Introductory Linux Tutorial for Life Sciences

Session 9: Introduction to shell scripting 1

Basics

In this session

- Motivation for shell scripting.
- What a script is and how to create one.
- Basic shell scripts.
  - Variables (creating and using them).
  - Command substitution (i.e. getting output from a command).
  - Quoting (it matters).
  - Command line arguments (how to inspect the arguments given to a script).
  - Arithmetics.
  - Reading strings into variables (from a user or file).

Daily work at the Unix command prompt

- Bioinformatics work often involves a great deal of data processing tasks.
- Data processing may include running a number of sequential command to, for example,
  - extract data from files based on various conditions, or
  - combine data in various ways by merging or joining files, or
  - reformat or otherwise modify the contents of files.
- These processing steps are often the same for a set of files, and may often have to be re-run several times within the course of a single project (for whatever reasons).

Daily work at the Unix command prompt

- Typing commands at the command prompt is error prone and tedious.
- Processing pipelines can be formulated to take inputs and to produce outputs.
- Workflows can often be formalised, generalised, and arranged into robust and reproducible pipelines.

Daily work at the Unix command prompt

- On a Unix system, a shell script can be seen as a way of “recording the steps for doing a set of operations/running a pipeline, and later replay them.”
- Benefits:
  - Encapsulates the operation needed to perform a task.
  - Provides a record of what was actually done.
  - Easy to modify to cope with new circumstances.
- A shell script becomes “a new command” that you may use to do something specific.

Shell scripting

- The command prompt is actually provided by a command called “the shell”.
- Unix systems often provide several different shells, each with its own specializations and extensions.
- The most basic shell is the “sh” shell.
  - Its syntax and behaviour is specified by “the POSIX standard”.
  - Other shells are often compatible with it, but may provide extensions to it.
- The most common shell on Linux systems is bash.
- Alternative shells include zsh, ksh and others.

Shell scripting

- The shell can be told to run a shell script.
- A shell script is a set of commands.
- Useful for simple tasks or for gluing other tasks together.
- Does not replace knowledge of other programming languages.
- Lacks support for
  - floating point arithmetics,
  - most common data structures, and
  - fine-grained manipulation of strings etc.

The shell script

- A shell script is a plain text file that is read and executed by the shell.
- Anything that you can do at the command line can be done in a shell script, and it will behave the same (given the same environment).
- A script is created or modified with any plain text editor
  - nano
  - emacs
  - vim
- Scripts can have any filename, but are often given short names and a “.sh” filename suffix.

A “Hello World” script

```
#!/bin/bash

echo "Hello World"
```

- The first line always starts with “#!” followed by the pathname of the shell executable to use for running the script.
- In this case, the script is executed using /bin/bash (the bash shell).
- The rest of the file consists of commands that the shell should run.
- This script will just print out the string “Hello World” in the terminal using “echo”.
- When a script is done, control is given back to the user and a new prompt is shown.

Running our “Hello World” script

```
$ chmod +x hello-world.sh
```

```
$ ./hello-world.sh
Hello World
```

```
$ bash ./hello-world.sh
Hello World
```

- The “chmod +x” command makes the script (which is a text file) executable.
  - You only have to do this once for a script.
- This means that if you try to run the script, the shell will
  - Investigate the “#!”-line.
  - Call the command mentioned there.
  - Pass it the script for running.
- A script can be run using “an explicit interpreter” (as show last to the left).
  - This requires you to know that the script is actually a bash script, and not a Python or zsh script (which would cause issues).

Writing a script

- Go to where you want to create the script.
  - `cd ~/myLinuxProject/scripts`
- Open a text editor.
  - `emacs my-script.sh`
  - `nano my-script.sh`
  - `vim my-script.sh`
- Write the script.
  - ```
    #!/bin/bash
    echo "This script was written by me"
    ```
- Save the script.
- Make the script executable.
  - `chmod +x my-script.sh`
- Run the script.
  - `./my-script.sh`
- See the result
  - `This script was written by me`
    ```

Shell scripting: Variables

- Shell variables allow for storing information during the duration of the shell script.
- The data stored in a shell variable is a string.
- A string can represent a number, a set of words, a pathname, a timestamp, or any other text.
- Some shells provides array variables (bash does, for example).
- All created variables disappear when the script terminates.

Shell scripting: Variables, syntax

- Variables are created by assigning a value to them:
  - `variable_name="value"`
- No space is allowed around the “=” in an assignment.
- Variable names may start with a letter or an underscore, and the name may otherwise consist of letters, digits, and underscores.
- Variable names are case sensitive.
- A user’s shell variables should not be all upper-case.
  - Upper-case variable names are by convention reserved for environment (“system”) variables.

Shell scripting: Variables, expansion

- To get a variable's value, prefix the variable's name with "$":
  - `echo "$my_variable"`
  - `subject="Hello dear $USER"`
- Getting a variable's value is called "expanding the variable".
- In the second example above, the variable "USER" is expanded as part of constructing a new value for the variable "subject".
- (The "USER" variable is usually set by the shell to the username of the current user.)

A modified “Hello World” script

```
#!/bin/bash

# This script prints a message
# and lists the files visible in
# the current directory.

message="Hello World"
echo "$message"

echo "These are your files:"
ls -l
```

- The script says when to do what and how.
- The script may contain variables (you could use variables on the command line too, obviously).
- Commands are executed sequentially.
- Execution is not halted if there are errors.
    - You may want to use “set -e” and “set -u” to have your scripts terminate immediately upon error, or when an undefined variable is accessed.
    - `#!/bin/bash -eu`

Variables: Trimming values

${variable#pattern}	remove shortest prefix matching “pattern”
${variable##pattern}	remove longest prefix matching “pattern”
${variable%pattern}	remove shortest suffix matching “pattern”
${variable%%pattern}	remove longest suffix matching “pattern”

- This shows how, given some variable, one may trim off a prefix or suffix from the value of that variable using a globbing pattern.
- Example:
  - name="01-file-A.txt"
  - echo "${name#01-}" # prints "file-A.txt"
    echo "${name#*-}" # prints "file-A.txt"
    echo "${name##*-}" # prints "A.txt"
  - echo "$name%.txt}" # prints "01-file-A"
    echo "${name%-*}" # prints "01-file"
    echo "${name%%-*}" # prints "01"

Shell scripting: Command substitution

- A “command substitution” can be used to insert the output of a command in a string.
- The syntax is “$(...)”.
    - ```
      script_path="$HOME/myLinuxProject/my-script.sh"
      script_name="$(basename "$script_path")"
      echo "The name of the script is $script_name"
      ```
- $HOME will always expand to your home directory.
- The “~” (tilde) character is a shorter way of writing $HOME, but it’s most commonly used on the command line, not in scripts, and does not behave like a variable, e.g. does not expand within quotes.
      ```

Shell scripting: Quoting, word-splitting + globbing

- An unquoted expansion (variable expansion, arithmetic expansion, or command substitution etc.) will undergo
  - word-splitting on whitespace characters (spaces, tabs, and newlines by default), which generates separate words from the variable's value, and
  - each generated word will undergo filename generation ("globbing"), which may expand the list with matched filenames.
- Example:
  - ```
    var="hello * world"
    echo $var
    ```
  - ```
    hello myfile-1 myfile-2 myfile-3 myfile-4 myfile-5 world
    ```

Shell scripting: Quoting, avoiding split + glob

- Double quoting the expansion inhibits word-splitting and filename-generation.
- Example
  - ```
    var="hello * world"
    echo "$var"
    ```
  - ```
    hello * world
    ```
- An expansion is quoted as soon as it occurs somewhere in a quoted string. The quotation marks do not have to be tightly around the expansion.
  - ```
    echo "The variable's value is $var"   # $var is quoted there
    ```
    ```

Shell scripting: Quoting, always double quote

- **Always double quote variable expansions and command substitutions.**
  - There are instances where quoting is not needed, but it's much easier to remember to always quote expansions.
  - It's very rare to want to avoid double quoting expansions, and it's often an error or a source of future errors to not use double quotes around expansion.
  - Common issue: It is very common to forget that filenames may contain spaces (or "*", "?", etc.)
    - `mv "$file" "$destdir"     # double quote to avoid issues`
- Single quotes inhibit all expansions.
  - `echo '$USER $HOME'`
  - Will print the literal string $USER $HOME

Shell scripting: Comments

```
#!/bin/bash -eu

# Lists all visible names in the current
# directory by means of a globbing
# pattern and echo.

echo ./*    # do the thing
```

- Comments are introduced with a hash character (“#”).
- The line from the “#” to the end of the line will be ignored. The “do the thing” text is also a comment in this example.
- Comments are often used to describe what the script does.
- This example also shows the use of a filename globbing pattern. These are often used in loops (more on this later).

Shell scripting: Command line arguments

- Shell script may take arguments on the command line.
  - `./my-script.sh myfile.fa "hello world"`
- Within the script, these are available as the “positional parameters”. These are accessed as “$1”, “$2”, “$3”, etc. (“$0” usually contains the name of the script itself).
- The number of such parameters are recorded in the special variable “$#”.
- With the example command above:
  - “$1” will be the string “myfile.fa”, “$2” will be the string “hello world” (the quotes on the command line are not included in the variable’s value), and “$#” would be 2.

Shell scripting: Command line arguments

```
#!/bin/bash -eu

# Shows the number of command line
# argument, and will also output
# the first two arguments.

echo "Got $# arguments"
echo "Argument 1: $1"
echo "Argument 2: $2"
```

```
$ ./my-script.sh myfile.fa "1 2 3"
Got 2 arguments
Argument 1: myfile.fa
Argument 2: 1 2 3
```

- What would this script print if “1 2 3” wasn’t quoted?
- What would it print if it was only given a single (or no) argument?

Shell scripting: Arithmetics

- The shell can do simple integer arithmetics.
- Calculations are done within an “arithmetic expansion”: “$((...))”.
- The expression inside “$((...))” may use the ordinary arithmetic operators, and may contain variables. Variables do not need to be prefixed by “$”.
- ```
  a=17
  b=3
  echo "$a + $b = $(( a + b ))"   # prints "17 + 3 = 20"
  echo "$a / $b = $(( a / b ))"   # prints "17 / 3 = 5"
  c="$(( a * b ))"                # c gets the value "51"
  ```
  ```

Shell scripting: Interacting with a user

- A shell script can use the command “read” to read data from a user (or from a file, or from another command; we will see example of this later).
- The “read” command reads data from a single line and assigns the words in that line to one or several variables.
- “read” can be made to prompt the user with a custom string using its “-p” option, and can also be made to hide whatever the user is typing (for e.g. passwords) with its “-s” option. A timeout can be set with “-t n” to “n” seconds. (etc.)
- Don’t read pathnames with “read”, it’s easier for the user to just type them on the command line as arguments to the script (with e.g. tab-completion).

Shell scripting: Interacting with a user

```
#!/bin/bash -eu

read -p 'Name and age: ' name age
read -s -p 'Secret: ' secret

echo "Hello $name"
echo "You were born ca. $((2019-age))."
echo "Your secret is safe with me."
```

- This script asks two questions and expects two lines of input from the user.
- The first line is expected to contain at least two words.
    - The first word will be read into “name” and the second (and any other words) will be read into “age”.
- No text will show when the “secret word” is typed in by the user.

Exercises

- If appropriate, find a partner to work with.
- The aim of the exercises is to transfer some of the shown commands and concepts into simple shell scripts.
- The exercises may be found in the E-learning platform page for this course as “Exercises session 9”.