Bioinformatics Practical Part 1

Introduction to the Linux command line and Bash scripting

Solutions

MP235

SS 2022

1 Navigating the File System

1.1 Listing the Items of a Directory

Solution 1.1

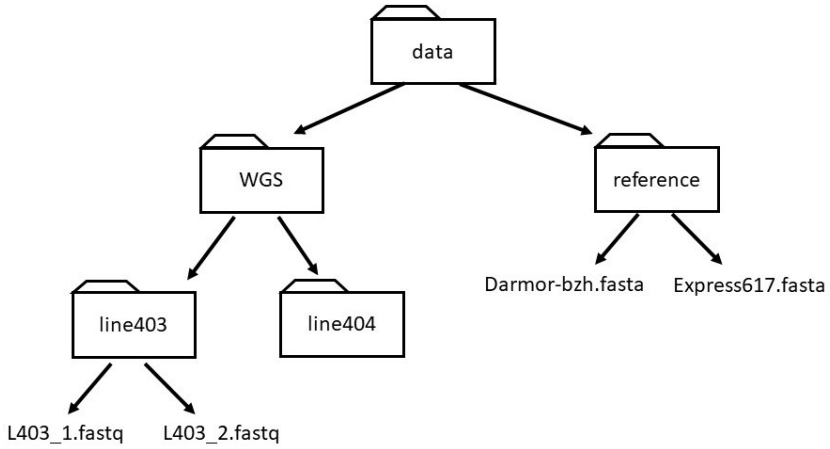
The `-l` option makes `ls` use a long listing format, showing not only the file/directory names but also additional information, such as the file size and the time of its last modification. If you use both the `-h` option and the `-l` option, this makes the file size 'human readable', i.e. displaying something like 5.3K instead of 5369.

1.2 Where are you and where do you want to go?

Solution 1.2

1. No: `.` stands for the current directory.
2. No: `/` stands for the root directory.
3. No: Amanda's home directory is `/Users/amanda`.
4. No: this command goes up two levels, i.e. ends in `/Users`.
5. Yes: `~` stands for the user's home directory, in this case `/Users/amanda`.
6. No: this command would navigate into a directory called home within the current directory, if it exists.
7. Yes: unnecessarily complicated, but correct.
8. Yes: shortcut to go back to the user's home directory.
9. Yes: goes up one level

1.3 Relative Paths



Solution 1.3

1. No: there is a directory called WGS in `~/data`.
2. Yes: `../WGS` refers to going up one level from the current working directory and then down into the directory called WGS, i.e, `~/data/WGS`.
3. No: this is the content of `~/data/reference`, the directory we started in.
4. No: this is the content of `~/data/WGS/line403`, but the command given in the exercise only gets us one level above this folder.

1.4 `ls` Reading Comprehension

Solution 1.4

1. No: pwd is not the name of a directory.
2. Yes: `ls` without a directory argument lists files and directories in the current directory.
3. Yes: uses the absolute path explicitly. Note that the `-F` option does not change the output, when the directory only contains regular files.
4. Yes: short options can be strung together and the `.`, though in this case unnecessary, refers to the current working directory.

2 Working With Files and Directories

2.1 Creating Files and Directories

Solution 2.1

The most recently changed file (`my_second_file`) is listed last when using `-rt`. This can be very useful for finding your most recent edits or checking to see if a new output file was written.

Solution 2.2

`ls -l my_directory`
Both files have a size of 0 bytes (5th field from the left), since they are empty files.

Solution 2.3

The first two sets of commands achieve this objective. The first set uses relative paths to create the top-level directory before the subdirectories.

The third set of commands will give an error because the default behavior of `mkdir` won't create a subdirectory of a non-existent directory: the intermediate level folders must be created first.

The fourth set of commands achieve this objective. The `-p` option, followed by a path of one or more directories, will cause `mkdir` to create any intermediate subdirectories as required.

The final set of commands generates the 'raw' and 'processed' directories at the same level as the 'data' directory.

2.2 Moving Files to Another Folder

Solution 2.4

```
$ mv alignment1.bam alignment2.bam ../alignments
```

Recall that `..` refers to the parent directory (i.e. one above the current directory) and that `.` refers to the current directory.

2.3 Copying and Renaming Files

Solution 2.5

1. No. While this would create a file with the correct name, the incorrectly named file still exists in the directory and would need to be deleted.
2. Yes, this would work to rename the file.
3. No, the period (`.`) indicates where to move the file, but does not provide a new file name.
4. No, the period (`.`) indicates where to copy the file, but does not provide a new file name; identical file names cannot be created.

Solution 2.6

We start in the `/Users/jamie/data` directory, and create a new folder called `variants`. The second line moves (`mv`) the file `variants.vcf` to the new folder (`variants`). The third line makes a copy of the file we just moved. The tricky part here is where the file was copied to. Recall that `..` means 'go up a level', so the copied file is now in `/Users/jamie`. Notice that `..` is interpreted with respect to the current working directory, not with respect to the location of the file being copied. So, the only thing that will show using `ls` (in `/Users/jamie/data`) is the `variants` folder.

1. No, see explanation above. `variants-saved.vcf` is located at `/Users/jamie`
2. Yes
3. No, see explanation above.
   `variants.vcf` is located at `/Users/jamie/data/variants`
4. No, see explanation above. `variants-saved.vcf` is located at `/Users/jamie`

2.4 Removing Files and Directories Safely

Solution 2.7

```
$ rm -ir my_directory
rm: descend into directory 'my_directory'? y
rm: remove regular empty file 'my_directory/my_first_file'? y
rm: remove regular empty file 'my_directory/my_second_file'? y
rm: remove directory 'my_directory'? y
```

The `-i` option will prompt before (every) removal (use "y" to confirm deletion or "n" to keep the file). The Unix shell doesn't have a trash bin, so all the files removed will disappear forever. By using the `-i` option, we have the chance to check that we are deleting only the files that we want to remove.

2.5 Copying Multiple Files

Solution 2.8

If given more than one file name followed by a directory name (i.e. the destination directory must be the last argument), `cp` copies the files to the named directory.
If given three file names, `cp` throws an error such as the one below, because it is expecting a directory name as the last argument.

Error:

```
cp: target 'basil.dat' is not a directory
```

2.6 List file names Matching a Pattern

Solution 2.9

1. No: this gives us all file names that start with any number of characters followed by "`vulgar`" which is then followed by **exactly** two characters and "`.fasta.fai`" at the end, i.e., "`vulgar??`" can expand to "`vulgaris`", but not "`vulgare`".
2. Yes: this gives us all filesnames that contain a "`v`" that is preceded and/or followed by any number of characters. The three file names we are looking for are the only ones that contain a "`v`".
3. No: all file names end on `.fasta.fai`, not `.fasta`.
4. Yes: as opposed to the first example, the `*` wildcard can also replace zero characters, therefore `vulgar?*` can expand to `vulgaris` and `vulgare`.

Solution 2.10

```
mv *.sam alignments
```

Jamie needs to move her files `leaf_rna.sam` and `root_rna.sam` to the `alignments` directory. The shell will expand `*.sam` to match all `.sam` files in the current directory. The `mv` command then moves the list of `.sam` files to the `alignments` directory.

3 Pipes and Filters

3.1 Counting File Contents

Solution 3.1

If we run the command `wc -l *.fai`, the `*` in `*.fai` matches zero or more characters, so the shell turns `*.fai` into a list of all `.fai` files in the current directory. The resulting output shows the number of lines for each individual file and also the total number across all files in the last line of the output.

3.2 Sorting File Contents

Solution 3.2

The `-n` option specifies a numerical rather than an alphanumerical sort.

3.3 Routing the Output of a Command into a File

Solution 3.3

In the first example, using '>', the string 'hello' is written to `testfile01.txt`, but the file gets **overwritten** each time we run the command.
We see from the second example that the `>>` operator also writes 'hello' to a file (in this case `testfile02.txt`), but **appends** the string to the file if it already exists (i.e. when we run it for the second time).

Solution 3.4

Option 3 is correct. For option 1 to be correct we would only run the `head` command. For option 2 to be correct we would only run the `tail` command. For option 4 to be correct we would have to pipe the output of `head` into `tail -n 2` by doing

```
$ head -n 3 Brassica_napus.fasta.fai | tail -n 2 > Bnapus_subset.txt
```

3.4 Routing the Output of a Command into another Command

Solution 3.5

Option 4 is the solution. The pipe character `|` is used to connect the output from one command to the input of another. `>` is used to redirect standard output to a file.

Solution 3.6

The `head` command extracts the first 4 lines from `numbers.txt`. Then, the last 3 lines are extracted from the previous 4 by using the `tail` command. With the `sort -rn` command those 3 lines are sorted numerically in reverse order (in descending order) and finally, the output is redirected to the newly created file `final.txt`. The content of this file can be checked by executing `cat final.txt`. The file should contain the following lines:

```
22
19
2
```

Solution 3.7

The answer is 11. After excluding the header using `tail -n +2`, we can use `sort` and `uniq` to get rid of duplicates and then use `wc -l` to count the number of lines.

```
$ cut -f 2 sample_info.txt | tail -n +2 | sort | uniq | wc -l
11
```

3.5 Recap: Removing Files

Solution 3.8

1. This would only remove .csv files with one-character names
2. This is the correct answer
3. The shell would expand * to match everything in the current directory, so the command would try to remove all matched files and an additional file called .csv
4. The shell would expand *.* to match all files with any extension, so this command would delete all files

4 Loops

4.1 Writing your own Loop

Solution 4.1

```
$ for loop_variable in 0 1 2 3 4 5 6 7 8 9
> do
>     echo $loop_variable
> done
0
1
2
3
4
5
6
7
8
9
```

4.2 Loop Comprehension

Solution 4.2

The first code block lists every file on each loop iteration since the `*` wildcard is expanded to include every file ending in `.fai` each time.

The second code block, however, lists a different file on each loop iteration. The value of the datafile variable is evaluated using `$datafile`, and is then listed using `ls`.

Solution 4.3

4 is the correct answer. * matches zero or more characters, so any file name starting with the letter H, followed by zero or more other characters will be matched.

4.3 Saving Files within Loops

Solution 4.4

1 is the correct answer. Each file name is echoed before the text from each file gets written to the `herbs.dat` file. However, the file gets overwritten on each loop iteration, so in the end, the `herbs.dat` file only contains the text from the `oregano.dat` file.

Solution 4.5

3 is the correct answer. `>>` appends to a file, rather than overwriting it with the redirected output from a command. Given that the output from the `cat` command has been redirected, nothing is printed to the screen.

4.4 Doing a Dry Run

Solution 4.6

The second version is the one we want to run. This prints to screen everything enclosed in the quote marks, expanding the loop variable name because we have prefixed it with a dollar sign. It also does not modify nor create the file `all.dat`, as the `>>` is treated literally as part of a string rather than as a redirection instruction.

The first version appends the output from the command `echo cat $datafile` to the file `all.dat`. This file will just contain the list; `cat basil.dat`, `cat mint.dat` and `cat oregano.dat`.

4.5 Nested Loops

Solution 4.7

We have a nested loop, i.e. a loop within another loop, so for each species in the outer loop, the inner loop (the nested loop) iterates over the list of experiments, and creates a new directory for each combination.

5 Bash Scripting

5.1 Using Variables in Bash Scripts

Solution 5.1

The correct answer is 2.
The special variables `$1`, `$2` and `$3` represent the command line arguments given to the script, such that the commands run are:

```
$ head -n 1 basil.dat mint.dat oregano.dat
$ tail -n 1 basil.dat mint.dat oregano.dat
```

The shell does not expand `"*.dat"` because it is enclosed by quote marks. As such, the first argument to the script is `"*.dat"` which gets expanded within the script by head and tail.

Solution 5.2

```
# Script to find the date when a file was last updated,
# where the date is the second field in the third row,
# with a space as the delimiter.
# This script accepts any number of file names
# as command line arguments

# Loop over all files
for file in $@
do
    echo "$file last updated:"
    # Extract date
    head -n 3 $file | tail -n 1 | cut -d " " -f 2
done
```

Solution 5.3

```
# Shell script which takes two arguments:
#    1. a directory name
#    2. a file extension
# and prints the name of the file in that directory
# with the most lines which matches the file extension.

wc -l $1/*.$2 | sort -n | tail -n 2 | head -n 1
```

The first part of the pipeline, `wc -l 1/*.2 | sort -n`, counts the lines in each file and sorts them numerically (largest last). When there's more than one file, `wc` also outputs a final summary line, giving the total number of lines across all files. We use `tail -n 2 | head -n 1` to throw away this last line.

With `wc -l 1/*.2 | sort -n | tail -n 1` we'd only see the final summary line: we can build our pipeline up in pieces to be sure we understand the output.

5.2 Script Reading Comprehension

Solution 5.4

In each case, the shell expands the wildcard in `*.dat` before passing the resulting list of file names as arguments to the script.

Script 1 would print out a list of all files containing a dot in their name. The arguments passed to the script are not actually used anywhere in the script.

Script 2 would print the contents of the first 3 files with a .dat file extension. `$1`, `$2`, and `$3` refer to the first, second, and third argument respectively.

Script 3 would print all the arguments to the script (i.e. all the .dat files), followed by .dat. `$@` refers to all the arguments given to a shell script.

5.3 Debugging Scripts

Solution 5.5

The `-x` option causes bash to run in debug mode. This prints out each command as it is run, which will help you to locate errors. In this example, we can see that echo isn't printing anything. We have made a typo in the loop variable name, and the variable `datfile` doesn't exist, hence returning an empty string.

6 Finding Things

6.1 Using `grep`

Solution 6.1

The correct answer is 3, because the `-w` option looks only for whole-word matches. The other options will also match 'silique_pericarp'.

Solution 6.2

```
grep -w $1 -r .  | cut -d / -f 2 | sed 's/\t/:/' | cut -d :  -f 1,3
--output-delimiter=$'\t' > $1.txt
```

We search (`grep`) for all instances of the gene name, which had been provided as the argument (`$1`), and since we provide the complete gene name, we use the `-w` option. The `-r` option searches every file within the current working directory from which the script is run (`.`). From this we get a list of files in which the gene name occurred, together with the content of the lines in which the gene name occurred. All file names start with `./` since they are located in the present working directory. To get rid of this character string we pipe the output of the `grep` command to `cut -d / -f 2`, however this step does not have to be in second place, it only needs to come at some point **after** the `grep` command. The output is piped to the `sed` command which replaces tabs with colons, leading to the following change in output:

from

```
<file name>:<gene name>    <expression value>
```

to

```
<file name>:<gene name>:<expression value>
```

Now we can cut each line in the output using a colon as the delimiter (`-d :`), retaining only the first and third field of each line (`-f 1,3`) and then separating the retained fields with a tab (`--output-delimiter=$'\t'`).
Lastly we redirect the output into a file using the gene name which was provided as an argument (`$1`), with the extension `.txt`.

6.2 Finding Files

Solution 6.3

1. Option 1 is correct. Putting the match expression in quotes prevents the shell expanding it, so it gets passed to the find command.
2. Option 2 is also works in this instance because the shell tries to expand *.dat but there are no *.dat files in the current directory, so the wildcard expression gets passed to find.
3. Option 3 is incorrect because it searches the **contents** of the files for lines which do not match 'oregano', rather than searching the file names.