Foundations of Data Science for Biologists

Unix shell scripts

BIO 724D

20-NOV-2023

Instructors: Greg Wray and Paul Magwene

Basics of Unix shell scripting

What is a shell script?

A mechanism for automating commands in a Unix shell

When the script is run, commands listed in the file are executed one line at a time

Any valid shell command can be incorporated, including:

- Standard shell commands with options, redirection, and pipes
- Third-party programs
- Other shell scripts

Why write shell scripts?

Record how to carry out operations: make them re-usable and modifiable

Simplify common tasks: type one line instead of many

Automate tasks: includes scheduling one-time and recurring tasks

Make your work flows **reproducible**: including file operations, data wrangling, etc.

Create and run a shell script

Create a file:

```
>nano my_script.sh
```

extension not required, but recommended

Type the “shebang” followed by your commands:

```
#!/usr/bin/env bash
<commands go here>
```

required; no extra line or space

add commands and comments

… then save and exit (`^O return ^X`)

Invoke the script:

```
>bash my_script.sh
```

runs the script

A very simple script

```
#!/usr/bin/env bash

echo “Hello world”
```

A simple script with an argument

```
#!/usr/bin/env bash

echo “Hello, $1, how are you today?”
```

A script to count directory entries

```
#!/usr/bin/env bash

lines=$(ls $1 | wc -l)

if [[ $# -ne 1 ]]
then
 echo "Error: please provide a single valid directory path"
 exit 1
fi

echo "The directory $1 contains $(($lines-1)) objects."
```

Variables in scripts

Create variables simply by assigning a value:

`my_var='Charles Darwin'`	assigns the string `Charles Darwin`
`my_var=42`	assigns the string `42`
`DECLARE -i my_var=42`	assigns the integer `42`
`DECLARE -4 my_var=Hello`	assigns the string `Hello` as read-only

Refer to variables by adding `$` to the variable name:

`echo $my_var`	returns `Charles Darwin`

Type variable names carefully; referring to an unassigned variable returns an empty string

Expansion during assignment

It is possible to assign the output of a command directly to a variable:

```
lines=$(ls | wc -l)
```

called **command expansion**

It is also possible to assign the value of a simple mathematical expression:

```
new_var=$((5 - 1))
new_var=$((lines -1))
```

called **arithmetic expansion**

more useful when combined with variables

Use single brackets for command expansion and double for arithmetic expansion

Do not insert spaces between variable name, = and $

Both kinds of expansions automatically assign numeric values when appropriate

Passing arguments to scripts

You can pass one or more arguments to a script:

```
>bash my_script.sh ../data
```

passes the path

Any tokens encountered after the script's name are treated as arguments

Refer to arguments within the script using `$1`, `$2`, etc. for the first, second, etc.:

```
lines=$(ls $1 | wc -l)
```

expands `$1` into the path

Built-in variables available to scripts

argument variables — arguments passed to your script; also called positional variables

`$1`	first argument
`$2`	second argument; works up to 9; for higher numbers use `{10}`, `{11}`, etc.
`$!`	all of the arguments passed
`$#`	number of arguments passed

process variables — metadata about the current script

`$0`	name of the script
`$?`	the exit status of the most recently run command within the script

environment variables — information about the current environment

`$USER`	the user name of the person who is logged in
`$HOME`	path to the current user's home directory
`$PATH`	set of paths used to search for commands

Using a built-in numeric variable

You can retrieve the number of variables passed to the script using `$#`:

```
num_args=$#
```

In the script that counts directory entries, `$#` is used to trap errors:

```
if [[ $# -ne 1 ]]
```

The double square brackets contain the test condition for an if structure

Read this as: *if the number of arguments is not equal to 1*

Interrupting script execution

Sometimes it is useful to be able to terminate scripts before they get to the end

- Report an error
- Avoid unnecessary computation that is time-intensive

You can terminate (exit) your bash script with an `exit` command:

```
exit 1
```

the integer indicates exit status

Similar to `break` in R

Exit status

Commands and programs return an **exit status** when they finish running

Exit status is an integer (0-255) that indicates success or error

0	success (no errors during execution)
1	an error occurred
126	command found but could not be executed
127	command not found

Values > 1 can indicate the specific kind of error, but are not fully standardized

Comment your code

Use # as the first character of a line to indicate comments

```
#!/usr/bin/env bash

# script to count the number of entries in a directory
# usage: provide the path to the directory as an argument

lines=$(ls $1 | wc -l)

if [[ $# -ne 1 ]]
then
 echo "Error: please provide a single valid directory path"
 exit 1
fi

echo "The directory $1 contains $(($lines-1)) objects."
```