

Exercises Linux Tutorial - Part 3

Stefan Wyder
URPP Evolution
University of Zurich

August 2014

Content

Exercise 1: Parallelizing jobs using GNU parallel
Exercise 2,3: Introduction into awk
Exercise 4: Regular expression



Exercise 1: GNU parallel

All the commands we were running in the preceding tutorials were running on a single processor. But some jobs are time-consuming and one would like to speed them up by using all the processors/cores of a modern computer. One way is to use GNU parallel (<http://www.gnu.org/software/parallel/>).

Install GNU parallel on Ubuntu:

```
sudo apt-get install parallel
```

For Mac OS X users only:

On Mac OS X GNU parallel is not available by default, you can install it with your favorite package manager e.g if you have installed homebrew, simply type: `brew install parallel`
Alternatively, parallel is also available under fink.

Once installed you can use parallel to speed up your analyses. Parallel also takes care not to overload the system (preventing to make it slow and unresponsive).

GNU parallel often replaces a for loop and runs faster as jobs are run in parallel - one per CPU core.

This example runs fastqc on all .fq files in the directory:

```
find *.fq | parallel fastqc {} --outdir .
```

This example indexes all bam files:

```
find *.bam | parallel samtools index {}
```

You can use parallel for your own scripts. The following line runs the script `run_single_plink.sh` in parallel by running it once per chromosome.

```
seq 1 22 | parallel -j 20 --progress run_single_plink.sh --chromosome {}
```

GNU parallel has many functions e.g. for controlling the output, limiting resources and executing on a remote computer. To get an idea of the possibilities and options of GNU parallel check `man parallel_tutorial`. As usual options are explained under `man parallel`. As for most GNU tools a detailed manual is available under `<info parallel>`.

You find an excellent page on GNU parallel with several bioinformatics examples (BLAST, own scripts) under <https://www.biostars.org/p/63816/> and <http://davetang.org/muse/2013/11/18/using-gnu-parallel/>

When trying out `parallel --dryrun` is convenient for testing as it shows the actual commands without running them.



Exercise 2: awk

awk is handy for using in shell pipes. Most people just use it for one-liners and in shell pipes. For more complicated problems it is often better to use a standard programming language (e.g. python) that is easier to read and maintain. I will try to give you an idea of what awk is capable of.

First let us prepare an example data file

We use a genome annotation file for the model plant *Arabidopsis thaliana* – The file is in GFF format that is a standard format to describe genome annotations.

1. Download the annotation file from the Arabidopsis TAIR website:

```
wget  
ftp://ftp.arabidopsis.org/home/tair/Genes/TAIR10_genome_release/TAIR10_gff3/TAIR10_GFF3_genes.gff
```
2. Extract the first 100 lines into a new file `At.gff`:

```
head -100 TAIR10_GFF3_genes.gff > At.gff
```
3. Look at the file, the file is tab-separated with 9 columns/fields:

```
<chromosome> <source> <feature> <start> <end> <score> <strand> <frame> <group>
```

Non-available values are denoted with a dot ('.'). You find a detailed format description under <http://genome.ucsc.edu/FAQ/FAQformat.html#format3>

The basic structure of awk

pattern {action}

is the basic pattern in awk. For each line, it will check whether the pattern is true/fulfilled and only then perform <action>. You can also have multiple patterns like

pattern1 {action1} pattern2 {action2} pattern3 {action3}

awk reads the input (file) line by line and by default, each line is a record and is broken into fields that are called \$1,\$2,\$3 etc. \$0 is the whole line.


```
awk '$3 == "mRNA" {print $0}' At.gff
```

reads in the file At.gff and prints out all the lines where column 3 is equal to mRNA.

By default the output is separated by a blank space.

Commented Examples

Now that we know the awk structure we will revise what we have done in part 1 of the Linux/bash tutorial.

 Try to understand the following examples. Run them on your own machine and check the output. Modify them and check whether they produce the expected result.

A pattern can be a condition or a regular expression.

<code>awk '\$3 == "mRNA" {print \$0}' At.gff</code>	Print all lines where column 3 is equal to 'mRNA'
<code>awk '\$3 == "mRNA"' At.gff</code>	The same as above - we can leave out {print \$0}
<code>awk '\$3 != "mRNA"' At.gff</code>	Print all lines where column 1 is NOT equal to 'mRNA'
<code>awk '\$4 >= \$5' At.gff</code>	Shows lines where column 4 is larger or equal than column 5. Can be used to spot errors in GFF files.
<code>awk '/gene/' At.gff</code>	Here the pattern is a regular expression (denoted by '/'). The simplest regular expression is a simple string. Gives the same result as <code>grep "gene" At.gff</code>
<code>awk '\$3 ~/^[a-f]/' At.gff</code>	Print each line where column 3 matches the regular expression. This made-up example prints all lines where column 3 starts with any of the letters abcdef. CDS is not included as regexps are case-sensitive.
<code>awk '\$3 !~/^[a-f]/' At.gff</code>	Print each line where column 3 does NOT match the regular expression.

awk '\$9 ~/^ID=AT[34]G/' TAIR10_GFF3_genes.gff	Print each line where column 9 starts with 'ID=AT3G' or 'ID=AT4G'
awk '{sub(/Note/, "Type", \$0);print}' At.gff	Replace 'Note' with 'Type' sub replaces only once per line.
awk '{gsub(/scarlet ruby puce/, "red")}'	Replace 'Scarlet' or 'ruby' or 'puce' with 'red' In contrast to sub gsub (global sub) replaces all occurrences.

A pattern can also comprise multiple conditions

awk '\$4>1000 && \$5<6000' At.gff	Shows all features at positions between 1000 and 6000
awk 'NR > 10 && NR <= 17' At.gff	Print lines 11 to 17 of a file NR is the built-in variable for the line number
awk '\$4<2000 \$4>4000' At.gff	Shows all features with positions smaller than 2000 or larger than 4000

The following examples do have a non-existing pattern what is considered to match every input record. Therefore their action will be executed at every line.

awk '{print \$1,\$2,\$4,\$5}' At.gff	awk splits a tab- or space-delimited file into variables. \$0 contains the full input line, \$1 column 1, \$2 column 2, and so on.
awk '{print \$4,\$2,\$3,\$1}' At.gff	Rearrange columns 1 and 4.
awk '{sum=\$4+\$5;print sum,\$0}' At.gff	Print out the sum of columns 4 and 5 before the line
awk '{sum=\$4+\$5;print "Sum:"sum,\$0}' At.gff	As above, but we print a leading "Sum:" in front of the sum
awk '{\$2=\$4+\$5;print}' At.gff	Replace column 2 by the sum of columns 4 and 5

The patterns <BEGIN> and <END> are executed only once, either before the first input record is read or after all the input is read. <BEGIN> is convenient to set variables like the input field separator (FS) and the output field separator (OFS). By default the output is separated by a blank space.

awk 'BEGIN {OFS="\t"} {print \$4,\$2,\$3,\$1}' At.gff	We want to create tab-separated output (default is space-separated output)
awk 'BEGIN {FS=",";OFS="\t"} {print \$4,\$2,\$3,\$1}' indB.txt	We have to change the input field separator as the input file indB.txt is comma-separated (",")

<END> is used to print out a result.

awk '{SUM+=\$4} END {print SUM}' At.gff	Print out the sum of column 4 over the whole file In awk variables do not need to be declared. When first used they are 0 when used a number or "" when used as a string.
awk '{SUM+=\$4} END {print SUM/NR}' At.gff	Print out the mean of column 4 NR: variable containing the number of rows
awk '/gene/ {c++} END {print c}'	Count lines containing "gene"

At.gff	
awk '\$4>max {max=\$4; maxline=\$0} END {print max,maxline}' At.gff	Print the maximum value of column 4 observed in the file
awk '\$3!="chromosome" && \$5-\$4>max {max=\$5-\$4; maxline=\$0} END {print max,maxline}' At.gff	Print the longest feature that is not a chromosome

Awk is a Unix filter, it can be combined with other Unix tools by Piping

awk 'BEGIN {FS="\t"} {print NF}' At.gff sort uniq -c	Handy to check proper formatting – same number of columns over all lines. NF is the built-in variable containing the number of field/columns
zcat file.gff.gz awk '\$1=="chr1" && \$2>34 && \$2<453 {print \$0}' wc -l	Count the number of entries fulfilling the conditions in a gzipped file.



Some exercises to practice awk:

1. How many exons are larger than 300nt?
2. Write an awk one-liner that prints the shortest feature annotated in the file At.gff.
(Hint: if you don't declare a variable it is 0)
3. Find out which is the shortest exon a) in the file At.gff b) in the file TAIR10_GFF3_genes.gff.

You find many more awk one-liners under
<http://www.pement.org/awk/awk1line.txt>



Exercise 3 (Advanced): More awk functions

Most people use awk just for one-liners like the examples from Exercise 2. This exercise is for people considering to learn more awk otherwise you can skip this exercise.

Some more advanced awk examples:


We can also loop over the columns/fields of a line

awk 'NR > 1 {s=0; for (i=2; i<=NF; i++) s=s+\$i; print s}' file.txt	Starts from the second line (file with header!) and prints, for each line, the sum of all the fields starting from the 2 nd column.
---	--

We can also read in two different files using the getline function

<pre>awk 'BEGIN{while((getline<"file1.txt") >0)lin[\$1]=\$0}\$1 in lin {print \$0"\t"lin[\$1]}' file2.txt > output.txt</pre>	Join two files on column 1 Here we use an array to make a lookup table from file1.txt (containing the whole line) which we then use with file2.
---	--

What is possible with awk?

 Open the awk manual at https://www.gnu.org/software/gawk/manual/html_node/ (gawk is simply the GNU implementation of awk). Have a look at the built-in functions for https://www.gnu.org/software/gawk/manual/html_node/Built_002din.html , particularly the numeric and string functions.

Some frequently used string functions I have listed in a table:

Action	Function name
extract	substr()
search text	index(), match()
replace	sub(), gsub(), gensub()
split	split()



Exercise 4: Regular Expressions


Here we practice regular expression using R. We could use also the shell (grep/egrep, sed, awk), python (with the re module) or any other programming language.

You find the (complicated) documentation of regexps in R under <http://stat.ethz.ch/R-manual/R-devel/library/base/html/regex.html>

A speciality of R is that the meta-characters '\ ' and '[' need to be doubled. E.g. the wildcard for an digit is \\d or [[:digit:]]

Functions that use regular expressions in R

sub(), gsub()	Make some substitutions in a string sub() replaces only the first occurrence, gsub() all occurrences
grep() regexpr(), gregexpr()	Extract some value
grepl()	Detect the presence of a pattern
strsplit	Split a string according to a pattern

 Launch R. Run the following examples and check the output.

glob2rx() is a useful function for people beginning to use regular expressions: it takes a wildcard as used by most shells and returns an equivalent regular expression

```
> glob2rx("*.txt")  
[1] "^.*\\.txt$"
```

```
> glob2rx("*.t*")  
[1] "^.*\\.t"
```

Replacing text / Substitution

sub() and gsub() are for replacing text. sub() replaces only the first occurrence, whereas gsub() replaces all occurrences. Their arguments are <pattern> (the regular expression), <replacement> (the substitution pattern or capturing index) and <x> (the string to work on). <x> can be a single string or a vector containing multiple strings.

The simplest regular expression pattern are just literal characters:

```
> string <- "23 mai 2000"  
> sub("2000", "2014", x=string)  
[1] "23 mai 2014"
```

We can remove any white spaces (space, tab, end-of-line)

```
> gsub('\\s', '', x=" Hello World ")
[1] "HelloWorld"
```

But regular expressions have much more abilities. Here we use a more expressive notation `[:digit:]` as a wildcard for any number and `[:alpha:]` for any alphanumeric character.

```
> sub(pattern = "([[:digit:]]+\\.\\d{3})\\d+", replacement = "\\1\\2",
x = "34.73322532")
[1] "34.733"
```

```
> regexp <- "([[:digit:]]{2}) ([[:alpha:]]+) ([[:digit:]]{4})"
> sub(pattern = regexp, replacement = "\\1", x = string)
# returns the first part of the regular expression
[1] "23"
> sub(pattern = regexp, replacement = "\\2", x = string)
# returns the second part
[1] "mai"
> sub(pattern = regexp, replacement = "\\3", x = string)
# returns the third part
[1] "2000"
```

In the above simple example we could also use `strsplit()` to split the character vector into substrings separated by ".".

```
> strsplit(string, split=" ")
[[1]]
[1] "23"    "mai"   "2000"
```

Escape special characters '\$ * + . ? [] ^ { } | () \

If you look for a meta-character '\$ * + . ? [] ^ { } | () \' , precede them with a doubled backslash. As they have a double meaning we need '\\' to interpret them as an ordinary characters.

In Arabidopsis, isoforms of a gene X are called like X.1, X.2, X.3. Sometimes we need a list of genes which we can achieve using a regular expression. We have It even works with X.11 or X.100.

```
> sub(pattern = "([^\\.]+)\\.[:digit:]", replacement = "\\1", x =
"AT5G11100.3")
[1] "AT5G11100"
sub(pattern = "([^\\.]+)\\. [0123456789]", replacement = "\\1", x =
"AT5G11100.3")
```



```
[1] "AT5G11100"
sub(pattern = "([A-z0-9]+).[0-9]", replacement = "\\1", x =
"AT5G11100.3", perl=T)
[1] "AT5G11100"
```

There are often many different ways to write a working regular expression. With `[]` we can define our own character class. The `^` in square brackets means that it will match any character except the one written. Thus `[^\\.]` means any character but a dot. This way is a safe way to capture everything until the first dot.

Finding text

`grep()` tells you whether a regular expression matches the input string `x`. If it matches it returns 1 and if it does not match integer(0).

```
grep(pattern = "21", x = "21 Aug 2014")
[1] 1
> grep(pattern = "\\d{2} \\w+ \\d{4}", x = "21 Aug 2014")
[1] 1
> grep(pattern = "\\d{2} \\w+ \\d{4}", x = "hello")
integer(0)
```

Format Checking

Regular expressions are often used to check correct formatting. We want to check whether a string is correctly formatted doing `grepl`:

```
> grepl(pattern = "\\d{2} \\w+ \\d{4}", x = "21 Aug 2014")
[1] TRUE
```

or, alternatively, in a more expressive notation:

```
> grepl(pattern = "[[:digit:]]{2} [[:alpha:]]+ [[:digit:]]{4}", x =
"21 Aug 2014")
[1] TRUE
```

The following one is false since there is only one digit in the first number in `x`:

```
> grepl(pattern = "\\d{2} \\w+ \\d{4}", x = "1 Aug 2014")
[1] FALSE
```

`grepl` tells us that the pattern/regexp is matching our string. Here we check a single string, but `x` could also be a vector containing many elements. `grepl` will return a logical vector telling us for each element of the vector whether it matches.

regexpr() and gregexpr() are more verbose versions of grep(). In addition to the position of the match (counting from left) they also return the length of the match. -1 is the result if the regexp does not match at all. Again, regexpr() only returns the first occurrence whereas gregexpr() returns all occurrences.

```
> regexp <- "[[:digit:]]{2} ([[:alpha:]]+) ([[:digit:]]{4})"
> string <- "blabla 23 mai 2000 blabla 18 mai 2004"
> regexpr(pattern = regexp, text = string)
[1] 8
attr(,"match.length")
[1] 11
attr(,"useBytes")
[1] TRUE
> gregexpr(pattern = regexp, text = string)
[[1]]
[1] 8 27
attr(,"match.length")
[1] 11 11
attr(,"useBytes")
[1] TRUE
```

Application in Bioinformatics

Regular expressions are used a lot for data mangling (format conversion). In addition, regular expressions are often used for parsing, e.g. if you want to extract information of a BLAST report (e.g. the Sequence ID or the E-value). Regular expressions can also be used to identify Sequence motifs, e.g. to search for a motif with 3 basic AAs in 5 positions (see PROSITE).

Final Comments

Trial and error: Sometimes regular expressions do not behave as expected. In case of difficulties try to start simple, test parts of the regular expression and combine them once the subparts work. Often it also helps to do two rounds of replacements. Another level of complication is that there are 2 different types of regular expression, in R the default are 'extended regular expressions' (perl=FALSE) but there are also Perl-like regular expressions (perl=TRUE) that look different from the default type. It is easy to confuse them. Last, there unfortunately also differences between languages so that sometimes we have to find the correct version by trial and error.

If you use regular expressions a lot in R it might be worth looking into the 'stringr' package in CRAN that is a package dedicated to operations on strings.



1. Modify some of the input strings and patterns/regular expression and check whether they produce the expected results. You can e.g. add additional text to the input string or remove parts of the regexp.
2. Try out the 2 online tools (url given below). Choose the one that appeals most to you. Develop a regexp to remove the noninformative parts of sample names:
X20120401_Jaggirun31_1367.05.1_05.RCC
X20120401_Jaggirun31_1482.05.1_08.RCC

Sources & Links

Online tools to try regular expressions

<http://regex101.com/>

<http://www.regexr.com/>

Regular expressions in R

http://en.wikibooks.org/wiki/R_Programming/Text_Processing#Functions_which_use_regular_expressions_in_R

Regular expression using sed

<http://www.grymoire.com/Unix/Sed.html#uh-4>