Foundations of Data Science for Biologists

Scripting in Unix

BIO 724D

2025-FEB-11

Instructors: Greg Wray and Paul Magwene

Disambiguation: scripts and programs

Scripts and **programs** both consist of a set of instructions to a computer

Scripts are stored as plain text files that are **interpreted** when run

- The same script can usually run on different hardware and under different OSs
- Pros: simplifies reproducibility and sharing code, future-proofs code
- Con: slower execution
- Examples: R, SQL, Python, Unix/bash, Awk

Programs are written in plain text but then **compiled** into machine code

- Stored as binary files with a special flag that tells the OSs that they are programs
- Pros: fast execution, efficient use of memory
- Cons: must re-compile for most combinations of hardware and OS
- Examples: C, C++, Rust, assembly language

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 line entry can be incorporated, including:

- Standard shell commands with options
- Input, output, redirection, and pipes
- Awk programs
- Other shell scripts
- Third-party programs (if they can be invoked from the command line)

Simple Unix script

Create and run a shell script

Create a file and edit in VSCode

```
>touch my_script.sh
```

extension not required, but recommended

Type the “shebang” (or “hashbang”) on the first line, followed by your commands:

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

required; no extra line or space

add commands and comments

Invoke the script:

```
>bash my_script.sh
```

runs the script

A very simple script

```
#!/bin/bash

echo "Hello world"
```

A simple script with an argument

```
#!/bin/bash

echo "Hello $1, how are you today?"
```

A script to count directory entries

```
#!/bin/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} objects."
```

Comment your code!

```
#!/bin/bash

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

lines=$(ls -la $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} objects."
```

Basics of Unix shell scripting

Variables in scripts

Create variables by assigning a value:

```
my_var="Charles Darwin”
my_var=42
declare -i my_var=42
declare -r my_var=Hello
```

quotes required unless a single word

assigns the string `42` (unlike R and Python)

assigns the integer `42`

assigns the string `Hello` as read-only

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

```
echo $my_var
echo ${my_var}
```

returns the current value of `my_var`

“safe” way to return string values

Type variable names carefully; referencing an unassigned variable returns an empty string!

Referencing string variables

To incorporate string variables into longer strings:

- Use the “safe” string reference method with curly braces
- Use double quotes

```
v1="data"
echo $v1
echo 'Where is the ${v1}'
echo "Where is the ${v1}"
touch "${v1}_A.txt"
touch '${v1}_A.txt'
```

assigns a string variable

correctly returns the value of `v1`

does not substitute the value of `v1`

now it substitutes the value of `v1`

creates the file `data_A.txt`

creates the file `${v1}_A.txt`

Expansion during assignment

Use `$( )` for **command expansion** (substitution) during assignment:

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

assigns an integer to `lines`

assigns a long string to `lines`

Use `$(( ))` for **arithmetic expansion** (substitution) during assignment:

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

assigns an integer to `new_var`

much more more useful with variables!

Do not insert spaces between `variable_name`, `=` and, `$`

Both kinds of expansions automatically assign numeric values when appropriate

Passing arguments to scripts

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

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

passes the path and string `hello`

Within the script, refer to arguments as `$1`, `$2`, etc. for the first, second, etc.:

```
lines=$(ls $1 | wc -l)
echo "${2} world"
```

expands `$1` into the path

returns `hello world`

Built-in variables available to scripts

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

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

process variables: metadata about the current script

`$0`	name of the script
`$LINENO`	the current line number within the script
`$?`	the exit status of the most recently run command within the script

environment variables: information about the current environment (there are many more)

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

Exit status

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

Exit codes:

- 0 success (no errors during execution)
- 1+ failure (an error occurred)

Notes:

- Specific values >1 can indicate the type of error, but are not standardized
- You can provide a custom error code when using the `exit` command

Interrupting script execution

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

- Report an error
- Debug your code
- Avoid unnecessary computation that is time-intensive

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

```
exit
exit 7
```

halts execution; exit status = 1 (default)

same but with exit status = 7

Similar to `break` in R

Conditions for control flow in Unix

Specifying conditions

Conditions can be specified using three different formats (see next two slides):

`[ <condition> ]`	basic
`[[ <condition> ]]`	provides extended features
`(( <condition> ))`	cleaner syntax for integer conditions

Leave a space on both sides of the opening and closing brackets

Only `[ ]` is POSIX, which makes it more portable

Conditions with strings

Use single brackets unless you need special features:

```
[ $var1 = "hello" ]
[ $var1 != $var2 ]
[ -n $var1 ]
[ -z $var1 ]
```

tests for string identity (`==` in some shells)

tests for string difference

tests for non-zero string length

tests for zero string length (empty variable)

Use double brackets for the following features:

```
[[ $var1 == "hello world" ]]
[[ $var1 =~ [cw]ow ]]
[[ $var1 == *hello* ]]
```

allows spaces in string literals

`=~` operator tests for regex within `var1`

globbing searches for string within `var1`

Conditions with integers

Use square brackets for portability:

```
[ $var1 -eq 42 ]
[ $var1 -ne $var2 ]
[ $var1 -gt 42 ]
[ $var1 -ge $var2 ]
```

tests for integer equality

tests for integer inequality

tests for grater than (also: `-lt`)

tests for greater than or equals (also: `-le`)

Use double round brackets for readability:

```
(( $var1 == 42 ))
(( $var1 != $var2 ))
(( $var1 > 42 ))
(( $var1 >= $var2 ))
```

tests for integer equality

tests for integer inequality

tests for grater than (also: `<`)

tests for greater than or equals (also: `<=`)

Conditions with files

Use the `-f` option to test for the existence of files:

`[ -f my_file.txt ]`	tests for `my_file.txt` in pwd
`[ -f *.txt ]`	tests for any file ending in `.txt`
`[ ! -f *.txt ]`	tests for absence of files ending in `.txt`
`[ -f ../project_01/*.txt ]`	tests for any `.txt` file in another directory

Additional options provide flexibility and specificity (see `[` man page for full list):

`[ -d m* ]`	tests for directory starting with `m`
`[ -e m* ]`	tests for file or directory starting with `m`
`[ -s my_file.txt ]`	tests existence and non-zero size
`[ -w my_file.txt ]`	tests existence and whether writable
`[ -x file_name ]`	tests existence and whether executable

Boolean operators for conditions

Use `!` to negate a condition:

```
[ ! -f *.txt ]
[ ! $var1 -eq 100 ]
```

tests for absence of files matching pattern

avoid negation if an operator exists!

How to specify AND and OR conditions:

```
[ $v1 > 10 -a $v2 == 42 ]
[ $v1 > 10 ] && [ $v2 == 42 ]
[[ $v1 > 10 && $v2 == 42 ]]
[ $v1 > 10 -o $v2 == 42 ]
[ $v1 > 10 ] || [ $v2 == 42 ]
[[ $v1 > 10 || $v2 == 42 ]]
```

condition1 AND condition2

same as above

same as above

condition1 OR condition2

same as above

same as above

Control flow in Unix scripts

Basics of control flow

Most shells offer several different control flow structures:

- Conditional: `if`, `if/else`, `if/elif`, `if/elif/else`, `case`, `case/*`
- Loop: `while`, `until`, `for`

Nesting and mixing control flow structures is allowed

- As with all languages, avoid nesting more than 2 deep whenever possible

Indenting is not required but highly recommended for readability

Conditional flow

Template for **if** and **if/else**:

```
if [ <condition> ]
then
   <commands>
fi
```

```
if [ <condition> ]
then
   <commands>
else
   <commands>
fi
```

Include an `elif` block to specify alternative conditions (with or without an `else` block)

More than 1 `elif` can be included, but for many different conditions `case` is clearer

Conditional loops

Template for a **while** loop:

```
while [ <condition> ]
do
   <commands>
done
```

runs as long as condition evaluates to **true**

Template for an **until** loop:

```
until [ <condition> ]
do
   <commands>
done
```

runs as long as condition evaluates to **false**

Important: include a way of terminating the loop! (See discussion of while loops in R)

Counting loops

Template for a **do** loop:

```
for <variable> in <set>
do
   <commands>
done
```

executes once for each item in the set

executes in the presented order of items

Examples of simple sets:

```
for i in 1 2 3 4 5
for i in J Q 6 A 99 B
for i in ant bug bee moth
for i in {1..5}
for i in {5..-5}
```

executes 5 times; `i` takes values 1…5

executes 6 times; `i` takes specified values

executes 4 times; `i` takes specified values

executes 5 times; `i` takes values 1…5

executes 11 times; `i` takes values 5…-5

Counting loops, continued

Use the `seq` command for more control over the set:

```
s=1 ; t=2; e=10
for i in $(seq $s $e)
for i in $(seq $s $t $e)
```

set variables in advance (not required)

executes 10 times; `i` takes values 1..10

executes 5 times; `i` = 1, 3, 5, 7, 9

When 3 values are pass to `seq` they are interpreted as start, step, and end

Note that `seq` requires start to be smaller in value then end (won't work in reverse)

To specify a **set of files**, simply provide a path; globbing is allowed:

```
for f in *
for f in ./*.txt
for f in ../analysis/*
```

iterates over every file in current directory

iterates over files with `.txt` extension only

iterates over files in a different directory

Functions in Unix scripts

Defining and calling functions

Template for a **function**:

```
<func_name> () {
    <commands>
}
```

To call a function, simply give its name:

```
$ my_function
```

Do *not* include brackets, as in most other languages (R, Python, etc.)

Passing arguments to a function

To pass arguments to a function, place them after the function name:

```
greet () {
   echo "Hello $1!"
}

greet "Rosalind Franklin"
```

simply include an **argument variable** within the function (no need to declare)

call the function and include an argument

This is just like passing arguments to a command or script

Warning: often no error if you supply to many or too few arguments