Using Shells and Scripting (slides)

In the intro portion of the workshop you will learn:

- What are shells?
- Which one(s) to use?
- What is scripting and what are its applications?
- How to write scripts.
- How to use some useful tools in scripts.

What are shells?

The program that is started after logging on a Linux machine and that allows users to type commands (aka the login shell). For historical reasons there are several shells:

1. The Bourne shell (`sh`) or the Bourne again shell (`bash`)
2. The C shell (`csh`) or the tC shell (`tcsh`)
3. The Korn shell (`ksh`)
4. The Z shell (`zsh`)

You can check the Introduction to the Command Line for Genomics Carpentries lesson for an intro to `bash`.

> *Everything about each shell is explained in the "man pages", there are books written about scripting and there is a ton of info on the web too.*

Which shell(s) to use?

On Linux machines `sh` and `bash` are the same program, `csh` and `tcsh` are also the same program.

- `bash` is the shell most used by biologists (and used to install packages for historical reasons)
- `csh` is the shell most used by astronomers.

  The `csh` was written after `sh` to use a syntax more like the C programming language (hence the name), while the `tcsh` is an improvement to the C shell for terminal use (hence the `t`).
- a few people and systems use the Korn shell.

You can write scripts in any shell syntax you care to, independently of what your login shell is.

What is scripting and what are its applications?

Scripting vs. Programming

- Programming languages are sets of instructions compiled to produce executables with machine level instructions (a more “complicated” process).
- By contrast, scripting refers to sets of instructions that are parsed and executed by a program, hence
  - there is no need to compile the script (convenient)
  - but, *in principle*, they run more slowly than executables.
- Scripts can combine existing modules or components, while programming languages are used to build more sophisticated/complicated applications from scratch.

Scripting vs. Programming (cont'd)

- Some scripting languages are parsed to check for syntax errors before executing them, but *not* shell scripts.
- Scripts are used to help run applications, on Hydra a job file is a (simple) script.
- You can write complex scripts, and scripts can invoke (i.e., start) other scripts.

How to write scripts?

- A script is a text file that holds a list of commands, and thus can be written with any type of editor (`nano`, `vi`, `emacs`).
- The commands in the script must follow the syntax of the shell used to parse the script.
- A script can take arguments (options) and thus be more general, it allows you to define variables, use expressions or execute commands
  - A variable is a mechanism to hold a value and refer to it by its name, or a way to modify how some commands behave - variables can also be one dimensional arrays (lists)
  - An expression allows a user to perform simple arithmetic or use commands to create values held by variables (for example: set the variable `num` to hold the number of lines in a file).

How to write scripts (cont'd)

- Scripting syntax allows for "flow control" namely it allows for
  - tests and logical operators - `if` statements
  - loops - `for` statements
  - more flow control: `case`, `while`, `until` and `select`
  - `bash` also allows users to define functions (not covered)
  - the precise syntax is shell specific, i.e. `[ba]sh` syntax is different from `[t]csh` syntax.
- Scripts allow for I/O redirection
  - input: aka `stdin`
  - output: aka `stdout`
  - error: aka `stderr`
  - pipes: redirecting output of one command to be the input to another command

Sophisticated shell scripting is akin to programming, we can't & won't teach programming today.

We will focus on basic skills and make time for hands-on learning.

There is more than one way to bake a cake, which makes things confusing at first.

Script Variables

What are variables

- A variable is a character string to which a value is assigned.
- The assigned value can be a number, some text, a filename, a device, or any other type of data.
- The value of a variable is obtained by using $ followed by the variable name.

Variable Examples

bash syntax

```
filename=/here/goes/nothing.txt
string='hello class'
let num=33
echo $filename $string $num
```

csh syntax

```
set filename = /here/goes/nothing.txt
set string = 'hello class'
@ num = 33
echo $filename $string $num
```

Result in both cases

```
/here/goes/nothing.txt hello class 33
```

Quotes when setting variables

Single quotes ' vs double quotes "

```
blah='tell me more'
name0=hello
name1='hello $blah'
name2="hello $blah"
echo name1=$name1
echo name2=$name2
```

In case of doubt, use { and }

```
blah='tell me more'
name1='hello $blah'
name2="hello ${blah}"
echo name1=${name1}
echo name2=${name2}
```

Output:

```
name1=hello $blah
name2=hello tell me more
```

Variable arrays

bash syntax

```
arr=(1 2 3)
echo ${arr[@]}
1 2 3
echo ${#arr[@]}
3
echo ${arr[0]}
1
echo ${arr[1]}
2
```

Variable arrays (cont'd)

csh syntax

```
set arr = (1 2 3)
echo $arr
1 2 3
echo $#arr
3
echo ${arr[1]}
1
echo ${arr[2]}
2
```

Also

Note 1-indexed vs 0-indexed

Exporting variables

The `export` command in `bash`

- makes the exported variables seen by child processes, like programs or scripts
- example

```
cat hello.sh
echo hello=$hello
```

- result:

```
hello='yo!'
echo hello=$hello
sh hello.sh
hello=

export hello
sh hello.sh
hello=yo!
```

csh equivalent is `setenv`

csh uses `setenv`

```
setenv hello 'yo!'
```

Note no '=' sign

Handling full paths

bash with basename & dirname

```
path=to/some/where/some.txt
base=$(basename $path)
file=$(dirname $path)
echo base=$base file=$file
```

- `base=some.txt file=to/some/where`

csh, using the :h :t :r :e constructs

```
set path = to/some/where/some.txt
set base = $path:h
set file = $path:t
echo base=$base file=$file root=$file:r ext=$file:e
```

- `base=to/some/where file=some.txt root=some ext=txt`

:tea: Let’s pause here for 5-10 minutes :coffee:

Simple Arithmetic

Using `let` in bash

```
let x=1
let y=$x+3
echo x=$x y=$y
```

Using `@` in `csh`

```
@ x = 1
@ y = $x + 3
echo x=$x y=$y
```

output:

```
x=1 y=4
```

I/O and pipes

Various redirections

- redirect a command to read from a file: `command < file`
- redirect output and error to a file
  - `>`, `>>`, `&>`, `&>>`, and `2>&1`
- redirect output of one command as input to the next
  - `command1 | command2`
- read input from script or terminal

```
command <<EOF
some text
ok to use $variables
EOF
```

- duplicate output to show on the output and to be written in a file; the `tee` command:

```
command | tee file
```

Using $(command) in bash

Examples

- You can set a variable to be the result of a command

```
let x=1
let y=$x+3

Y=$(echo $y | awk '{printf "%3.3d", $1}')
echo Y=$Y
```

- output: Y=004

Using the backticks ‘ in csh and in bash

When using csh

```
@ x = 1
@ y = $x + 3

set Y = `echo $y | awk '{printf "%3.3d", $1}'`
echo Y=$Y
```

Equivalent in bash

```
let x=1
let y=$x+3

Y=`echo $y | awk '{printf "%3.3d", $1}'`
echo Y=$Y
```

Script Arguments

Arguments

- Arguments are a way to supply parameters to a script.
- Arguments are useful when a script has to perform differently depending on the values of some input parameters.

Example

- A trivial bash example, `echo.sh`

```
$ cat echo.sh
echo this is a demo of args
echo the script name is "'$0'"
echo "you have passed $# argument(s)"
echo the first argument is "'$1'"
echo the second argument is "'$2'"
echo etc...
```

How it works

No argument

```
$ sh echo.sh
this is a demo of args
the script name is 'echo.sh'
you have passed 0 argument(s)
the first argument is ''
the second argument is ''
etc...
```

Same with 3 arguments

```
$ sh echo.sh help me now
this is a demo of args
the script name is 'echo.sh'
you have passed 3 argument(s)
the first argument is 'help'
the second argument is 'me'
etc...
```

Script Flow Control

Tests and Logical Operators

- Logical operators can be used to test conditions and create complex expressions by combining conditions.
- These operators allow you to evaluate if a condition or multiple conditions is/are true, and provide a way to control the flow of execution of scripts.

The if statement

bash example

```
let num=$1
if [ $num -gt 5 ]
then
  echo this is big
else
  echo this is small
fi
```

Note

Note the blank spaces around the [, while none after a = when setting a variable.

The indentation is optional, and you can write this in a more compact way, using ;.

bash example (more compact notation)

```
let num=$1
if [ $num -gt 5 ]; then
  echo this is big
else
  echo this is small
fi
```

csh equivalent

```
@ num = $1
if ($num > 5) then
  echo this is big
else
  echo this is small
endif
```

Loops

Purpose

- Loops allow us to repeat a command or set of commands for each item in a list.

bash examples

```
for val in one two three
do
  echo val=$val
done

for val in $(ls)
do
  echo val=$val
done
```

Loops (cont'd)

bash loops on indices

```
for (( i=1; i<=5; i++ ))
do
   echo i=$i
done

for i in {0..10..2}
do
   echo i=$i
done
```

Other Flow Control Instructions, $?, || and && ($statusforcsh‘)

Other flow control

- case - multiple options “if”
- while - loop on a simple condition
- etc. . .

Checking command status and error check

- $?, || and && - error checking for bash
- $status - error checking for csh

The case, or a mutiple options test

In bash

```
$ cat case.sh
#!/bin.sh

var="$1"

case $var in
  big)
    num=300
    color=none
    ;;
  small)
    num=1
    color=none
    ;;
  blue*)
    num=0
```

The case (cont'd)

In csh equivalent

```
$ cat case.csh
#!/bin/csh

set var = "$1"

switch ($var)
case big:
    set num   = 300
    set color = none
    breaksw
case small:
    set num   = 1
    set color = none
    breaksw
case blue*:
    set num   = 0
```

Error status: check if command executed properly

bash

```
$ gerp junk test
sh: gerp: command not found
$ echo $?
127
$ grep junk test
grep: test: No such file or directory
$ echo $?
2
$ grep junk case.sh
$ echo $?
1
$ grep exit case.sh
    exit 1
$ echo $?
0
$ grep junk case.sh  || echo no junk
```

Making scripts excutables

To start a script w/out the sh or csh in front

```
chmod +x script
```

Best practice to add as 1st line

```
#!/bin/sh
```

or

```
#!/bin/csh
```

Can run the script using its name if in the path

bash

```
PATH=($PATH /where/the/script/is)
```

csh

```
set path = ($path  /where/the/script/is)
rehash
```

alternatively

- edit `.bashrc`, `.cshrc` or use a module

Useful tools for scripting

Simple ones

- `sed` - the stream editor for filtering and transforming text
- `awk` - pattern scanning and processing language
- `grep` - search a file for a pattern
- `tr` - translate or delete characters
- `cut` - remove sections from each line of files
- `bc` - An arbitrary precision calculator language
- `date` + -date="specification" - date handling

More sophisticated ones

- `PERL` - Practical Extraction and Report Language
- `Python` - high level general purpose programming language
- `Tcl` - Tool Command Language
- etc. . .

sed - the stream editor for filtering and transforming text

Examples

```
sed 's/this/that/' file > newfile

sed 's/this/that/g'
sed -e 's/one/two' -e 's/blah/foo/'
sed -e "s/XXX/$values/" template.txt > input.txt
```

awk - pattern scanning and processing language

Intro

- awk instructions are

```
<condition> { what to do }
```

- Special conditions

```
BEGIN {}
END {}
```

- Special variables

```
NR
NF
```

- Operators

`>`, `<`, `==`, `=!`, etc. . .

- no `{what to do}` means `{print $0}` or print the line

Examples

```
awk '{print $1}' file
awk -F: '{print $1}' file
awk 'NR == 3' file
awk 'NF > 4  { print $4}' file
awk '$2 == 0 { print $1}' file
awk -f instructions.awk file
```

grep - search a file for a pattern

Examples

```
grep hello file
grep -i hello file
grep -v hello file
egrep 'hello|bye' file
```

tr - translate or delete characters

Examples

```
tr '[a-z]' '[A-Z]' < input > output
tr -d '[^a-z]'  < input > output
```

cut - remove sections from each line of files

Example

- missing

bc - An arbitrary precision calculator language

Examples

bash

```
qty=23.45
blob=$(echo "$qty * 12.4" | bc)
echo $blob
```

csh

```
set qty = 23.45
set blob = `echo "$qty * 12.4" | bc`
echo $blob
```

date - date handling

Example

- subtract one hour or 3600 seconds

```
let x=$(date --date='1/5/2022' +%s)
let x-=3600
then=$(date --date=@$x)

date --date='1/5/2022'
Wed Jan  5 00:00:00 EST 2022

echo $then
Tue Jan  4 23:00:00 EST 2022
```

- change the format with +

```
then=$(date +%y%m%d_%H%M --date=@$x)
echo $then
220104_2300
```

scripting hands-on

In the hands-on portion of the workshop you will learn how to:

- Run a set of commands: Scripts
- Simplify and avoid errors
- Test assumptions
- Generalize a script to be used for multiple files or executions: Arguments
- Ease hands-on time by repeating a command for every file: for loops
- Re-use arguments by manipulating the text
- Get parameters from another file

Log in to Hydra

If you need a reminder about how to log into Hydra and how to change your password, check out our Intro to Hydra tutorial: `https://github.com/SmithsonianWorkshops/Hydra-introduction/`