fastq-@SRR097977.80 209DTAAXX_Lenski2_1_7:8:3:258:281 length=36
SRR097977.fastq:GTGGCGCTGCTGCATAAGTTGGGTTATCAGGTCGTT
--
SRR097977.fastq-@SRR097977.92 209DTAAXX_Lenski2_1_7:8:3:353:318 length=36
SRR097977.fastq:GGCAAAATGGTCCTCCAGCCAGGCCAGAAGCAAGTT
--
SRR097977.fastq-@SRR097977.139 209DTAAXX_Lenski2_1_7:8:3:703:655 length=36
SRR097977.fastq:TTTATTTGTAAAGTTTTGTTGAAATAAGGGTTGTAA
--
SRR097977.fastq-@SRR097977.238 209DTAAXX_Lenski2_1_7:8:3:592:919 length=36
SRR097977.fastq:TTCTTACCATCCTGAAGTTTTTTCATCTTCCCTGAT
--
SRR098026.fastq-@SRR098026.158 HWUSI-EAS1599_1:2:1:1:1505 length=35
SRR098026.fastq:GNNNNNNNNCAAAGTTGATCNNNNNNNNNTGTGCG
```

Redirecting output

`grep` allowed us to identify sequences in our FASTQ files that match a particular pattern. All of these sequences were printed to our terminal screen, but in order to work with these sequences and perform other operations 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 are taking what would ordinarily be printed to the terminal screen and redirecting it to another location. In our case, we want to print this information to a file so that we can look at it later and use other commands to analyze 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`.

Bash

```
$ 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.

Bash

```
$ wc bad_reads.txt
```

Output

```
  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`.

Bash

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

Output

```
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

Bash

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

Output

```
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.

Bash

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

Output

```
537 bad_reads.txt
```

Bash

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

Output

```
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.

Bash

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

Output

```
537 bad_reads.txt
```

Bash

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

Output

```
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:

Bash

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

Output

```
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.

Bash

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

Output

```
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 unintended results.

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 thus 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.

Bash

```
$ 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`.

If we don't want to create a file before counting lines of output from our `grep` search, we could directly pipe the output of the grep search to the command `wc -l`. This can be helpful for investigating your output if you are not sure you would like to save it to a file.

Bash

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq | wc -l
```

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.

Writing for loops

Loops are key to productivity improvements through automation as they allow us to execute commands repeatedly. Similar to wildcards and tab completion, using loops also reduces the amount of typing (and typing mistakes). Loops are helpful when performing operations on groups of sequencing files, such as unzipping or trimming multiple files. We will use loops for these purposes in subsequent analyses, but will cover the basics of them for now.

When the shell sees the keyword `for`, it knows to repeat a command (or group of commands) once for each item in a list. Each time the loop runs (called an iteration), an item in the list is assigned in sequence to the **variable**, and the commands inside the loop are executed, before moving on to the next item in the list. Inside the loop, we call for the variable's value by putting `$` in front of it. The `$` tells the shell interpreter to treat the **variable** as a variable name and substitute its value in its place, rather than treat it as text or an external command. In shell programming, this is usually called "expanding" the variable.

Sometimes, we want to expand a variable without any whitespace to its right. Suppose we have a variable named `foo` that contains the text `abc`, and would like to expand `foo` to create the text `abcEFG`.

Bash

```
$ foo=abc
$ echo foo is $foo
foo is abc
$ echo foo is $fooEFG      # doesn't work
foo is
```

The interpreter is trying to expand a variable named `fooEFG`, which (probably) doesn't exist. We can avoid this problem by enclosing the variable name in braces (`{` and `}`, sometimes called "squiggle braces").

Bash

```
$ foo=abc
$ echo foo is $foo
foo is abc
$ echo foo is ${foo}EFG      # now it works!
foo is abcEFG
```

Let's write a for loop to show us the first two lines of the fastq files we downloaded earlier. You will notice the shell prompt changes from `$` to `>` and back again as we were typing in our loop. The second prompt, `>`, is different to remind us that we haven't finished typing a complete command yet. A semicolon, `;`, can be used to separate two commands written on a single line.

Bash

```
$ cd ../untrimmed_fastq/
```

Bash

```
$ for filename in *.fastq
> do
> head -n 2 ${filename}
> done
```

The for loop begins with the formula `for <variable> in <group to iterate over>`. In this case, the word `filename` is designated as the variable to be used over each iteration. In our case `SRR097977.fastq` and `SRR098026.fastq` will be substituted for `filename` because they fit the pattern of ending with .fastq in the directory we've specified. The next line of the for loop is `do`. The next line is the code that we want to execute. We are telling the loop to print the first two lines of each variable we iterate over. Finally, the word `done` ends the loop.

After executing the loop, you should see the first two lines of both fastq files printed to the terminal. Let's create a loop that will save this information to a file.

Bash

```
$ for filename in *.fastq
> do
> head -n 2 ${filename} >> seq_info.txt
> done
```

Note that we are using `>>` to append the text to our `seq_info.txt` file. If we used `>`, the `seq_info.txt` file would be rewritten every time the loop iterates, so it would only have text from the last variable used. Instead, `>>` adds to the end of the file.

Using Basename in for loops

Basename is a function in UNIX that is helpful for removing a uniform part of a name from a list of files. In this case, we will use basename to remove the `.fastq` extension from the files that we've been working with.

Bash

```
$ basename SRR097977.fastq .fastq
```

We see that this returns just the SRR accession, and no longer has the .fastq file extension on it.

Output

```
SRR097977
```

If we try the same thing but use `.fasta` as the file extension instead, nothing happens. This is because basename only works when it exactly matches a string in the file.

Bash

```
$ basename SRR097977.fastq .fasta
```

Output

```
SRR097977.fastq
```

Basename is really powerful when used in a for loop. It allows to access just the file prefix, which you can use to name things. Let's try this.

Inside our for loop, we create a new name variable. We call the basename function inside the parenthesis, then give our variable name from the for loop, in this case `${filename}`, and finally state that `.fastq` should be removed from the file name. It's important to note that we're not changing the actual files, we're creating a new variable called name. The line > echo $name will print to the terminal the variable name each time the for loop runs. Because we are iterating over two files, we expect to see two lines of output.

Bash

```
$ for filename in *.fastq
> do
> name=$(basename ${filename} .fastq)
> echo ${name}
> done
```

Exercise

Print the file prefix of all of the `.txt` files in our current directory.

Solution

Bash

```
$ for filename in *.txt
> do
> name=$(basename ${filename} .txt)
> echo ${name}
> done
```

One way this is really useful is to move files. Let's rename all of our .txt files using `mv` so that they have the years on them, which will document when we created them.

Bash

```
$ for filename in *.txt
> do
> name=$(basename ${filename} .txt)
> mv ${filename}  ${name}_2019.txt
> done
```

Exercise

Remove `_2019` from all of the `.txt` files.

Solution

Bash

```
$ for filename in *_2019.txt
> do
> name=$(basename ${filename} _2019.txt)
> mv ${filename} ${name}.txt
> done
```

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.
- for loops are used for iteration.
- `basename` gets rid of repetitive parts of names.

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

› (../05-writin scripts