Introductory Linux Tutorial for Life Sciences

Session 10: Introduction to shell scripting 2

More advanced topics

In this session

- Shell script control structures.
  - “if”-statements.
  - “while”-loops.
  - “for”-loops (of various kinds).
- Extra material (if we have time)
  - Arrays in bash.
  - Tests on variables using globbing patterns and regular expressions in bash.
  - Where to go for help from a human.
- Not covered
  - Shell functions.

Shell script control structures

- Shell scripts allow the use of control structures familiar to anyone that knows other programming languages.
- These include
  - “if”-statements and logical operations,
  - “while”-loops, and
  - “for”-loops.

“if”-statements

```
if <test-command>; then
    <something>
elif <test-command>; then
    <something>
else
    <something>
fi
```

```
if <test-command>
then
    <something>
fi
```

- The “if”-statement conditionally executes a set of commands based on the successful execution of a test-command.
- The most common command used for performing tests is “[“ (yes, that’s a left square bracket).
- ```
  if [ "$reply" = "answer" ]; then
      echo "Yes, correct"
  else
      echo "No, wrong"
  fi
  ```
  ```

“if”-statements: The “test” and “[“ commands

- The two commands “test” and “[“ are the same but “[“ requires matching “]” as last argument:
  - `if [ "$reply" = "answer" ]; then`
  - `if test "$reply" = "answer"; then`
- This command supports various tests on strings, integers, and also on files.
- The “test” and “[“ commands do not output anything, but return an “exit status” to the shell signalling success or failure.
- All commands return an exit status.
  - “if” can be used with any command.
- String tests:
  - `[ -n "$string" ]`
    test for non-empty string
  - `[ -z "$string" ]`
    test for empty string
  - `[ "$string" = "$other" ]`
    test for string equivalence
  - `[ "$string" != "$other" ]`
    test for string non-equivalence

“if”-statements: The “test” and “[“ commands

- Integer tests:
  - `[ "$number" -eq 10 ]`
    integer equivalence
  - `[ "$number" -ne 10 ]`
    integer non-equivalence
  - Also, “-lt”, “-gt”, “-le”, “-ge” for integers.
  - Note: “<” and “>” are string tests on lexicographical sort order (seldom used).

- File tests:
  - `[ -e "$name" ]` $name exists
  - `[ -f "$name" ]` is a regular file
  - `[ -d "$name" ]` is a directory
  - `[ -L "$name" ]` is a symbolic link
- Note the space after “[“ and between arguments, and before final “]”.
- The test operator (“=”, “-ne” etc.) is always a separate argument, so leave a space around it on either side.

“if”-statements: Example 1

```
#!/bin/bash -eu

name="$1"

if ! [ -e "$name" ]; then
    echo "does not exist"
elif [ -f "$name" ]; then
    echo "it is a regular file"
else
    echo "it is not a regular file"
fi
```

- This script is invoked with a filename as its only argument.
  - `./test-script.sh some-filename`
- The given filename is stored in the variable “name”.
- The script then figures out whether the filename exists at all, and if it does, whether it’s a regular file or not.
- Note the use of “!” to negate the result of a test.
  - `[ ! -e “$name” ]`
  - `! [ -e “$name” ]`

“if”-statements: Example 2 (advanced)

```
#!/bin/bash -eu

lockdir="$HOME/lockdir"

if ! mkdir "$lockdir"; do
    echo "Can't lock, exiting..."
    exit 1
done

# Do some work here, and then...

rmdir "$lockdir"
```

- This script implements a simple “lock”.
- The test ensures that the rest of the script does not run unless a directory called “lockdir” in the home directory can be created.
- If the script manages to create the directory, the rest of the script runs (otherwise it exits).
- The directory is deleted at the end.
- This prevents multiple copies of the script from running at once.

“if”-statements: Logical operators

- Several test-commands may be strung together with boolean operators:
  - `&&` “and”
  - `||` “or”

```
if [ -z "$string" ] ||
   [ "$string" = "empty" ]
then
    echo "the string is empty"
    echo "or it is the word 'empty'"
fi

if [ -d "$name" ] && [ -L "$name" ]; then
    echo "$name is a symlink to a directory"
fi
```

- These are “short-circuiting” operators, meaning that
  - if the left side of “&&” is true, then the right hand side will be executed, and
  - if the left hand side of “||” is true, then the right hand side will not be executed.
- These three are equivalent:
  - `[ -n "$str" ] && echo "$str"`
  - ```
    if [ -n "$str" ]; then
        echo "$str"
    fi
    ```
  - `[ -z "$str" ] || echo "$str"`
    ```

“while”-loops

```
while <test-command>; do
    <something>
done
```

```
until <test-command>; do
    <something>
done
```

- The “while”-loop performs an action for as long as a particular test is true.
- The test-command is executed once at the start of each iteration, and if it runs successfully, the code in the body of the loop is executed.
- “while”-loops are used when the loop has to run an indeterminate number of times, or if the test-command for some other reason needs to be run for each iteration.
- The “until”-loop is the same as the “while”-loop with a negated test, and is not terribly often used.

“while”-loops: Example 1

```
#!/bin/bash -eu

number=10

while [ "$number" -ne 0 ]; do
    echo "Now at $number"
    number=$(( number - 1 ))
done
```

- The scripts sets the “number” variable to 10.
- For as long as the variable’s value is not zero, the “while”-loop prints out a short message and then decrements the value in the variable by one.
- The script will print “Now at N” for all integers N from 10 down to 1.

“while”-loops: Example 2

```
#!/bin/bash -eu

count=0

while read words; do
    echo "Got the string '$words'"
    count=$(( count + 1 ))
done

echo "Read $count times"
```

- This is an example of using “read” as the test-command.
- The “read” command reads from “standard input”.
- The “read” command will fail if there is nothing more to read (or when timing out when you use “-t” to set a timeout).
- This script reads words from standard input until there is nothing more to read, and counts the number of iterations performed.
- To signal “end of input” if you run this as-is, press “Ctrl+D”.

“while”-loops: Example 2 (variation, reading from a file)

```
#!/bin/bash -eu

infile="sample.txt"
count=0

while read words; do
    echo "Got the string '$words'"
    count=$(( count + 1 ))
done <"$infile"

echo "Read $count times"
```

- This variation of the script on the last slide reads from a file called “sample.txt” (which is assumed to exist in the current directory, otherwise the full path to the file would need to be specified).
- The script on the last slide could have been invoked as follows to get exactly the same effect (assuming both the script and the input file exists in the current directory):
    - `./read-script.sh <sample.txt`

“while”-loops: Example 3 (input validation)

```
#!/bin/bash -eu

while true; do
    read -p 'Filename: ' name
    if [ -e "$name" ]; then
        break
    fi
    echo "Try again..."
done

echo "'$name' is a valid filename"
```

- This script implements what’s commonly known as an “input validation” loop.
- “true” is a command that always runs successfully (there is also “false”).
  So, the loop seems to be infinite.
- The “break” command breaks out of the innermost loop (there is also “continue”, which immediately continues with the next iteration).
- After the loop, “$name” is validated to be the name of an existing file or directory.
- To exit a script prematurely, press “Ctrl+C”.

“for”-loops

```
for <variable> in <list>; do
    <something>
done
```

- “for”-loops iterates over a static list of words.
- In each iteration, the given variable will have the next word as its value.
- “for”-loops are often used for iterating over a set of pathnames of files, applying a set of commands to each of them in turn.

“Fun fact”: In almost any command, as in the loop above, the semicolon (“;”) may be replaced by a newline (or vice versa, both “;” and newline serves as “command terminators”). The loop above could therefore have the “do” keyword on the second line without using “;” at all, or be written on a single line by replacing each newline with “;” (except there can’t be a “;” directly after “do” since it’s a keyword, not a command; there are always exceptions to short and snappy rules...)

“for”-loops: Example 1 (looping over a globbing pattern)

```
#!/bin/bash -eu

datadir="./data"

for pathname in "$datadir"/*.fa; do
    wc -l "$pathname"
done
```

“Fun fact”: A “pathname” is the path to a file or directory. A “filename” is the last path component of a pathname.

- This script runs “wc -l” on each file under “./data” whose filename ends in “.fa”.
- The “wc -l” command counts the number of lines in a file.
- The given filename globbing pattern is expanded before the first iteration of the loop starts, so the loop runs over a static list of pathnames.

“for”-loops: Example 2 (using brace expansions)

```
#!/bin/bash -eu

for number in {1..50}; do
    if [ "$(( number % 2 ))" -eq 0 ]
    then
        echo "Even: $number"
    else
        echo "Odd:  $number"
    fi
done
```

- This scripts prints out all numbers between 1 and 50 and also whether they are odd or even.
- The loop uses a brace expansion that will expand to all numbers in the range 1 to 50 before the loop starts running.
  - `{a..z}`
  - `xxx{ab,cd,ef}yyy`
- The test for even numbers is done using an arithmetic substitution which uses the modulus operator “%”.

“for”-loops: Example 3 (the arithmetic “for”-loop)

```
#!/bin/bash -eu

for (( number=1; number<=50; ++number ))
do
    if [ "$(( number % 2 ))" -eq 0 ]
    then
        echo "Even: $number"
    else
        echo "Odd:  $number"
    fi
done
```

- This is the same example as on the last slide, but it uses an “arithmetic loop” which is available in the bash shell.
- This type of loop works in the same way as similar loop constructions in other languages.
- The loop header consists of three parts delimited by semicolons within “((...))”.
    - Initialization
    - Conditional test
    - Increment

“for”-loops: Example 4 (command line argument)

```
#!/bin/bash -eu

count=0

echo "Got $# command line arguments"

for arg do
    count=$(( count + 1 ))
    echo "Argument $count is '$arg'"
done
```

- This shows a special form of the “for”-loop that seems to not iterate over anything.
- When this form is used, the loop will iterate over the command line arguments given to the script.
- `./arg-script.sh 1 2 3 "a b c"`
  - Would say it got 4 arguments, and would then list these.
- `./arg-script.sh *`
  - Would take all visible names in the current directory as arguments.

Extra material: Arrays

- Never store multiple separate strings in a single ordinary variable.
- The bash shell allows for the use of arrays (as an extension to the POSIX standard).
- Arrays are lists of distinct strings.
- Useful for storing a list of filenames, or any other collection of strings.
- Arrays are created by assigning to a variable using “variable=(...)”.

```
namelist=( "One Name" "Another Name" )

filelist=( "$HOME"/data/*.fa )
```

Extra material: Arrays

- Arrays in bash are indexed from 0.
- To access an element of an array, use the syntax “${variable[index]}” where “index” is some positive integer, or a variable holding a positive integer.
- The number of items in an array may be had with “${#variable[@]}”.
    - The length of the string in an ordinary variable, by the way, is “${#variable}”.
- Use “${variable[@]}” (always double quoted, since it is an expansion) in “for”-loops to loop over all elements of the array “variable”.

Extra material: Arrays

```
names=( "John Cook" "Mary Smith" )

for name in "${names[@]}"; do
    echo "Name: $name"
done
```

```
pathnames=( ./data/*.fa )

echo "There are ${#pathnames[@]} .fa files"
echo "They are:"

for pathname in "${pathnames[@]}"; do
    echo "$pathname"
done
```

- The first loop iterates over the two names in the array, printing each.
- If the double quotes around “${names[@]}” were removed, each name would be split on whitespaces, so the loop would iterate four times.
- The second loop expands a filename globbing pattern to collect the pathnames of a set of “.fa” files.
- It then prints the number of pathname found, and each individual pathname.
- Remember that filenames may contain both spaces and newlines on Unix systems.

Extra material: Arrays

- Arrays could also be used to hold command line options that you later use when calling some other program from a script.
- The following additionally shows how to append to an array.

```
options=( --optimize=heavy --parallel=4 )
if [ -n "$mode" ]; then
    options+=( --analyze-mode="$mode" )
fi

for input_name in "$HOME"/data/*.in; do
    analyze-everything "${options[@]}" --input="$input_name"
done
```

Extra material: Pattern matching

- Two main types of patterns are used in shell scripting:
  - Filename globbing patterns.
  - Regular expressions.
- Filename globbing patterns are mainly used on filenames.
- Regular expressions are mainly used on text.
- Globbing patterns are “simpler” than regular expressions (not as expressive).

Extra material: Globbing patterns

- `*` matches any string
- `?` matches any single character
- `[ ]` matches one character in the set or range
- `[ ! ]` matches one character not in the set or range

- A globbing pattern must always match the full length of the query string. It is automatically “anchored” to both the start and end of the string.
- The patten `???` (three question marks) will therefore only ever match a string with exactly three characters.

Extra material: Globbing tests in bash

- The bash shell allows for matching the value of a variable against a globbing pattern.
- The syntax is “[[“$variable” == pattern]]” (note double brackets and “==”).
  - ```
    if [[ "$url" == "https://"* ]]; then
        echo "This is a HTTPS URL"
    fi
    ```
  - ```
    if [[ "$string" == *"*"* ]]; then
        echo "This string contains at least one * character"
    fi
    ```
  - ```
    if [[ "$name" == *.??? ]]; then
        echo "This filename ends with a 3-character suffix"
    fi
    ```
    ```

Extra material: (extended) Regular expressions

- `.` matches any single character
- `[ ]` matches one character in the set or range
- `[^ ]` matches one character not in the set or range
- `|` alternation (“this or that”)
- `( )` expression grouping
- `^` anchor expr. to the start of the string
- `$` anchor expr. to the end of the string

Modifiers:

- `?` matches previous expr. 0 or 1 times
- `*` matches zero or more of the previous
- `+` matches one or more of the previous
- `{n}` matches n of the previous
- `{n,m}` matches previous between n and m times
- `{n,}` matches prev. at least n times

- A regular expression is not automatically anchored, like globbing patterns are.
- The pattern `...` (three dots) matches any string that contains at least three characters.

Extra material: Regular expression tests in bash

- The bash shell allows for matching the value of a variable against a regular expression.
- The syntax is “[[“$variable” =~ regex]]” (note double brackets and “=~”).
  - ```
    if [[ "$url" =~ ^"https://" ]]; then
        echo "This is a HTTPS URL"
    fi
    ```
  - ```
    if [[ "$string" =~ "*" ]]; then
        echo "This string contains at least one * character"
    fi
    ```
  - ```
    if [[ "$name" =~ ".".{3}$ ]]; then
        echo "This filename ends with a 3-character suffix"
    fi
    ```
    ```

Extra material: Globbing patterns vs. Regular expressions

1.	`*`	any string
2.	`?`	any single character
3.	`[a-kq]`	one char, a through to k, or q
4.	`[!0-9.]`	one char, but not a digit or a dot
5.	`ab*yz`	starts with ab, ends with yz
6.	`*[0-9][0-9]*`	two digits anywhere
7.	`ab* cd*`	starts with ab or cd (need two separate tests!)
8.	---	
9.	`*abab*`	any string containing abab

1.	`.*`	any string
2.	`.`	any single character
3.	`[a-kq]`	one char, a through to k, or q
4.	`[^0-9.]`	one char, but not a digit or a dot
5.	`^ab.*yz$`	starts with ab, ends with yz
6.	`[0-9]{2}`	two digits anywhere
7.	`^(ab|cd)`	starts with ab or cd
8.	`^xb+`	any string starting with x followed by one or more b
9.	`(ab){2}`	any string containing abab

One final thing: Globbing tests with “case ... esac”

```
case "$name" in
    *.txt)
        echo "ends with .txt" ;;
    *.png)
        echo "ends with .png" ;;
    *.???)
        echo "other 3-char suffix" ;;
    *)
        echo "some other filename" ;;
esac
```

- This is the more portable way to do globbing pattern matches.
- The variable’s value is tested against the patterns in turn.
- The code associated with the first pattern that matches is executed.
- The code for a pattern is terminated by “;;” (double semicolon).
- More compact than “if ...; then ...; elif ...; then ...; elif ...; then ...; else ...; fi”

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 10”.

Shell scripting issues, where to find help?

- Examples of question-and-answers sites where one may ask questions about shell scripting and get peer-reviewed answers from experts (or at least real people):
  - The “Unix & Linux” StackExchange site: https://unix.stackexchange.com/
    - Primarily about Unix-related issues (not just shell scripting).
    - Requires well-formulated questions showing actual code, actual errors, actual expected outcome. Also requires that you’ve tried to solve your issue yourself and ideally can show what you’ve tried and how/why it failed. No effort = downvotes = no answers.
    - Read https://unix.stackexchange.com/help first.
  - The “Bioinformatics” StackExchange site: https://bioinformatics.stackexchange.com/
    - Primarily about tools used in bioinformatics, or scripts/code written to perform bioinformatics-related tasks (no general shell-related questions).
    - See above.
    - Read https://bioinformatics.stackexchange.com/help first.