Bash Scripting

657.019 Scripting for Biotechnologists (WS 2018/19)

Outline

- Running shell scripts
- Arguments
- Loops
- Conditionals
- Variables
- Operators
- Functions

What is a shell script?

- Commands collected in a text file
- What we do:
  - In a text file, store a sequence of frequently used commands to do something
  - Run (or execute) this file (or script), instead of typing multiple commands each time
  - Modify to perform a similar task
- Goal:
  - Get the computer to do repetitive, tedious tasks

What shells are there?

- Two major families of shells
  - Bourne Shell family: sh, bash
  - C Shell family: csh, tcsh

- We use Bourne again shell (Bash)
  - Standard shell in Linux
  - Most widely used

Scripts without arguments

- Put a sequence of commands in a file
  - One command per line
  - Just as you would have typed at command line
  - File name can be anything (`.sh` extension usual, but not required)

- Run the script file, like any other command
  - Without arguments, does the same thing each time
  - Not flexible, but still saves work

First example

status.sh

```
#!/bin/bash
echo -n "It is now: "
date # Print date and time
echo -n "I am logged on as: "
whoami # Print my user name
echo -n "This computer is called: "
hostname  # Print host name
echo -n "I am in the directory: "
pwd # Where am i?
echo -n "The number of people currently logged on: "
who | wc -l # How many people are using the machine?
```

- Printing by **echo** (without **-n**, line ends)
- After **#** is comment up to end of line, except in the first line

#!/bin/bash

- Normally first two characters of script are **#!**
  - What follows is the path to the shell to use
  - Called 'shebang'
- Two rules
  - **#!** must be on the very first line of file
  - No space before # or after !
- Without #!
  - Script is run by user's current shell
  - No guarantee Bash will be used (remember, many different shells exist)
- Best to always have **#!/bin/bash** as the first line of your script

Running a script

- Make the file executable (runnable)

```
$ chmod +x status.sh
```

- Run it with `./` in front to indicate you want to run the script file in the current directory

```
$ ./status.sh
```

Scripts with one argument

- Argument given after script name
- **$1** to use the first argument

funix.sh

```
#!/bin/bash
# Find UNIX, Unix, uNiX, unix, etc from a file
grep -i unix $1
```

- Run it on the test file **unix.txt** to find occurrences of **unix**, case ignored

```
$ chmod +x funix.sh
$ funix.sh unix.txt
```

Using arguments

- Pre-defined variables are set automatically when you run a Bash script

- `$1,$2,$3`, … : first, second, third, … argument
- `$0` : name of the Bash script file
- `$@` : all the arguments supplied
- `$#` : number of arguments supplied

Scripts with multiple arguments

- `$@` : all arguments (no matter how many)

`funix_multi.sh`

```
#!/bin/bash
# Find UNIX, Unix, uNiX, unix, etc from multiple files
grep -i unix $@
```

- Run it on multiple text files

```
$ funix_multi.sh unix.txt unix2.txt
$ funix_multi.sh unix*.txt
```

Multiple ways to do one thing

- Give all files together to **grep**

funix.sh

```
#!/bin/bash
# Find UNIX, Unix, uNiX, unix, etc from multiple files
grep -i unix $@
```

- Give one file at a time to **grep**

funix_for.sh

```
#!/bin/bash
# Find UNIX, Unix, uNiX, unix, etc from multiple files
for i in $@
do
  grep -i unix $i
done
```

for ~ in ~ do ~ done

```
#!/bin/bash
for file in $@
do
  grep -i unix $file
done
```

- A **for** loop
  - takes a variable (**file**) and a list of items (**$@** = all arguments)
  - assigns each item to the variable, one at a time
  - does something on/with current assigned value (**$file**)
- What follows **in** must be a list
- Variable name can be anything, value accessed by adding **$** in front

3 kinds of loops

- `for` loops
  - Most useful to do something on items of interest
    - files, directories, species names, gene IDs, protein families, …
  - Most important in this course, and scripting in general
  - Easier to build and think about than other kinds of loops
- `while` loops
  - Moderately useful – will cover later
- `until` loops
  - Least useful – can always be replaced by a `while` loop

Checking number of arguments

funix_check_args.sh

```
#!/bin/bash
# Check that we have at least one argument
if [ $# -lt 1 ]; then
  echo $0 needs at least one file name.
  exit 1
fi
grep –i unix $@
```

- Good habit to check the number of arguments
  - Prevent unexpected behaviour
  - Sanity check is more work but pays off

If ~ then ~ fi

- Do some action if certain condition is met (otherwise do nothing)

```
#!/bin/bash
if [ $# -lt 1 ] # test condition inside [ ]
then
  echo "$0 needs at least one file name." # Action
  exit 1                                  # Action
fi # End of if statement

grep -i unix $@
```

- What if we want to do something otherwise?

If ~ then ~ else ~ fi

- Two-way comparison with **else**

```
#!/bin/bash
if [ $# -lt 1 ]
then
  # Actions for bad arguments
  echo "$0 needs at least one file name."
  exit 1
else
  # Action for good arguments
  echo "Good, we have at least one file."
fi

grep –i unix $@
```

- But there could be more than two outcomes

If ~ then ~ elif ~ then ~ else ~ fi

- Multi-way comparison with **elif**(s)

```
#!/bin/bash
if [ $# -lt 1 ]; then
  # Actions for no arguments
  echo "$0 needs at least one file name."
  exit 1
elif [ $# -lt 2 ]; then
  # Action for one argument
  echo We have exactly one file.
else
  # Action for >1 arguments
  echo We have two or more files.
fi

grep –i unix $@
```

Tests

- Square brackets `[]` is a shorthand for `test` command
  - `[ EXPRESSION ]` = `test EXPRESSION`
  - `[ $# -lt 1 ]` = `test $# -lt 1` : Test if the number of arguments is less than 1

- There **must** be a space after `[` and before `]`, because they are interpreted as literal `test` command
  - `[EXPRESSION]` = `testEXPRESSION` → Cannot be interpreted

File exists?

lcnt_check_exists.sh

```
#!/bin/bash
if [ $# -lt 1 ]; then  # Check for argument
  echo $0 needs a file name.
  exit 1
fi
if [ -e $1 ]; then     # file exist - count lines
  wc -l $1
else                   # file does not exist - complain
  echo File $1 not found.
fi
```

- Good to check if file exists before running a command on file

File queries

Expression	What's tested
`-e file`	***file*** exists?
`-r file`	***file*** exists and is readable by user?
`-w file`	***file*** exists and is writable by user?
`-x file`	***file*** exists and is executable by user?
`-s file`	***file*** exists and has size greater than 0?
`-f file`	***file*** exists and is a regular file? (rather than a directory)
`-d file`	***file*** exists and is a directory? (rather than a file)
`-O file`	***file*** exists and is owned by user?

Equality: string vs number

equal.sh

```
#!/bin/bash
number1=14                  # Set a variable
echo -n "I have $number1. Enter a number: "  # Prompt
read number2                # Get user input

if [ $number1 -eq $number2 ]; then
  echo -n "$number1 and $number2 are equal numbers, "
  if [ $number1 = $number2 ]; then      # Nested if
    echo and equal strings.
  else
    echo but different strings.
  fi
else
  echo $number1 and $number2 are different numbers.
fi
```

- String equality `=` tests character by character sameness
- Numeric equality `-eq` tests number value sameness

Comparison operators

Expression	Test if
`-n STRING` `STRING`	Length of `STRING` is non-zero
`-z STRING`	Length of `STRING` is zero (`STRING` is empty)
`STRING1 = STRING2`	`STRING1` and `STRING2` are equal
`STRING1 != STRING2`	`STRING1` and `STRING2` are not equal
`INTEGER1 -eq INTEGER2`	`INTEGER1` and `INTEGER2` are equal
`INTEGER1 -ne INTEGER2`	`INTEGER1` and `INTEGER2` are equal
`INTEGER1 -gt INTEGER2`	`INTEGER1` is greater than `INTEGER2`
`INTEGER1 -ge INTEGER2`	`INTEGER1` is greater than or equal to `INTEGER2`
`INTEGER1 -lt INTEGER2`	`INTEGER1` is less than `INTEGER2`
`INTEGER1 -le INTEGER2`	`INTEGER1` is less than or equal to `INTEGER2`

Equality: string vs number

```
$ ./equal.sh
I have 14. Enter a number: 14
14 and 14 are equal numbers, and equal strings.
$ ./equal.sh
I have 14. Enter a number: 0014
14 and 0014 are equal numbers, but different strings.
$ ./equal.sh
I have 14. Enter a number:          019
14 and 019 are different numbers.
```

- Use proper equal test, depending on whether you want treat variables as strings or numbers

Assigning value to variable

string_quotes.sh

```
#!/bin/bash

city1=Graz
city2="Bruck an der Mur"
echo Both $city1 and $city2 are in Steiermark.
```

```
$ string_quotes.sh
Both Graz and Bruck an der Mur are in Steiermark.
```

- **variable=value**
  - Assign **value** to **variable**
  - No space before or after **=**
  - Use quotes to treat multiple things as one

User input: `read`

- **`read variable`**
  - Waits (forever) for the user to enter something
  - Assigns what user types to **`variable`** (leading spaces trimmed)
- Preceding it with a specific prompt is a very good idea

```
echo "Enter something"                # vague
read something
echo -n "Please type a number between 1 and 100: "
read number
echo "Date of birth (ddmmyy)? "   # specific
read dob
```

Options for `read`

`login.sh`

```
#!/bin/bash
# Authenticate login credentials
read -p "Username: " user
read -sp "Password: " password
echo
echo Thank you $user!
```

```
$ ./login.sh
Username: jsoh
Password:
Thank you jsoh!
```

- Two most useful options
  - **-p**: Prompt to print
  - **-s**: Silent, do not show what is typed

Command substitution

command_sub.sh

```
#!/bin/bash
# Ask for file name if not supplied
if [ $# -lt 1 ]; then
  read -p "$0 needs a file name: " file
else
  file=$1
fi
# Assign command output to a variable
num_users=$(who | wc -l)
# Use command output directly
echo $(hostname) has $num_users current users.
echo There are $(cat $file | wc -l) lines in $file.
```

```
$ ./command_sub.sh unix2.txt
cbt01.cbt.tugraz.at has 13 current users.
There are 7 lines in unix2.txt.
```

- Surround command with `$(` and `)`
- Use command output directly or assign it to a variable

Literal lists

cities_for.sh

```
#!/bin/bash
# Iterate through a list of cities
for city in Graz "Bruck an der Mur" Kapfenberg; do
  echo $city is a city in Steiermark.
end
```

```
$ cities_for.sh
Graz is a city in Steiermark.
Bruck an der Mur is a city in Steiermark.
Kapfenberg is a city in Steiermark.
```

- Use quotes to include spaces in an item

Number ranges

range_by1.sh

```
#!/bin/bash
# Iterate through
# a range of numbers
for number in {1..10}
do
  echo $number
done
```

- No space within { }
- Start value can be bigger
- Step can be bigger than 1

- Output

```
$ ./range_by1.sh
1
2
3
4
5
6
7
8
9
10
```

Number ranges

range_by2.sh

```
#!/bin/bash
# Iterate through
# a range of numbers
for number in {10..-10..2}
do
  echo $number
done
```

- Step given after end number
- Number can go down

- Output

```
$ ,/range_by2.sh
10
8
6
4
2
0
-2
-4
-6
-8
-10
```

List by command substitution

- Use with **`cat`** or **`ls`** to generate a list of files/items

```
#/bin/bash
# All files in current directory
for file in $(ls); do
  # run commands on $file
done
# Text files in my home directory
for k in $(ls ~/*.txt); do
  # run commands on $k
done
# A list of proteins in a text file (one per line)
for prot in $(cat my_proteints.txt); do
  # run commands on $prot
done
```

Predefined variables

- Arguments: `$1`, `$2`, …., `$0`, `$@`, `$#`
- Exit status of most recently run process: `$?`
- Process ID of current script: `$$`
- Some others
  - `$USER`, `$HOSTNAME`, `$PWD`, `$HOME`
  - `$LINENO`: current line number in script
  - `$RANDOM`: random number
- `env` command shows all variables set by system

Arithmetic

- Surround expression with `(( )` and `)`, add `$` to assign

arithmetic.sh

```
#/bin/bash
a = $(( 3 + 9 ))          # 12
((a++))                   # 13 (increment by 1)
(( a-- ))                 # 12 (decrement by 1)
b = $((7+14))             # 21
(( b+=2 ))                # 23 (add 2)
num = $(( 4*6 ))          # 24
quot = $(( num/9))        # 2 (quotient when divided by 9)
rem = $(( num % 9 ))      # 6 (remainder when divided by 9)
z="Hello, world!"
echo ${#z}                # 13
echo ${#a}                # 2
```

- No restriction on use of spaces inside `(( ))`
- `${#var}` gives the length of (=number of characters in) variable **`var`**

Logical operators

logical.sh

```
#!/bin/bash
num=$1
# not num = 10
if [ ! $num -eq 10 ]; then
  echo Number $num is not 10
fi
# 10 < num < 20
if [[ $num -gt 10 && $num -lt 20 ]]; then
  echo Number $num is greater than 10 and less than 20
fi
# num < 10 or num > 20
if [[ $num -lt 10 || $num -gt 20 ]]; then
  echo Number $num is less than 10 or greater than 20
fi
# not (num <= 10 or num >= 20); same as 10 < num < 20
if [[ !($num -le 10 || $num -ge 20) ]]; then
  echo Number $num is greater than 10 and less than 20
fi
```

Logical operators

Expression	What's tested
`[ ! EXPR ]`	Negation: `EXPR` is false
`[[ EXPR1 && EXPR2 ]]`	Logical AND: Both `EXPR1` and `EXPR2` are true
`[[ EXPR1 || EXPR2 ]]`	Logical OR: At least one of `EXPR1` and `EXPR2` is true

- `[[ ]]` required to use `&&` or `||`
- Use `( )` to group expressions

Double and single quotes

quotes.sh

```
#!/bin/bash
x=1
y=2
message="current values are:"
echo "Double quotes - $message x = $x, y = $y"
echo 'Single quotes - $message x = $x, y = $y'
```

```
$ ./quotes.sh
Double quotes - current values are: x = 1, y = 2
Single quotes - $message x = $x, y = $y
```

- Double quotes (or no quotes): evaluate before using
- Single quotes: use literally

Arrays

arrays.sh

```
#!/bin/bash
array=(one two three four)      # Create an array
echo ${array[2]}                # three
echo ${array[@]}                # one two three four
array[1]=zwo                    # Change second element
echo ${array[@]}                # one zwo three four
echo ${#array[@]}               # Get array size (4)
array[4]=five                   # Add element
array+=(six)                    # Add another element
echo ${array[@]}                # one zwo three four five six
```

- Array = list (for our purpose)
- `${}` to get the value
- `()` : list, `#` : size (number of items), `@` : all, `[]` : index

Loops on arrays

array_loops.sh

```
#!/bin/bash
names=(harry sally mary tony) # Create an array
# Find number of items in the array
echo -n "We have ${#names[@]} names:"
# Print names one by one with for loop
for name in ${names[@]}; do
  echo -n " $name"
done; echo
echo -n "names again:"
# Print names one by one with while loop
k=0
while [ $k -lt ${#names[@]} ]; do
  echo -n " ${names[$k]}"
  ((k++))
done; echo
```

for or while?

```
#!/bin/bash
names=(harry sally mary tony)
# Print names with for loop
for name in ${names[@]}; do                    # Set variable & list
  echo $name                                   # Use variable
done
# Print names with while loop
k=0                                            # Initialize index
while [ $k -lt ${#names[@]} ]; do              # Set test condition
  echo ${names[$k]}                            # Use indexing notation
  ((k++))                                      # Increment index
done
```

- **`for`** loops are simpler to use most of time
- Two main uses of **`while`**
  - When index values themselves are required
  - To process lines of a file

Reading lines of a file

cat.sh

```
#!/bin/bash
if [ $# -lt 1 ]; then
  read -p "File to read from? " file
else
  file=$1
fi
IFS=""  # Set to empty to preserve formatting
# Read a line from file, assign to variable (line)
while read line
do
  echo $line            # Process each line
done < $file
```

- Doing something for each line of a file
  - Very common task in bioinformatics

Functions

functions.sh

```
#!/bin/bash
num_lines() {          # Get the number of lines of file
  cat $1 | wc -l
}
last_modified() {      # Get the last modified time of file
  ls -l $1 | gawk '{print $6,$7,$8}'
}
print_message() {      # Print some message
  echo Welcome $USER!
  echo Your current directory is $PWD.
  echo The script $0 received $# arguments.
}

# Use the functions as commands
print_message; num_lines $1; last_modified $1
# Use command substitutions to build better output
echo "$1 has $(num_lines $1) lines and was last modified
$(last_modified $1)."
```

Functions

```
$ ./functions.sh unix2.txt
Welcome jsoh!
Your current directory is /export/home/jsoh/lab3.
The script ./functions.sh received 0 arguments.
7
Oct 7 15:44
unix2.txt has 7 lines and was last modified Oct 7 15:44.
```

- One well-defined task per function
  - Not too complicated, not too trivial
  - Give meaningful names to functions
- Define before first use

Debugging

- Echo commands before execution
  - Add `-v` option to shebang line: `#!/bin/bash -v`
  - Or do: `bash -v script.sh`

- Print all details
  - `#!/bin/bash -x`
  - `bash -x script.sh`

- Check syntax error without execution
  - `#!/bin/bash -n`
  - `bash -n script.sh`

Summary

- Shell scripts are used mainly for
  - Collecting a series of repetitive tasks in a file, to be reused
  - Building a simple command pipeline, to be used as a single command
- Shell scripts provide a framework for tackling a task
  - Loops, conditionals, functions etc. are structuring elements
  - Actual analysis (*e.g.* using a Perl one-liner or Python script) can be inserted into the framework and controlled