Unix/Linux Tutorial for Beginners Session VIII

Mihaela Martis

NBIS & Faculty of Medicine and Health Sciences
Division Cell Biology, IKE

Daily work

- bioinformatics work often involves a great deal of data processing

Daily work

- bioinformatics work often involves a great deal of data processing
- run regularly a sequence of commands on multiple files

Daily work

- bioinformatics work often involves a great deal of data processing
- run regularly a sequence of commands on multiple files
- summarize various processing steps into a pipeline

Daily work

- bioinformatics work often involves a great deal of data processing
- run regularly a sequence of commands on multiple files
- summarize various processing steps into a pipeline
- robust and reproducible pipelines
  - errors are easily introduced in the complex processing of bioinformatics data
  - automated pipelines provide a perfect record of exactly how data was processed

Bash scripting

- the shell can be used interactively, but it's also a full-fledged scripting language
- used to tape many commands together into a cohesive workflow

Bash scripting

- the shell can be used interactively, but it's also a full-fledged scripting language
- used to tape many commands together into a cohesive workflow
- useful for simple tasks → does **not replace** the knowledge of other programing languages

Bash scripting

- the shell can be used interactively, but it's also a full-fledged scripting language
- used to tape many commands together into a cohesive workflow
- useful for simple tasks $\rightarrow$ does **not replace** the knowledge of other programing languages
- it lacks better numeric type support, useful data structures, better string processing, powerful functions ...

Bash scripting

- the shell can be used interactively, but it's also a full-fledged scripting language
- used to tape many commands together into a cohesive workflow
- useful for simple tasks $\rightarrow$ does **not replace** the knowledge of other programing languages
- it lacks better numeric type support, useful data structures, better string processing, powerful functions ...

  $\rightarrow$ often the **best** and **quickest** 'duct tape' solution

The shell script

- is a text file that contains a sequence of shell commands and which can be invoked as a program
- can be created in your favorit text editor
- by convention, it has the extension .sh
- consists of 2 parts: the shell header (shebang) and the body (commands)

The shell script header

```
#!/bin/bash
set -e
set -u
```

- #!/bin/bash → **shebang** – indicates the path to the interpreter used to execute the script
- set -e → terminates the entire script if any command exits with a nonzero exit status
- set -u → avoids running a script, if a variable's value is unset

The shell script body

- describes '*what to do and how to do it*'
- defines variables
- lists the commands, which should be executed by the shell
- the shell processes the body sequentially

```
path="/home/duck/data/fasta/"

echo "List the content of the folder $path"
echo "Content:"
ls -l $path
```

How to run a shell script

1. give the script execute permissions

```
$ chmod u+x myFirstScript.sh
$ chmod 755 myFirstScript.sh
```

How to run a shell script

1. give the script execute permissions

```
$ chmod u+x myFirstScript.sh
$ chmod 755 myFirstScript.sh
```

2. execute your script

```
$ bash myFirstScript.sh
```

or

```
$ sh myFirstScript.sh
```

or

```
$ ./myFirstScript.sh
```

Example: myFirstScript.sh

```
#!/bin/bash
set -e
set -u

path="/home/mihaela/data/fasta/"

echo "List the content of the folder $path"
echo "Content:"
ls -l $path
```

```
$ ./myFirstScript.sh
List the content of the folder /home/mihaela/data/fasta/
Content:
total 118228
-rwxr-xr-x 1 mihaela root 17808819 Mar 22 13:10 ZMpep.bz2
-rwxr-xr-x 1 mihaela root 32247082 Mar 22 13:10 barley_CDS.fa
-rwxr-xr-x 1 mihaela root 42843621 Mar 22 13:10 brachy_CDS.fa
drwxr-xr-x 2 mihaela root       78 Mar 22 13:10 subset
-rwxr-xr-x 1 mihaela root 28162050 Mar 22 13:10 wheat_PEP.fa
```

Write your first shell script

- go to folder *scripts* in your home directory

```
$ cd ~/myLinuxProject/scripts
```

Write your first shell script

- go to folder *scripts* in your home directory

```
$ cd ~/myLinuxProject/scripts
```

- open a text editor

```
$ nano
or
$ vi
```

Write your first shell script

- go to folder *scripts* in your home directory

```
$ cd ~/myLinuxProject/scripts
```

- open a text editor

```
$ nano
or
$ vi
```

- type the shebang

```
#!/bin/bash
set -e
set -u
```

Write your first shell script

- go to folder *scripts* in your home directory

```
$ cd ~/myLinuxProject/scripts
```

- open a text editor

```
$ nano
or
$ vi
```

- type the shebang

```
#!/bin/bash
set -e
set -u
```

- write the command

```
echo "This is my first shell script"
```

Write your first shell script (II)

- save the file under the name *myFirstShellScript.sh*

Write your first shell script (II)

- save the file under the name *myFirstShellScript.sh*
- set the permissions

```
$ chmod u+x myFirstShellScript.sh
```

Write your first shell script (II)

- save the file under the name *myFirstShellScript.sh*
- set the permissions

```
$ chmod u+x myFirstShellScript.sh
```

- execute the script

```
$ ./myFirstShellScript.sh
This is my first shell script
```

Write your first shell script (II)

- save the file under the name *myFirstShellScript.sh*
- set the permissions

```
$ chmod u+x myFirstShellScript.sh
```

- execute the script

```
$ ./myFirstShellScript.sh
This is my first shell script
```

$\rightarrow$ CONGRATULATIONS TO YOUR FIRST SHELL SCRIPT!

Variables

- allow to store information and do something with it
- you can store input files, parameter values for commands, results directories ...
- syntax: variable_name=value
- 2 types:

Variables

- allow to store information and do something with it
- you can store input files, parameter values for commands, results directories ...
- syntax: variable_name=value
- 2 types:
  - system variables → created and maintained by the operating system itself
    - defined in *CAPITAL LETTERS*
    - e.g. *BASH=/bin/bash, HOME=/home/duck*

Variables

- allow to store information and do something with it
- you can store input files, parameter values for commands, results directories ...
- syntax: variable_name=value
- 2 types:
  - system variables → created and maintained by the operating system itself
    - defined in *CAPITAL LETTERS*
    - e.g. *BASH=/bin/bash, HOME=/home/duck*
  - user defined variables (UDV) → created and maintained by the user
    → defined in lowercase letters

Rules

- variable name must begin with alphanumeric character or underscore character (_)
- don't put spaces on either side of the equal sign when assigning a value to variable
- variables are case-sensitive
- don't use ? or * to name your variables
- add a '$' in front of a variable name to access its value

```
#!/bin/bash
set -e
set -u

results=/home/duck/results
mkdir -p $results
```

Command substitution

- use the command substitution $() to run a shell command and store the output to a variable
- usage: var=$(command)

```
path=$HOME/data/fasta/mySeq.fa
command_out=$(basename $path)
echo $command_out
```

- $HOME is an alias for ∼ (tilde) and stores the path to the user's home

Quoting

- Bash splits up the input in words using the whitespace between them to determine where each argument starts and ends
- quotes keep your strings in one piece → pass the whole string as one argument
- double quotes (") vs single quotes (')

Quoting

- Bash splits up the input in words using the whitespace between them to determine where each argument starts and ends
- quotes keep your strings in one piece → pass the whole string as one argument
- double quotes (") vs single quotes (')
    - enclosing simple text → no difference which you use

Quoting

- Bash splits up the input in words using the whitespace between them to determine where each argument starts and ends
- quotes keep your strings in one piece → pass the whole string as one argument
- double quotes (") vs single quotes (')
  - enclosing simple text → no difference which you use
  - shell variable expansion → double quotes (") allow expansion of variables, single quotes (') don't

```
test="Hello World"
echo $test
   Hello World
echo "$test"
   Hello World
echo '$test'
   $test
```

Comments

- comments make your life easier → reflect what the script does and which data was used
- comments start with a hash mark '#'
- the shell ignores lines starting with a # and they are only visible upon opening the file

```
#!/bin/bash
set -u
set -e
# This scripts clears the terminal and displays a greeting

clear                    # clear terminal window
echo "Hello world!"
```

Command-line arguments

- shell script can take arguments from the command-line
- those are assigned to the values $1, $2 etc
- $0 stores the name of the script
- $# contains the number of arguments

Command-line arguments

- shell script can take arguments from the command-line
- those are assigned to the values $1, $2 etc
- $0 stores the name of the script
- $# contains the number of arguments

```
#!/bin/bash
set -e
set -u

echo "number of arguments: $#"
echo "script name is: $0"
echo "first argument is: $1"
echo "second argument is: $2"
```

Command-line arguments

- shell script can take arguments from the command-line
- those are assigned to the values $1, $2 etc
- $0 stores the name of the script
- $# contains the number of arguments

```
#!/bin/bash
set -e
set -u

echo "number of arguments: $#"
echo "script name is: $0"
echo "first argument is: $1"
echo "second argument is: $2"
```

```
$ ./myFirstScript.sh Hello world
number of arguments: 2
script name is: myFirstScript.sh
first argument is: Hello
second argument is: world
```

Shell arithmetic

- arithmetic expansion and evaluation is done by placing an integer expression using the following format:

```
    $(( expression ))
    $(( n1 + n2 ))
    $(( n1 / n2 ))
    $(( n1 - n2 ))
    $(( n1 * n2 ))
```

- examples:

```
#!/bin/bash

x=5
y=10
ans=$(( x + y ))
echo "$x + $y = $ans"

echo $(( 10 + 5 ))
```

Shell arithmetic II

- bash arithmetic works only with integer

```
#!/bin/bash
set -e
set -u

x=5.5
y=10

res=$(( x + y ))
echo "$x + $y = $res"
```

```
$ ./myMath.sh
./myMath.sh: line 5: 5.5: syntax error: invalid arithmetic operator (error
      token is ".5")
./myMath.sh: line 8: res: unbound variable
```

Interactive shell scripts

- scripts can ask questions, and get and use responses
- read → takes input from the keyboard and assigns it to a variable

```
#!/bin/bash
set -u
set -e

echo -n "Enter your name > "
read name
echo "You entered: $name"
```

echo -n keeps the cursor on the same line

```
./read_demo.sh
Enter your name > Kurt
You entered: Kurt
```

read options

- -t followed by a number – provides an automatic timeout for the read command (in seconds)

```
#!/bin/bash
set -u
set -e

echo -n "Hurry up and type something! >"
if read -t 3 response; then
  echo "Great, you made it in time!"
else
  echo "Sorry, you are too slow!"
fi
```

- -s – causes the user's typing not to be displayed

Pair exercises

- find a partner for the next exercise session
- aim: learn to transfer taught commands to basic shell scripts
- we will use some of the exercises from session 5
- the tasks can be found on the e-learning platform under session 8