Shell Tutorial

Janos Binder[1]

[1]European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany
janos.binder@embl.de

June 2, 2014

Contents

1 Getting started

Exam-like question Why would you use terminal and shell in the 21th century?

First you the necessary softwares to work remotely. On windows you need the necessary clients such as `Putty` and the `WinSCP` program. On Linux and Mac OSX you can use `ssh`:

```
> ssh server.name
```

Let's get familiar with the shell. You will see something like:

```
jbinder@red:~>
```

Where we are now? The `pwd` command will tell us. Try it out! Check also `man` or `info` out.

```
man pwd
```

Press `q` to exit from the manual.

2 Setting up the project environment

We organize our work into folders. How does it work from the console.

In the modern operating system every user has her or his own home directory. One can reach it by:

```
cd ~
```

The `cd` command changes the current directory and the `~` is a special character, which refers to the home directory. What is in this directory exactly? The following command will tell:

```
ls
```

`-la` flags are widely used to list the hidden files and the details of the files.

```
ls -la
```

3 Working with directories

The `.` denote the current directory and `..` denote the parent directory. The classroom projects should be stored in `projects`. It can be created by:

```
mkdir projects
cd projects
```

We start working in the projects directory.

One can always go back to the parent directory, when necessary:

```
cd ..
ls -la
```

Or combining more levels:

```
cd ../..
ls -la
```

Going back to projects:

```
cd ~/projects
ls -la
```

When some directories are no longer used,they need to be removed. The `rmdir` is the opposite command of `mkdir`.

```
mkdir wrong_directory
ls -la
ls -la wrong_directory
rmdir wrong_directory
ls -la
```

We use `ls -la` to check whether everything have gone fine.

Task: Create a project directory. It should be located under: `~/projects/compartments`

4 Manipulating files

First we need the human knowledge dataset from http://compartments.jensenlab.org/Downloads. This file contains human subcellular localization information from model organisms databases (mostly from *UniProtKB*). It can be be downloaded:

```
cd ~/projects/compartments
wget http://download.jensenlab.org/human_compartment_knowledge_full.tsv
ls
```

Take note that you can change multiple directories. You can use this *path* to copy, move or manipulate files (e.g. you can refer to the file as `~/projects/compartments/human_compartment_knowledge_full.tsv`.

You can also try:

```
curl http://download.jensenlab.org/human_compartment_knowledge_full.tsv > hkcf2.tsv
ls
```

Now we try out a few file manipulation commands. Copying (try out the `TAB` button when typing):

```
cp human_compartment_knowledge_full.tsv ../hckf.tsv
ls
```

Renameof the copied files (be lazy, try to use `TAB` button again):

```
mv ../hckf.tsv ../hc.tsv
ls
```

Where did that file go? What does that `..` for? What would `ls ..` do?

Task Find the `hc.tsv` file and move back to the current directory. Try to use the single `..`

Delete them:

```
rm hc.tsv
ls
```

The `-v` stands for verbose, and it is useful to see what is happening behind. It works with most of the unix commands. The `*` character matches to everything.

```
rm -v hc*.tsv
```

Task Create a directory under `~/projects/compartments/mouse`. Download the mouse knowledge dataset and rename the file to `mckf.tsv`.

Exam-like question You need to copy (or move) a file located in your home directory under `projects/compartments/protein_mappi` to the current directory. What command and how would you use?

4.1 Manipulating files

Bioinformatics data comes often in text format. To get a glance, try the following commands.

```
head human_compartment_knowledge_full.tsv
tail human_compartment_knowledge_full.tsv
```

The `head` command shows the beginning of the file, while `tail` shows the end of the file. Using `-20` will show the first or last 20 lines. The `human_compartment_knowledge_full.tsv` contains the following columns: ENSEMBL identifier of the gene, common protein name, cellular component GO identifier, name of the cellular component, source database, evidence code, confidence code.

The columns in `tsv` files are separated with a tab character (`\t`). These files are closely related to the comma separated files (`csv`).

Let see what is inside:

```
less human_compartment_knowledge_full.tsv
```

You can use the cursors, the `space` and `G` and `g` button, which goes to the end and the beginning of the file respectively. Use `q` to exit the program. We need some statistics about the file, the `wc` command answers how many lines, words and characters exist. We can limit it further by using the flags (`-l -w` or `c-`).

```
wc human_compartment_knowledge_full.tsv
wc -l human_compartment_knowledge_full.tsv
```

We can also filter specific lines, for example what proteins are localized in the mitochondria. The Gene Ontology identifier for mitochondrion is `GO:0005739`. It can be done by:

```
grep 'GO:0005739' human_compartment_knowledge_full.tsv
```

It seems to be too long, let us take a sample from the output. The pipes are useful in Unix for this purpose. For this we send the output of the command into another command.

```
grep 'GO:0005739' human_compartment_knowledge_full.tsv | head
```

If it seems good, we can store the mitochondrial proteins. For that purpose we will use redirection.

```
grep 'GO:0005739' human_compartment_knowledge_full.tsv > mitochondrial_proteins.tsv
```

Let us ignore annotations about NR3C1 protein. In the `grep` command using `-v` flag lists all lines, which do not contain the pattern. One example:

```
grep -v 'NR3C1' human_compartment_knowledge_full.tsv | head
```

Task Ignore annotations which comes from the *Reactome* database. Store the results in a file.

Specific columns can be also selected. Now we see how a file with pair of Ensembl identifiers and GO terms can be created. To get a feeling about the structure of the file again we use the `head` command and the `cut` command will select the respective columns.

```
head human_compartment_knowledge_full.tsv
cut -f1,3 human_compartment_knowledge_full.tsv | head
```

If works then let us store the results:

```
cut -f1,3 human_compartment_knowledge_full.tsv > ensembl_go_pairs.tsv
```

Where does the data come from? Which databases provided the information? Selecting the fifth column answers this question:

```
cut -f5 human_compartment_knowledge_full.tsv
```

However the results should be sorted and the duplicates should be removed. `sort` orders the databases in an alphabetical order, and the `-u` flag removes the duplicates.

```
cut -f5 human_compartment_knowledge_full.tsv | sort -u
```

We are also interested how many entries are coming from the various databases. Counting using `uniq -c` will answer the question.

```
cut -f5 human_compartment_knowledge_full.tsv | sort | uniq -c
```

Task List the various evidence codes and count how many proteins are associated with that evidence code.

Exam-like question We ran the following command `grep -v 'NR3C1' proteins_go.tsv | wc -l`

The contents of `proteins_go.tsv` is:

```
NR3C1   GO:0005622   Compara
NR3C1   GO:0005739   UniProtKB
TRIM23  GO:0044464   HGNC
RARS    GO:0017101   HPA
BNIP1   GO:0005737   UniProtKB
```

What will be the output? Why?

Exam-like question We ran the following command `cut -f1 'NR3C1' proteins_go.tsv | sort | uniq -c`

The contents of the file is the same as before.

What will be the output? Why?

4.2 Advanced operations with AWK

AWK is an interpreted programming language designed for text processing and typically used as a data extraction and reporting tool. It is a standard feature of most Unix-like operating systems. Let us see some examples.

Printing specific columns with AWK

As the first exercise let us see how can we print the columns with AWK. We use the GNU variant of AWK.

```
gawk -F '\t' '{print $1 "\t" $3;}' human_compartment_knowledge_full.tsv | head
```

The `-F '\t'` expresses the separator between the columns. In the curly backets we express what we want to do with the column. Using print statement, we can select the columns. It says print the first and the third columns separated by a tab.

Filtering on specific values

We can also select the mitochondrial proteins with AWK:

```
gawk -F '\t' '$3 == "GO:0005739"' human_compartment_knowledge_full.tsv | head
```

Here `$3 == "GO:0005739"` is a logical expression and it says execute only if the third column is equal with that value. The opposite expression would be `$3 != "GO:0005739"`. There are no curly backets used, and AWK by default will print out the lines, where the condition is true. We can combine it with the previous example.

```
gawk -F '\t' '($3 == "GO:0005739"){print $1 "\t" $3;}' human_compartment_knowledge_full.tsv | head
```

Exam-like question We ran the following command `gawk -F '\t' '($1 == "NR3C1"){print $1 "\t" $2;}' proteins_go.tsv`

The contents of `proteins_go.tsv` is:

```
NR3C1   GO:0005622   Compara
NR3C1   GO:0005739   UniProtKB
TRIM23  GO:0044464   HGNC
RARS    GO:0017101   HPA
BNIP1   GO:0005737   UniProtKB
```

What will be the output? If you do not have `gawk` available which commands would you use to reach the same goal from the following: `grep, wc, sort, cut, uniq`?

Exam-like question We ran the following command `gawk -F '\t' '($2 == "NR3C1"){print $1 "\t" $2;}' proteins_go.tsv`

What will be the output? If you do not have `gawk` available which commands would you use to reach the same goal from the following: `grep, wc, sort, cut, uniq`?

More filtering possibilities

It is often neccesary to filter a file based on some numerical threshold. In the example the last column denotes how reliable is the localization evidence. Filtering on high confidence scores ($= 5$):

```
gawk -F '\t' '$7 > 4' human_compartment_knowledge_full.tsv | head
```

There are a few logical expressions, which can be used with numbers. These are `==, <, >, <=, >=`.

One can combine multiple conditionals. Here we examine the mitochondrial proteins with highly reliable evidence.

```
gawk -F '\t' '$3 == "GO:0005739" && $7 > 4' human_compartment_knowledge_full.tsv | head
```

The `&&` denotes that both statement needs to be true, while `||` denote that at least one statement needs be true. Let us store these resuls.

```
gawk -F '\t' '$3 == "GO:0005739" && $7 > 4' human_compartment_knowledge_full.tsv > mitochondrial_proteins_
```

Task 1 Print mitochondrial proteins which comes from the *UniProtKB* database.

Task 2 Print proteins which do not come from *Reactome*, but it has atleast a confidence score of 4. The format should be ENSEMBL identifiers and GO identifier pairs (e.g. get rid of the unnecessary colums).

Exam-like question We ran the following command `gawk -F '\t' '($1 == "NR3C1" || $2 == "GO:0044464")' proteins_go.`

The contents of `proteins_go.tsv` is:

```
NR3C1   GO:0005622   Compara
NR3C1   GO:0005739   UniProtKB
TRIM23  GO:0044464   HGNC
RARS    GO:0017101   HPA
BNIP1   GO:0005737   UniProtKB
```

What will be the output?

5 Shell programming

We will write a few script during the second part. First we learn how editing works remotely. Before please create a directory named `~/projects/scripts`.

Coding the first script – Hello BioWorld!

```
pico 1_hello_bioworld.sh
```

Write the following lines:

```
#!/bin/sh

# we can also have comments in the examples

echo "Hello BioWorld!"
```

Exit works with `CTRL+X`. Don't forget to save!

The `#!/bin/sh` defines that the script is a shell script. The `echo` prints to the console.

You have make the script runnable and then you can test it. On Unix every file has user, group and everybody permission levels. When you create a file you as the user can read and write it, but you cannot execute it, that is the reason why we use `chmod` to change the permission. `ls -la` also lists the permissions.

```
ls -la 1_hello_bioworld.sh
chmod a+x 1_hello_bioworld.sh
./1_hello_bioworld.sh
ls -la 1_hello_bioworld.sh
```

Arithmetics with shell scripting

One can do arithmetic integer calculation with the shell scripts. The `$(( ... ))` denotes an arithmetic expansion. Try the following lines in the script:

```
#!/bin/sh

# Here are the most important arithmetic operations
```

```
echo "Two plus two is $(( 2 + 2 ))"
echo "Two minus two is $(( 2 - 2 ))"
echo "Two multiplied by two is $(( 2 * 2 ))"
echo "The four divided by two is $(( 4 / 2 ))"
echo "The remainder of four divided by two is $(( 4 % 2 ))"
```

Using variables

In computer programming, a variable or scalar is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents. Variables in programming may not directly correspond to the concept of variables in mathematics.

Variables are useful to store intermediate results such text and numbers. Try the following example in the `2_hello_variable.sh` script.

```
#!/bin/bash

HW="Hello World!"

echo $HW
echo "It is boring to print" $HW

# Here we show an arithmetic variable.

FOUR=2+2

# Try out what happens if you leave the brackets and the expression becomes £FOUR.

echo "Two times two is $(( FOUR ))"
```

When we refer to the variables we need to use the $ sign. Arithmetic variables are not evaluated if they are not between brackets.

Using conditionals

In computer science, conditional statements, conditional expressions and conditional constructs are features of a programming language which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false. Apart from the case of branch predication, this is always achieved by selectively altering the control flow based on some condition.

A simple example of `3_basic_conditionals.sh`:

```
#!/bin/bash

FILE="3_basic_conditionals.sh"

# -f checks whether the file exists.

if [ -f $FILE ]; then
    echo "FILE:" $FILE "exists!"
fi
```

A more complicated example of `3a_complex_conditionals.sh`:

```
#!/bin/bash

FILE="3a_complex_conditionals"

if [ -f $FILE ]; then
    echo "FILE:" $FILE "exists!"
else
    echo "FILE:" $FILE "does not exists!"
fi
```

Task: Try to modify this script to run the other branch of the conditional.

A similar solution with integer in `3b_arithmetic_example.sh`. Take note of == operator. It is the same as the one in `gawk`.

```
#!/bin/bash

FOUR=2+2

echo "Is two plus to really four?"
if [[ $((FOUR)) == 4 ]] ; then
    echo "Yes!"
else
    echo "No!"
fi
```

Using loops

A loop is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met, or indefinitely.

One uses loops when a step or similar steps needs to be carried out multiple times.

Let us create the example `4_loop_basics.sh`.

```
#!/bin/sh

echo "We count to ten!"

for (( i = 1 ; i <= 10 ; i++ )) ; do
    echo $i
done
```

The double bracket denote that `i` refers to a number.

Task Modify the script to count back from 10.

Task Write "Hi five!" as well when `i` become 5.

There is another solution to the previous example with the `while` keyword (`4a_loop_until.sh`).

```
#!/bin/sh

echo "We count to ten!"

(( i = 1 ))
while (( i <= 10 )) ; do
    echo $i
```

```
    (( i++ ))
done
```

Task: Modify the script to count back from 10.

Task Write "Hi five!" as well when `i` become 5.

The shell can also store the output of a command. Here is an example, which writes the files out (`4b_loop_files.sh`):

```
#!/bin/sh

FILES=`ls -1`

for f in $FILES; do
    echo $f "is a file"
done
```

The `` `...` `` triggers command substitution; equivalent to `$(...)`, but is somewhat more error-prone.

Exam-like question Why would you use variables/conditionals/loops in a programming language?

Task Make a script which prints out the following pattern. Use `echo -n` to print in the same line:

```
* * * * *
* * * *
* * *
* *
*
```

Task Write a script which prints out how many lines every file has in the directory. You can invoke the `wc` command from the body of the script.