Manipulating files on the Linux command line

Oxford Biomedical Data Science Training Programme
University of Oxford
2020-04-28

Overview

- Linux streams
- Combining commands
- Looping over files
- Pattern matching/regular expressions
- Text processing tools
  - sort/uniq
  - grep
  - sed
  - awk

Linux streams

- Streams are mechanisms to move data from one place to another
- Standard streams
  - Standard in (stdin): the default place where commands listen for information
  - Standard out (stdout): the default place for output to go
  - Standard error (stderr): the default place for error output to go
- Unless redirected stdout and stderr both print to the terminal
- Pipes (|) connect the standard output of one command to the standard input of another

Redirecting input, output and error

```
command1 < file1          # input file1 to command1
command1 > file1          # standard output of command1 to file1
command1 >> file1         # append standard output of command1 to file1
command1 2> file2         # error output of command1 to file2
command1 > /dev/null      # discard standard output of command1
```

Combining commands

```
command1; command2         # run command1, then command2

command1 && command2       # run command2 if command1 is successful

command1 || command2       # run command2 if command1 is not successful

command1 | command2        # pipe stdout of command1 to stdin of command2
```

Loops

`for i in {1..5}; do COMMAND; done` # for i in 1-5, run a command

e.g. `for i in {1..5}; do echo "number" $i; done`

```
number 1
number 2
number 3
number 4
number 5
```

`for i in (i = 1; i <= 10; i += 2); do COMMAND; done` # for i from 1-10, by 2 (iterates by 2), run a command

Loops

You can also iterate over files

`for file in *.txt; do COMMAND; done` # for all .txt files, run a command

e.g. `for file in *.txt; do wc -l $file; done`

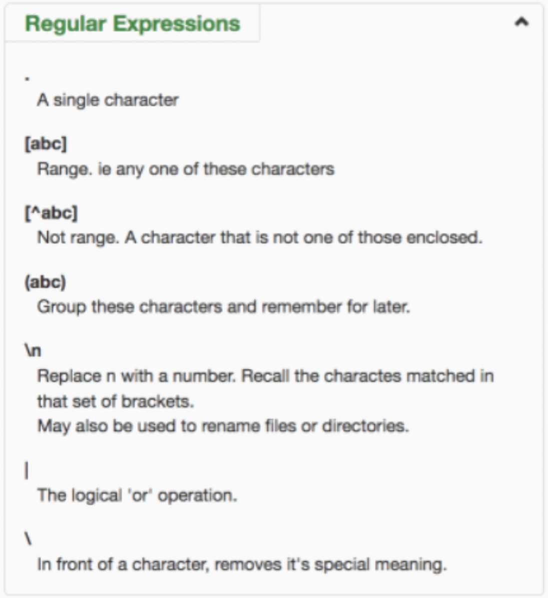
```
4 file1.txt
11 file2.txt
```

`for file in *.bed; do head -n 5 $file; done` # for all .bed files, output the top 5 lines

Regular expressions

- Sequence of characters that define a search pattern
- Often used to find sets of files with related names or to find lines in a file that contain a particular character pattern
- Most simple form - glob
    - Wildcard character *
    - Example *.jpg - finds all files with .jpg extension
- Common syntax for more specific patterns used in Unix and many other programming languages

Regular expressions



Regular Expressions

.
A single character

[abc]
Range. ie any one of these characters

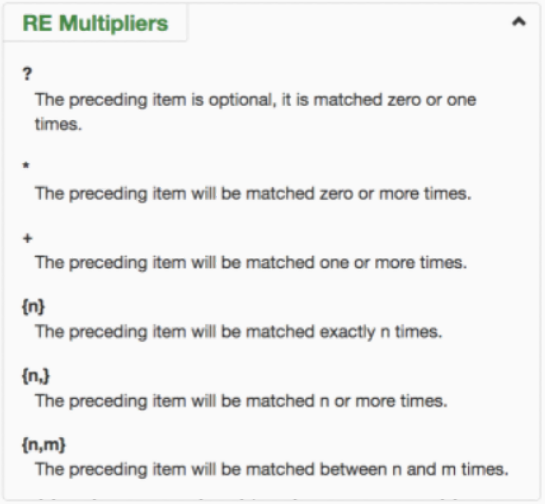
[^abc]
Not range. A character that is not one of those enclosed.

(abc)
Group these characters and remember for later.

\n
Replace n with a number. Recall the charactes matched in that set of brackets.
May also be used to rename files or directories.

|
The logical 'or' operation.

\
In front of a character, removes it's special meaning.



RE Multipliers

?
The preceding item is optional, it is matched zero or one times.

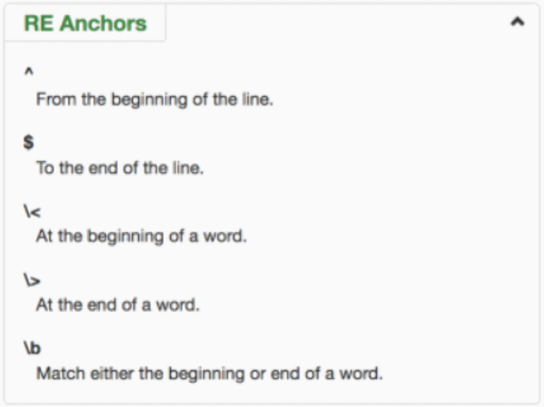
*
The preceding item will be matched zero or more times.

+
The preceding item will be matched one or more times.

{n}
The preceding item will be matched exactly n times.

{n,}
The preceding item will be matched n or more times.

{n,m}
The preceding item will be matched between n and m times.



RE Anchors

^
From the beginning of the line.

$
To the end of the line.

\<
At the beginning of a word.

>
At the end of a word.

\b
Match either the beginning or end of a word.

What to match? How many times? Where in the text?

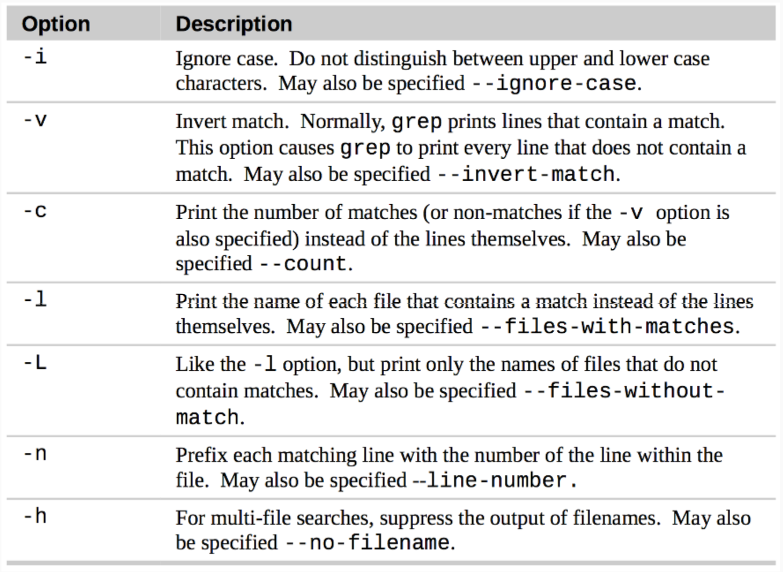
http://www.regexe.com - useful website for testing out regular expressions

Searching within files using grep

- Search for text within files that matches a specified pattern
- Line based
- Regular expressions are used to encode pattern

Searching within files using grep

Options for grep



Option	Description
`-i`	Ignore case. Do not distinguish between upper and lower case characters. May also be specified `--ignore-case`.
`-v`	Invert match. Normally, `grep` prints lines that contain a match. This option causes `grep` to print every line that does not contain a match. May also be specified `--invert-match`.
`-c`	Print the number of matches (or non-matches if the `-v` option is also specified) instead of the lines themselves. May also be specified `--count`.
`-l`	Print the name of each file that contains a match instead of the lines themselves. May also be specified `--files-with-matches`.
`-L`	Like the `-l` option, but print only the names of files that do not contain matches. May also be specified `--files-without-match`.
`-n`	Prefix each matching line with the number of the line within the file. May also be specified --`line-number`.
`-h`	For multi-file searches, suppress the output of filenames. May also be specified `--no-filename`.

`grep [options] regex [file...]`

`grep -i "exception" pipeline.log` # print out lines containing "exception" in pipeline.log, ignore case

`grep -c "chr1" p300.bed` # print the number of lines that contain "chr1" in p300.bed

`grep -v "#" pedigree.vcf` # print out the lines that don't contain a hash

Grep examples using regular expressions

```
grep "chr[1-9]$" file1.txt
```

Will select lines containing: "chr1", "chr5"

Won't select lines containing "chr11", "chr15"

```
grep "^[a-g][a-z]*[t|h|e]$" file1.txt
```

Will select lines containing: "cat", "fish", "giraffe"

Won't select lines containing: "dog", "cats", "rabbit"

```
grep --extended-regexp "ENSMUSG000000[0-9]{5}\b" file1.txt
```

```
grep "ENSMUSG000000[0-9]\{5\}\b" file1.txt
```

same as above but using basic regular expression syntax

Will select lines containing: "ENSMUSG00000051951", "ENSMUSG00000025903"

Won't select lines containing: "ENSMUSG000000051951" (extra 0)

Exercise 1

Data can be found in week1/bedtools (use the copy that you made in your own user directory - {USER}/obds/week1/bedtools)

1. Print the number of lines in each bed file
2. Print the number of lines in each bed file using a loop
3. Extract all the entries from chromosome 5 of "cpg.bed" into a new file
4. Extract all the entries EXCEPT those from chromosome 5 of "cpg.bed" into a new file
5. Extract all the entries from "chr1" and "chr6" from cpg.bed into a new file

Manipulating text files in Linux

- Linux philosophy: one tool for one task
- Can be combined using pipes
- Tools for processing text files
    - paste - displays corresponding lines of multiple files side-by-side
    - join - joining lines of two files on a common field
    - cut - select sections of each line from file
    - sort - sorts files using one or more keys
    - uniq - reports or filters out repeated lines in a file
    - tr - translate or delete characters
    - grep - search for lines which match a specified pattern
    - sed - a stream editor
    - AWK - programming language designed for text processing

Paste, join and cut

- `paste file1.txt file2.tsv` # merge files line by line
- `join file1.txt file2.tsv` # merge files using a sorted shared key
- `cut -f3 file1.tsv` # print selected parts of lines from file (field 3 in this example)

Sorting files and removing duplicates

- `sort file1.txt file2.txt file3.txt > sorted.txt`   # sort the three files

- `sort --key=1,1 --key=2n filename`
  `sort -k1,1 -k2n filename`

  start at field 1, end at field 1
  field two is the second sort key, n = numeric sort

- `uniq filename`   # remove duplicate lines, matching lines need to be adjacent (sort input first)

- `uniq -c filename`   # prefix lines by the number of occurences

tr - translating or deleting character

- Transliteration is the process of changing characters from one alphabet to another
- Character-based search-and-replace operation
- Can also be used to delete specific characters

`echo "lowercase letters" | tr a-z A-Z` # replace lowercase with uppercase letters

```
LOWERCASE LETTERS
```

`echo AATGATACGGCGA | rev | tr ATGC TACG` # get reverse complement of a sequence

tr - command for translating or deleting characters

`cat file1.txt | tr -s ' ' '\t' > file2.txt` # replace spaces with tabs and output to new file, -s option for search and replace

`echo "Python is a Programming language" | tr -d 'Pyt'` # -d option to search and delete, delete any instances of letters P, y and t (case sensitive)

```
hon is a rogramming language
```

sed - the stream editor

- sed works by making only one pass over the input, and is consequently efficient
- sed's ability to filter text in a pipe distinguishes it from other types of editors
- Text substitution:
  - Match a regular expression against the content of the pattern space
  - If found, replace matched string with replacement

sed examples

Command format:

`sed s/regexp/replacement/[flags]` # s for substitute

Examples:

- `sed 's/chr1/1/g'` # replace chr1 with 1, g flag - apply the replacement to all matches to the regexp
- `sed 's/chr1|Chr1/1/g' file.txt` # replace every instance of chr1 or Chr1 with 1
- `sed 's/^[ \t]*//' file.txt'` # trim leading whitespaces and tabs
- `sed 's/[ \t]*$//' file.txt` # trim trailing whitespaces and tabs
- `sed 's/^[ \t]*//;s/[ \t]*$//' file.txt` # trim leading and trailing whitespaces and tabs

sed examples

Command format:

`sed s/regexp/replacement/[flags]` # s for substitute

Examples:

- `sed '/^$/d' file.txt` # delete blank lines, NB: no 's'
- `sed '$d' file.txt` # delete the last line
- `sed -i 1d file.txt` # delete line1 in place
- `sed -n 42p file.txt'` # print a specific line (42 in this case)
- `sed -n '2~4p' file.txt'` # extract every fourth line starting at the second line (extract sequence from FASTQ file)

https://www.grymoire.com/Unix/Sed.html - sed tutorial

Exercise 2 - manipulating text files

Data can be found in /ifs/obds-training/apr20/shared/week1/bash (use the copy in your user directory)

1. Paste together the two counts files to create a new counts table file
2. Join together the two counts files to create another counts table file
3. Convert the joined counts table from space delimited to comma delimited
4. Delete the last two lines from sample2.counts - save output in a new file
5. With the coding_gene_region.bed file, count how many genes are on each chromosome (assume each gene only listed once for each chromosome)

Exercise 2 - manipulating text files

6. Remove the old header from your joined counts table and insert a new header into the file using sed

AWK

- Programming language for processing tabular text data
    - line oriented
    - default column delimiter is whitespace
- Conditional expressions
- Arithmetic operations
- String manipulation
- One liners

Basic structure

```
awk 'Pattern { action }' <file>
```

test to perform on each line

action to perform if test is true

Pattern can be blank, perform action for every line

Most common action is print e.g.

```
awk '{ print $0 }' <file>
```

print the entire line (each line is split into fields using the field separator; each field is represented by a positional variable e.g. $1, $2, $3; $0 contains the entire line)

Patterns

- BEGIN - perform this action before the first line
- END - perform this action after the last line
- Expressions
- Regular expressions - matches (~), doesn't match (!~)

```
awk 'BEGIN {print "The File Contents:"} {print $0}' file1.txt
```

```
The File Contents:
line 1
line 2
line 3
```

Patterns

`awk '$1 == "chr1" { print $2 }' coding_gene_region.bed` # if field 1 is chr1, print out field 2

`awk '($1 ~ /^chr1$/) { print $0 }' coding_gene_region.bed` # print out all lines where field 1 is chr1

Operations

- Arithmetic operations: +, -, *, /, %
- String concatenation

BED file format:

```
chr1   213941196  213942363
chr1   213942363  213943530
chr1   213943530  213944697
chr2   158364697  158365864
chr2   158365864  158367031
chr3   127477031  127478198
chr3   127478198  127479365
chr3   127479365  127480532
chr3   127480532  127481699
```

```
awk '{print $3-$2}' test.bed
```
subtract start from stop interval

```
awk '{print $1":"$2"-"$3}' test.bed
```
reformat bed data to IGV format

chr:start-stop

Variables

- X=5 (= assignment operator)
- $ used to signify positional variables (fields)
- Strings must be placed in double quotes
- Variables inside double quotes are not evaluated
- AWK uses \ (backslash) to escape special characters

```
awk 'BEGIN { SUM = 0 } { SUM += $3-$2 } END { print SUM }'
coding_gene_region.bed
```

set variable SUM to zero before reading first line

for each line, increment SUM by interval size

after last line, print variable SUM

Built-in variables

- FS - input field separator
- OFS - output field separator (space)
- RS - record separator (usually newline character \n)
- ORS - output record separator
- FILENAME - name of file being read
- NF - number of fields
- NR - number of records

`awk 'BEGIN { FS = ","; OFS = "\t"} {print $1, $2, $3}' joined_counts.counts` # convert comma delimited to space delimited; field separator inserted where commas are in the print statement

`awk 'NF < 4 { print $0 }' <file>` # print rows with less than 5 columns

`awk 'NR >= 100 && NR <= 150 { print $0 }' <file>` # print rows 100-150

Exercise 3 - AWK

Data can be found in /ifs/obds-training/apr20/shared/week1/bash (use the copy in your user directory)

1. Find the length of each gene in coding_gene_region.bed
2. Pad each interval by 100 bp on each side (make sure the output is tab separated)
3. Calculate the total number of bases covered by annotations
4. Convert the coding_gene_region.bed file into a GFF formatted file (see https://www.ensembl.org/info/website/upload/gff.html)

AWK help

https://www.tutorialspoint.com/awk/index.htm

https://www.shortcutfoo.com/app/dojos/awk/cheatsheet