Bulk data processing: Pattern matching, variables, subshells and loops, file permission, Link

7. Pattern matching

We provide specific files or strings to most of the UNIX commands. Using pattern matching (also called regular expressions) the command can match a group of files or strings. We have already been doing this for the ls command where we have listed files matching patterns using the *-sign. This can also be done for example for the grep command which then can extract lines matching a pattern rather than matching an exact string of characters.

Unfortunately pattern matching is done differently by different programs. Here, we will introduce pattern matching of UNIX file paths and pattern matching in commands like grep.

7.1 Pattern matching UNIX paths

We mentioned that we already have used the *-pattern for matching any number of any non- line break character in UNIX file paths.

The square brackets ([]) are used to match one out of a set of characters, or a range of characters ([0-9] matches zero to nine).

```
$ ls
sample1 . fa sample2 . fa sample3 . fa sampleA . fa
$ ls sample [12]. fa
sample1 . fa sample2 . fa
$ ls sample [0 -9]. fa
sample1 . fa sample2 . fa sample3 . Fa
```

Tools for pattern matching

? Matches any single character except newline
* Match any number (zero or more)
{} Can match one of several comma-delimited words
[] Match one of the enclosed characters
[0-9] Match any number
[a-z] Match any lower case character
[a-zA-Z] Match any character lower and upper case
\ Turn off the special meaning of the following character

```
$ ls
sample11 . fa sample12 . fa sample21 . fa sample22 . fa
$ ls sample1 ?. fa
sample11 . fa sample12 . Fa
```

The curly brackets ({}) can be used to match one of several words. It can also be useful for creating multiple files or directories in one command.

```
$ ls
sample11 . fa sample12 . fa sample21 . fa sample22 . fa
$ ls sample {11 ,22}. fa
sample11 . fa sample22 . Fa

$ ls
$ mkdir { analysis , results } _run1
$ ls
analysis_run1 results_run1
```

7.2 Using regular expressions with grep

Pattern matching in grep can be used any time you want to extract subsets from files based on patterns rather than exact strings.

For example, if you have different isoforms of genes in your FASTA file, where the different isoforms for a particular gene are distinguished with a ” i1”, ” i2”... suffix, and you only are interested in the first isoform, those can be extracted in the following way (the genes_with_isoforms.fa file is shown in figure 7.3)

```
. Match any single character except newline
* Match any number of the preceding pattern
^ Match the following expression at beginning of line
$ Match the preceding regular expression at end of line
\+ Match the preceding pattern one or more times
[] Match one of the enclosed characters
[0-9] Match any number
[a-z] Match any lower case character
[a-zA-Z] Match any character
\ Turn of the special meaning of the following characters
```

```
make isoforms.fn
>geneA_i1
ATCGATCGATCG
>geneA_i2
ATCGATCGATGACTCG
>geneB_i1
ATCGATCGATCGAAA
>geneB_i2
ATATGCTAGCCGATCGATCG
>geneB_i3
ATATATTAGCGA
```

$ grep - A1 " >.* _1 " genes_with_isoforms . Fa

7.3 Unix Variables

A variable in UNIX is a way of linking a string of text to a value. This value can then be reused or changed. Variables are defined by writing the string, directly followed by an equality sign and the value (no white spaces are allowed around the equality sign). The value of the variable is retrieved by typing a dollar sign in front of the variable name (shown in the example below). The value can also be retrieved by typing out curly brackets around the variable name (also shown in the example below).

$ a_variable = " This is my text "
$ echo $a_variable
This is my text
We retrieve the value stored in a_variable by prepending the dollar sign

$ echo $ { a_variable }
This is my text
This is the complete form for retrieving the value

$ echo a_variable
a_variable
If we not prepend the dollar sign a_variable is used as a regular string

$ echo " The content of the variable is : $ { a_variable } "
The content of the variable is : This is my text
Variables can be used inside strings

One useful way of using variables is to store long paths, instead of repeatedly typing them out

A variable can store a value for later reuse. This is both useful if you are using a particular value repeatedly, or if you have some way to automatically generate values which you want to reuse. You will see examples of both cases in the section about loops, and the coming chapter about scripts.

A characteristic of variables is that they can store one value at a time, while this value can be different at any time. You can still retrieve the current value of the variable by using the variables name.
$ head ../ other_analysis / first_experiment / data / raw_file1 . fa
...
$ head ../ other_analysis / first_experiment / data / raw_file2 . fa
...
$ data_path =../ other_analysis / first_example / data /
$ head $ { data_path }/ raw_file1 . fa
...

$ head $ { data_path }/ raw_file2 . Fa

A more permanent alternative would be to make a file link to the directory. We will come back to variables in the remaining sections of this chapter.

7.4 Subshells

Subshells are used for running commands within commands. This is often used to calculate a value within a line, which then can be assigned to a variable. There are two ways to write subshells:

- **Parenthesis syntax: $(command)**
- **Back-tick syntax: ‘command‘**

It is recommended to use the first syntax. This allows for nested subshells (subshells within subshells). A demonstration of subshells used to assign a value to variable is shown below.

```
$ grep -c " ^ > " nucleotides . fa
173
$ entry_count = $ ( grep -c " ^ > " nucleotides . fa )
echo $ { entry_count }
173
```

7.5 Loop: iterating loop on file

Looping over a set of files Before you go into this section, make sure that you know how to make variables and subshells in UNIX. The preferred way of iterating over a set of files (going through the files one by one) is to use pattern matching to decide the set of files. For example, if we want to perform a certain set of operations on all files in a directory, we can use the *-wild card matching.

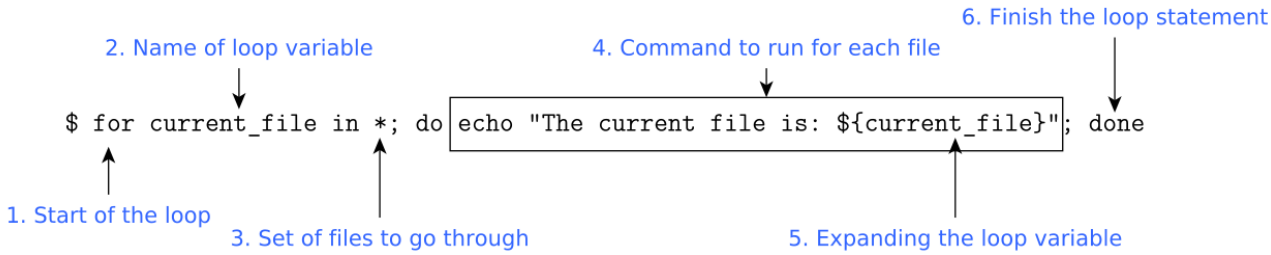
```
$ ls
file1. txt file2. txt a_file.txt a_file2.txt

$ for file in *; do echo $ { file } done
file1 . txt
file2 . txt
a_file . txt
a_file2 . Txt

$ for file in *;  do echo " The current file is : $ { file } " ; done

The current file is : file1 . txt
The current file is : file2 . txt
The current file is : a_file . txt
```

The current file is : a_file2 . txt



1. The for statement, starting the loop
2. The name of the loop variable which will get the values you are looping over (here, "current_file")
3. Pattern matching the files you want to loop over - Followed by a semi-colon and the ”do” statement
4. One or more commands to run for each file - Each ending with a semi-colon
5. The use of the loop variable, which will contain the name of the file it is currently looping past
6. The done statement ends the loop statement

$ ls
nucl1 . fa nucl2 . fa nucl3 . fa nucl4 . fa
$ for f in *; do count = $ (grep -c " ^ > " $ { f }); echo " $ { f }: $ { count } entries " ; done
nucl1 . fa : 173 entries
nucl2 . fa : 215 entries
nucl3 . fa : 98 entries
nucl4 . fa : 133 entries

In this loop two commands are run per iteration. (Are you keeping track of the variables and subshells?)
• count=$(grep -c "^>" ${f}) uses a subshell to retrieve the number of entries present in the fasta file and store this number in the variable count.
• echo "${f}: ${count} entries" prints the file name together with the number of entries found within it.

7.5.1 Iterating over the lines of a file

A common way of iterating over the content of a file is to use a while loop. A while loop uses the following syntax:
cat file | while read line ; do command ; done

If you for example want to print the lines of a file containing the lines ”First line”, ”Second line” and ”Third line” one by one, you could do the following:

cat my_file . txt | while read line ; do echo " Line : $ { line } " ; done
Line : First line
Line : Second line
Line : Third line

Exercise
Copy folder-7 from workshop Day-2. Inside it you will find a set of ten FASTA files, each containing ten FASTA entries. We will work with those ten files at the same time. Start out by checking how they are structured.

1. For some commands, we are able to get information about all ten files just by using pattern matching. First, get the number of FASTA entries for all the files using grep and the * pattern instead of file path.
$ grep " ^ > " *
Try the same thing with the wc command. Which of the subsets contains the most data? Why is the number of lines the same, but the number of characters in the files different?

2. Next, we want to calculate the total number of nucleotides in each FASTA. In this case we need to change approach. Try what you get if attempting the following:
$ grep -v " ^ > " * | tr -d " \ n " | wc -m

Here, we attempted to get the nucleotides for all the files similarly to how we did in the stream chapter. In this case, the pattern matching will not give us output for each single file. The tr -d and wc -m commands are not able to distinguish each file, and instead calculates the total number of nucleotides.

3. In order to calculate information for the individual files we will use a for-loop. Take a look at the illustration of a for-loop earlier in this chapter. Do you understand the different parts? We will build this loop step by step.

4. First specify what you want to loop over. Use a pattern which matches all of them. Also, specify the name of the variable which will get the values from the different files. End the start statement with ; and do. At this point, our statement looks something like the following:
for target_fs in *; do

5. Next, we will specify what the loop should do, and then terminate it with the done statement. As a start, let’s just print the file names that we loop over. Do this by entering an echo statement with the target_fasta as argument and end the statement with ; and done:

for target_fs in *; do echo ${target_fs}; done
This loop should be working. Try it!

6. Let's print the number of nucleotides in each file. We can now run this command within the for-loop. If we add this command after the echo command we will get the following:
**$ for target_fs in *; do echo $ { target_fs }; **
grep -v " ^ > " $ { target_fs } | tr -d " \ n " | wc -m ; done
(The backslash means that the line continues on the next line - it was to long to fit within a page width). Run it. For each cycle of the loop we first print the name of the file, and then its number of nucleotides. If we want to retrieve other information, we could extend this line with other commands.

7 We can make the output a bit nicer. Let's try to get the output for each file on a single line. In order to do this, we need to calculate the number of nucleotides before printing the information. We could use a subshell to calculate and store the number of nucleotides in a variable. Then, we can control how we want to print it.

$ for target_fs in *; do \
nucl_count = $ (grep -v " ^ > " $ { target_fs } | tr -d " \ n " | wc -m); \
echo " File : $ { target_fs } has $ { nucl_count } nucleotides " ; done

Can you see how we used a subshell to calculate the number of nucleotides, before storing it in the variable nucl_count for later usage?

7.7 Processing multiple files
You likely want to use loops any time where you are working with several samples. This is also true when using terminal-based bioinformatic software. Here, we will show a very simple example with the program seqtk, but this applies for any software. Remember how we converted multi-line FASTA files to single-line format? We used the command:

seqtk seq -l 0 multi_line.fa > single_line.Fa

We can use the same command to get multi-line format by adjusting the -l 0 argument to a particular width (for example -l 60). You task is to reformat the ten FASTAs. The biologist asking for the data have some demands:

• The data should be in a single file, containing all the sequences from the ten samples.
• You should be able to distinguish from which sample the different sequences came. This is best done by adding annotation to the header lines.
• The data should be in multi-line format.

Note: Think about the problem for a while. What changes do you need to make to each FASTA? Which step or steps is best done as a part of a loop? Are some steps easier to do before or after combining the different samples? If you get stuck on this one, ask the teachers and you will get some hints. After delivering the data you got an additional request. Could you provide a subset of the data only containing sequences from sample 2, 5, and 8?

8.1 Bash and scripting

Bash is a programming language, and the most common language used in the UNIX termi nals. It is also the type of UNIX terminal that we have been using in this course. Here, we will start scratching the surface of scripting - putting together small computer programs to run sequences of commands. Scripting is not necessarily as intimidating as it might sound. Building scripts and programs (scripts are in principle small programs) can be a highly complex task involving large teams of people, but could as well just be putting together some UNIX commands for later reuse.

Two main advantages of collecting commands into scripts are:
• **Reproducibility** - If you gather the commands you have been running to process a specific set of data in a script, you (or someone else) will easily be able to re-run the exact same steps at a later point.

• **Creating your own tools** - Often you find yourself re-running the same commands to process data in different contexts. You can gather those commands into a script, allowing you to perform the same task in a single command by invoking your script. Scripts are commonly used on computational clusters (like UPPMAX) by their queue systems which use them to run your commands at a later time.

8.2 First bash script

hello_world.sh
```
#!/bin/bash
echo "Hello world :)"
```

The script is shown in figure 8.1. (sh is a common file ending for this kind of scripts, and comes from the term shell, which is another name for the terminal). There are two lines in the script. The second one is a simple echo command, similar to what we have been using before. The first line tells the script which interpreter that the computer should use to read the script. The line starts with #!, followed by the absolute path to the interpreter which the computer uses to read Bash programs. The initial #! is the same for all scripts, while the interpreter is different for each computer language.

In order to run the script we must first set the appropriate permissions - We need to have read (r) and execution (x) rights for the script. Then, we can simply type out the path of the script to run it. Note - If the script is in the present working directory, the path is specified using the current directory sign (.).

$ ls -l
-rw - rw -r - - 1 reaper reaper 33 apr 10 08:11 hello_world . Sh

$ ./ hello_world . sh
bash : ./ hello_world . sh : Permission denied

$ chmod + x hello_world . sh
$ ./ hello_world . sh
Hello world :)

The .sh-suffix is commonly used for scripts written in Bash or other terminal languages.

Comments in Bash scripts

Sometimes, we want to have text in our scripts which isn't interpreted as commands. This can be done by adding a hash-sign (#)

Gathering processing steps in a script

seq_summary.sh

```
#!/bin/bash
# "nucleotides.fa" must be in the same directory as this script
echo "Calculating information about nucleotides.fa"
echo "Number of entries"
grep -c "^>" nucleotides.fa
echo "Number of lines"
wc -l < nucleotides.fa
echo "Number of nucleotides"
grep -v "^>" nucleotides.fa | tr -d "\n" | wc -m
```

Running of script

$ ls
seq_summary.sh nucleotides . fa
$ chmod + x seq_summary.sh
$ ./ seq_summary.sh
Calculating information about nucleotides . fa
Number of entries
14956
Number of lines
29912
Number of nucleotides

747800000

Providing input to a script

In some cases our analyses are specific for one single project, and will only be reproduced for that particular case. In other cases, the steps will be repeated many times in different projects - Similarly to how we have been using commands like grep, sort and cut in many different contexts.

It turns out that we with a few changes can take the script in figure 8.4 and generalize it so that it could calculate information about any FASTA file.

Inside the script, there is a special variable called argv, which contains all arguments that you have provided to the command line when running it in the terminal. For example, if you have a script called my_script.sh, and run it as shown below, we are able to retrieve the provided value additional_text from the argv-variable within the script.

$ ./ my_script . sh additional_text

The argv values can be accessed within the script by typing ${number} with number re-placed by the position of the argument. In the previous case, ${0} will give ./my_script.sh and ${1} will give additional_text.

fasta_stats.sh

```
#!/bin/bash
# Read an arbitrary FASTA file through argv
echo "Calculating information about ${1}"
echo "Number of entries"
grep -c "^>" ${1}
echo "Number of lines"
wc -l < ${1}
echo "Number of nucleotides"
grep -v "^>" ${1} | tr -d "\n" | wc -m
```

Running of script

```
$ ls
fasta_stats . sh nucleotides . fa other_nucleotides . fa
$ chmod + x fasta_stats . sh
$ ./ fasta_stats . sh nucleotides . fa
Calculating information about nucleotides . fa
Number of entries
14956
Number of lines
29912
```

Number of nucleotides
747800000
$ ./ fasta_stats . sh other_nucleotides . fa
Calculating information about other_nucleotides . fa
Number of entries
6672
Number of lines
13344
Number of nucleotides
333600000

8.2 Building a useful script on your own (*)
As a final exercise, let’s build a useful script which we could use later to retrieve information from our files. If you have an idea in mind of something which could be interesting or useful for you, feel free to try doing it! You could discuss with the teachers how to approach it, and what would be reasonable. One useful example could be to do a FASTA entry retriever. This script could take FASTA file and an ID as input. It should then output the sequence and useful information for that particular entry. Information which might be useful could be:

• The header of the ID (in a nicer format than the regular FASTA header)
• The sequence belonging to the ID.
• Information about the sequence (length, number of different letters)
• If using the potato reference genome, you could m

File permissions
File permissions are used in UNIX to control which users that are allowed to use files and directories in different ways. They can also be used to protect files from accidental editing or removal.

For files, the file permissions control who is allowed to:
• Read the file content
• Edit/re-write the file content
• If the file is a program or script - Who can run/execute the program

$ ls -l Documents
-rw -r - - - - - 1 user user 687438 sep 16 19:02 MyArticle . pdf
-rw -r - - - - - 1 user user 2481860 may 13 14:05 MyReport . Pdf

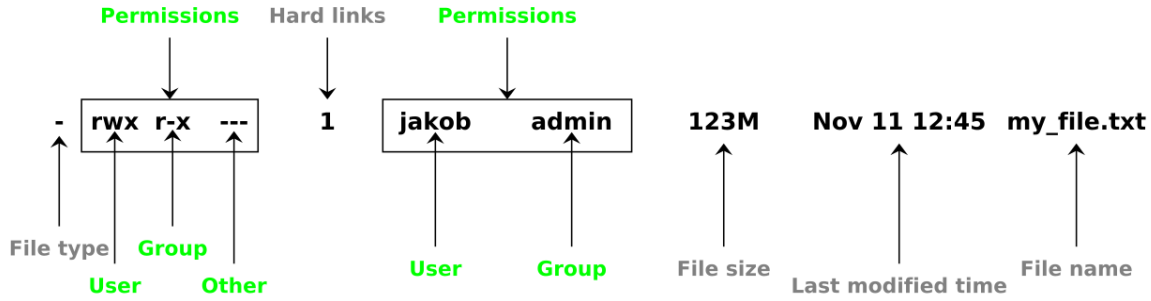


Figure : Explanation of the different parts of the ls -l output. Parts related to permissions are marked in green.

Chmod: change file mode
Command usage: chmod <new_permission> <target_file>

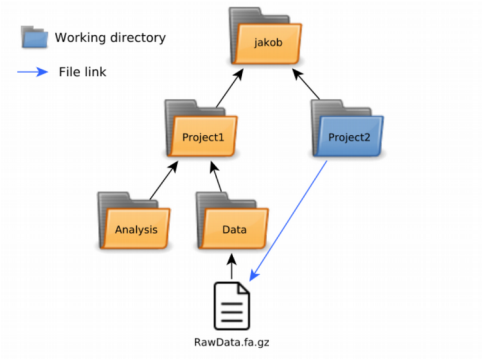
$ ls -l
$ chmod + x
$ ls -l
$ chmod -w
$ ls -l

$ chmod -w raw_data . fa
$ ls -l
$ rm raw_data . fa
rm : remove write - protected regular file ’ raw_data ’?
Type n and enter to not remove it

wget: The non-interactive network downloader/getter
Command usage: wget <internet_address>
wget ftp://ftp.ensemblgenomes.org/pub/release-41/plants/gff3/arabidopsis_thaliana/Arabidopsis_thaliana.TAIR10.41.chromosome.1.gff3.gz

Symbolic file links
When working with bioinformatic data, it is not uncommon for some files of interest to be re-used in many different parts of the project, or even in different projects. To avoid making copies of the data to all those different locations, or writing extensive file paths to reach the data, you can use a file link as a ’shortcut’ to the data from a more convenient location.

ln -s make links between files
Command usage: ln -s <path_to_target> [<link_location>]
The ln -s command is used to create so called ’soft links’ to files. Those can be thought of as shortcuts to a file using a relative path or absolute path. If the command is used without the -s flag, you get a hard link to the physical memory location of the computer that the target file is using. In this course we will only be using the soft links.



$ pwd
/ home / reaper / Project2
# less can read . gz - files
$ less ../ Project1 / Data / RawData . fa . gz
$ ln -s ../ Project1 / Data / RawData.fa.Gz

$ ls -l
lrwxrwxrwx 1 reaper reaper 12 apr 7 13:58 RawData . fa . gz -> \
../ Project1 / Data / RawData . fa . Gz

# Note the beginning ’l ’ for link filetype
$ less RawData . fa . Gz

Exercises

We will use a FASTQ file (named MhaptRNASeq.fastq) in the exercises form folder8 of Day2 of workshop. It is highly recommended to write-protect raw files, such as FASTQ, to safeguard the original sequence data that was delivered by the sequence facility.

1. Download the files to your home directory
This is the regular procedure. We will use the wget command to download the FASTQ file from the Linux server:
$ cp ~/workshop/Day2/MhaptRNASeq . fastq . Gz

2. ungzip the file for the rest of the exercise.
gunzip MhaptRNASeq.fastq.gz

3. Make a copy (using cp) of the MhaptRNASeq.fastq file and call it MhaptRNASeq2.fastq:
$ cp MhaptRNASeq.fastq MhaptRNASeq2.Fastq

4. Change file permissions -w of the MhaptRNASeq2.fastq that you just generated:
$ chmod -w MhaptRNASeq2 . Fastq

5. Use ls -l to compare the permissions of the two fastq files. Make sure that you understand the difference of the file permissions before you continue. Also note the difference in the results when using ls and ls -l.

6. Remove the MhaptRNASeq2.fastq file using the rm command:
$ rm -i MhaptRNASeq2 . fastq
Press y to remove the file.

Check the files in the directory by using ls again to ensure that MhaptRNASeq2.fastq was removed. Now write protect the original MhaptRNASeq.fastq file before continuing with the exercises.

Use Symbolic links approach
In order to organize your data files in an efficient and clear way it can be convenient to use symbolic links. This will keep the original raw files in their place but still easily accessible in the different projects where they may be used.

1. Make a new directory called Data using the mkdir command and use the cd command to enter into the Data directory:
$ mkdir Data
$ cd Data

2. We want to make a symbolic link in the Data directory that points to the MhaptRNASeq.fastq file.
$ ln -s ~/ MhaptRNASeq.fastq MhaptRNASeq.Fastq
Check with both ls and ls -l to see how the newly generated link works.

3. Use head to check the contents of the file that the symbolic link points to:
$ head MhaptRNASeq.Fastq

Notice that you can work with the symbolic link in exactly the same way as if it was a copy of the orignal file. In addition you save precious harddisk space since many of the files you are using could be many Gb is size.