Bash scripting

Code editors, readability, robust scripts, variables, passing arguments, conditionals, loops

Popular code editors that work in Linux

nano – small text editing utility, default in Ubuntu

vi or ***vim* –** default code editor in Linux, used by Linux admins, steep learning curve

***gedit* –** notepad's brother, but in Linux

Visual Studio Code – very nice multi-language cross-platform code editor https://code.visualstudio.com/

Komodo Edit – another great multi-language editor https://www.activestate.com/products/komodo-edit/

***Sublime text editor* –** nice lightweight code editor https://www.sublimetext.com/download

Brackets – primarily aimed at web developers https://brackets.io/

Geany – fast and small code editor https://geany.org/

Write your first script

- Bash is both command line environment and script interpreter
- You can create a file containing commands and programming language constructs that will be executed by Bash
- Bash is a programming language that works great for automating bioinformatics workflows and Linux administration tasks

Create Hello World! Script

```
#! /bin/bash
# This is our first script

echo “Hello World!”
```

Save it as ***hello_world.sh***

Remember comments start with #, always comment your code!

Write your first script

Give yourselves control over the file with permission to execute

```
$ chmod 755 hello_world.sh
```

Run the script

```
$ ./hello_world.sh
```

#! /bin/bash – every script should start with this line, it points Bash to the executive that will interpret this script. In our case this is Bash itself, but can be any interpreted programming language, line perl or python

This line is also called ***shebang***

```
$ which bash
```
Gives us the location of the interpreter

bin/ directory is in our PATH, so the system knows where to look, if the path to interpreter is wrong or the executable directory in not in our path the script will not work

Modify ***hello_world.sh*** by changing **/bin/bash** to **/usr/bin/bash** and try to run it again

Script readability

Use long options **--threads** as opposed to **-t**

Split long command or pipelines over multiple lines and use indentation, for example instead of:

```
command1 -A -B  -C  <file> | command2 -A | command -C
```

We can use brackets to split the commands over multiple lines:

```
( command1 -A -B  -C  <file> |
    command2 -A |
    command -C )
```

Use backslashes:

```
command1 -A -B  -C  <file> | \
    command2 -A |  \
    command -C
```

Split options over multiple lines:

```
command1 -A \
   -B  \
   -C  <file>
```

Script readability

Various ways to combine commands

&& - command2 runs if command command1 succeeds, etc

```
command1 && command2 && command3
```

|| - if command1 fails, run command2

```
command1 || command2
```

{} – create a block of commands with curly braces with a name and than address it by that name

```
run_block() {
   command1
   command2
   command3
}

run_block # run commands between curly braces
```

; - run commands one after another

```
command1; command2; command3
```

Much more: https://tinyurl.com/y3na77w5

Robust script

Robust script header:

```
#! /bin/bash
set -e
set -u
set -o pipefail
```

Add this header to every script we will be creating!

Let’s take a closer look at this statement

set -e :
By default, if a command in the script fails, the script proceeds as if nothing happened!

We don’t want this behavior

We want the script to fail entirely with explicit message if any command fails!

Robust script

Robust script header:

Let’s test **set -e**
Create the following script called ***test_e.sh***:

```
#! /bin/bash

ls -lh .

echo "first command worked"

ls -lh . | greps 'some_file’ # this command contains an error

echo "the third command did not work"
```

Now add **set -e** to the header and re-run this script
What is the difference?

Robust script

Robust script header:

Let’s take a look at **set -u**

By default, any command containing reference to unset variable will still run

For example: (**!!!DO NOT RUN THIS CODE**):

if the variable $TEMP is not set, and we try to run

```
rm -rf $TEMP/*
```

By default, bash will substitute $TEMP with nothing, resulting in disastrous command that will remove all of file system

```
rm -rf /*
```

set -u will disable this undesirable behavior

Robust script

Robust script header:

Let’s test **set -u**

Create the following script and call it ***test_u.sh***:

```
#! /bin/bash

echo $BAM_FILE
```

Change permissions, and run it

Now add **set -u** to the script header and run it again

What had changed**?**

Robust script

Robust script header:

Let’s take a look at **set -o pipefail**

- **set -e** ensures that the script quits if a command exists with non-zero status (fails)
- However, if the command is a part of a pipeline and it fails, the script will not quit unless this is the last command in the pipeline even with **set -e** set
- **set -o pipefail** instructs bash to quit if any command within a pipeline fails

Create the following script and call it ***test_pipefail.sh*** and run it:

```
#! /bin/bash

ls -lh | awk '{print $9}' | sed 1,1d

echo "First command worked"
```

Robust script

Robust script header:

Let’s take a look at **set -o pipefail** (continued)

Modify the file to make ***awk*** command invalid, for example change it ***awks***:

```
#! /bin/bash

ls -lh | awks '{print $9}' | sed 1,1d

echo "First command worked"
```

Run the script

Add **set -e** to the header and re-run the script

Add **set -o pipefail** and re-run again, how did the behavior changed

Robust script

Robust script header:

From now on will add all three statements to every script we create

Modify **hello_world.sh** to contain the safety lines

```
#! /bin/bash
set -e
set -u
set -o pipefail

echo "Hello World!"
```

Variables and command line arguments

Setting variables

We assign variables with “=“ sign
READS_DIR=“fastq/”

The variable names can be lower case or upper case
reads_dir=“fastq” or **Reads_Dir=“fastq”**

How to address variables? We use $
$READS_DIR

For example
```
$ READS_DIR=“fastq”
$ echo $READS_DIR
```

What happens if we omitted dollar sign?
```
$ echo READS_DIR
```

Variables and command line arguments

Setting variables (continued)

What if we use doble-quotes around the variable name

```
$ echo "$READS_DIR“
```

How does this change with single quotes?

```
$ echo '$READS_DIR’
```

What symbols are allowed in variable names?
We can always check on the command line:

```
$ some_var=10
$ echo $some_var

$ someVar=10
$ echo $someVar

$ 1some_var=10
$ some_var1=10
$ echo $some_var1
```

Variables and command line arguments

Setting variables (continued)

```
$ ?some_var=10
$ some_var?=10
$ some-var=10
```

In general, letters and underscore are safe choices, you can always test your variable name on the command line

We don’t want to use variable names reserved to shell environment variables

```
$ printenv
```

Some variables have special meaning in bash

- $0 – filename of the current script
- $<N> - Nth argument passed to the script on the command line, for example $1, $2, $3, etc
- $# - the number of arguments passed to the script or function
- $? – exit status of the last command executed

https://www.educative.io/answers/what-are-special-variables-in-unix-linux

Variables and command line arguments

Setting variables (continued)

Curly brackets **{}** as place holders

For example:

```
$ sample_name=sample1
$ echo sample_name
```

What if we are trying to create a directory for a specific sample from within the script
Let's say we need a directory names **sample1_reads/**

```
$ mkdir $sample_name_reads
```

This will not work; this is a case for using a place holder

```
$ mkdir ${sample_name}_reads
$ ls –lh
```

It is better to use double-quotes with variables like this:

```
$ mkdir mkdir "${sample_name}_reads"
```

Prevents commands from interpreting any spaces or other special characters in variable names

Variables and command line arguments

Passing arguments to scripts

$0 – stores the name of the script itself

$1 – stores the first argument

$2 – stores the second argument

etc.

```
./myscript.sh arg1 arg2 agr3 … argN
```

Variables and command line arguments

Passing arguments to scripts (continued)

Create the following script and call it ***args.sh***:

```
#! /bin/bash
set -e
set -u
set -o pipefail
# Test arguments script
first_arg=$1 # we will have 3 arguments
second_arg=$2
third_arg=$3
```

```
( echo "the script $0 has the following arguments: $first_arg,
$second_arg, $third_arg" ) # print arguments and the name of the script
```

Run the script

```
$ ./args.sh pig cat dog
$ ./args.sh pig cat # run with missing arguments
```

Conditionals

- Like other programming languages, bash supports ***conditionals***
- Conditional is a statement that prompts action if certain condition is ***true***
- Contrary to other languages, ***true*** is equal to 0, and failure or ***false*** is anything other than 0
- The failure of the command is also called ‘exiting with non-zero status’

Basic syntax:

```
if [commands] # commands can be any command, pipeline, test condition
then
   [if-statements] # statement executed if “commands” evaluates to 0/TRUE
else
   [else-statements] # statement executed if “commands” evaluates to FALSE
fi
```

Conditionals

Let’s create a script cond1.sh to practice conditionals

Download the practice file:

```
$ wget https://raw.githubusercontent.com/slavailn/bioinf_training/main/darkness_byron.txt
```

```
#! /bin/bash
set -e
set -u
set -o pipefail

target_file=$1
search_pattern=$2

if ( grep $search_pattern $target_file )
Then # command to run if grep finds a pattern
        echo "$search_pattern was found in $target_file"
else # command to run if no pattern was found
        echo "$search_pattern not found"
fi
```

Conditionals

Run ***cond1.sh*** script

```
$ ./cond1.sh darkness_byron.txt 'dream’

$ ./cond1.sh darkness_byron.txt ‘drum’
```

Negating conditional with **!**

```
if !( grep $search_pattern $target_file )
Then # command to run if grep finds a pattern
        echo "$search_pattern not found in $target_file"
else # command to run if no pattern was found
        echo "$search_pattern was found"
fi
```

Modify ***cond1.sh*** and run it again

Conditionals

Run modified ***cond1.sh*** script with negation

```
$ ./cond1.sh darkness_byron.txt 'dream’
$ ./cond1.sh darkness_byron.txt ‘drum’
```

Create ***cond2.sh***, read the code and try to understand what it does

```
#! /bin/bash
set -e
set -u
set -o pipefail

target_file=$1
pattern_count=`grep -o 'the' $target_file | wc -l`
echo found $pattern_count occurences of 'the' in $target_file

# Notice different ways we can compare values
#if [ $pattern_count -gt 50 ]
#if (( $pattern_count > 50 ))

if ( test $pattern_count -gt 50 )
then
    sed s/the/cat/g $target_file
fi
```

Conditionals

Run ***cond2.sh***

```
$ ./cond2.sh darkness_byron.txt
```

We can use backticks `` construct to assign the output of shell command to the variable:

```
pattern_count=`grep -o 'the' darkness_byron.txt | wc -l`
echo $pattern_count
```

The same can be done with **$()**

```
pattern_count=$(grep -o 'the' darkness_byron.txt | wc –l)
echo $pattern_count
```

We will be using ***test*** command in conditionals

test exits with either 0 or 1 based on the evaluation of it's arguments

```
$ test "cat" = "cat"; echo "$?" # test string equality
```

```
$ test "Cat" = "cat"; echo "$?"
```

```
$ test 5 -lt 3; echo "$?" # numeric comparison, -lt – less then
```

```
$ test 9 -le 12; echo "$?" # -le – less then or equal
```

Conditionals

String and integer comparison operators for ***test***

String/Integer	Description
-z str	The string is empty (null)
str1 = str2	equal
str1 != str2	not equal
int1 -eq int2	integers equal
int1 -ne int2	Integers not equal
int1 -lt int2	less then
int1 -gt int2	greater then
int1 -le int2	less then or equal
int1 -ge int2	greater then or equal

NOTE: bash supports only integer math

Conditionals

In practice ***test*** is frequently used for file and directory related operations

Examples of file and directory tests

Test if something is a directory

```
$ mkdir test_dir
$ test -d test_dir/; echo "$?"
```

Test if something is a file

```
$ test -f darkness_byron.txt; echo "$?"
```

Test if file is readable

```
$ test -r darkness_byron.txt; echo "$?"
```

Test if the file is executable

```
$ test -x darkness_byron.txt; echo "$?"
```

Conditionals

Directory and file operators for ***test***

File/directory expression	Description
-d <dir>	Check if directory
-f <file>	Check if file
-e <file>	Does the file exist?
-h <link>	Check if link
-r <file>	Check if file is readable
-w <file>	Check if file is writable
-x <file>	Check if file is executable

Globbing and loops

- In bioinformatics projects commands are applied over multiple files
- We can use loops to apply a set of commands to transform multiple files while keeping track of the file extensions

Fastq1 → QC_report1 → Fastq_trimmed1 → Aligned1 → Variant_calls1
Fastq2 → QC_report2 → Fastq_trimmed2 → Aligned2 → Variant_calls2
Fastq3 → QC_report2 → Fastq_trimmed3 → Aligned3 → Variant_calls3

- We will need a way to select a right subset of files when running each step of the pipeline

Download practice files for our mock workflow based on loops and globbing:

```
$ wget https://raw.githubusercontent.com/slavailn/bioinf_training/main/test_R1.fq
```

```
$ wget https://raw.githubusercontent.com/slavailn/bioinf_training/main/test_R2.fq
```

Globbing and loops

Create a script that will subsample fastq files we had just downloaded

The script will create 3 pairs of fastq files with 10,000 reads in each:

1. **s1_R1.fastq** AND **s1_R2**.fastq
2. **s2_R1**.fastq AND **s2_R2**.fastq
3. **s3_R1**.fastq AND **s3_R2**.fastq

We will use **seqtk** to subsample files like this:

```
seqtk sample -s100 <test_R1.fastq> <NUM_READS> > <sample_R1.fastq>

seqtk sample -s100 <test_R2.fastq> <NUM_READS> > <sample_R2.fastq>
```

Globbing and loops

Create a script ***subsample.sh*** that will use seqtk and subsample practice **fastq** files

```
#! /bin/bash
set -e
set -u
set -o pipefail

fastq1=$1 # read 1 file to subsample
fastq2=$2 # read 2 file to subsample
num_reads=$3 # number of reads to subsample

# Iterate over a range of numbers
for i in {1..3}
do
   seqtk sample -s100 $fastq1 $num_reads > s${i}_R1.fastq
   seqtk sample -s100 $fastq2 $num_reads > s${i}_R2.fastq
done
```

Globbing and loops

Create a script called ***convert_to_fasta.sh*** script the will generate summary files for the read pairs we generated with ***subsample.sh*** and convert them to fasta

```
#! /bin/bash
set -e
set -u
set -o pipefail

# Glob over files in the current directory
for file in ./*.fastq
do
   filename=`basename $file`
   sample_name=${filename%.*}
   echo Processing $sample_name
   seqkit stats $file > ${sample_name}.stats
   seqtk seq -a $file > ${sample_name}.fasta
done
```

Globbing and loops

Create a script called ***create_interleaved.sh.***

This script will combine read 1 and read 2 files into a single interleaved file

Download this script first:

```
$ wget
https://gist.githubusercontent.com/nathanhaigh/4544979/raw/1a3f6932e9a02c19faa086457a1a6dea7146ab8f/interleave_fastq.sh
```

Change permissions

```
$ chmod 755 interleave_fastq.sh
```

We will use interleave_fastq.sh internally in our script to create interleaved files

The usage of ***interleave_fastq.sh***

```
Usage: interleave_fastq.sh R1.fastq R2.fastq > interleaved.fastq
```

Globbing and loops

```
#! /bin/bash
set -e
set -u
set -o pipefail

# Populate array using for loop
i=0
sample_array=() # create empty array

for file in *R1.fastq
do
   filename=`basename $file`
   sample_name=${filename%.*}
   sample_name=${sample_name%_*} # remove suffix starting with _
   sample_array[$i]=$sample_name
   i=$i+1
done
echo  ${sample_array[@]} # print array

for sample in ${sample_array[@]}
do
   interleaved=${sample}.interleaved.fq
   ./interleave_fastq.sh ${sample}_R1.fastq ${sample}_R2.fastq > ${sample}_interleaved.fastq
done
```