Bash Scripting

Learning Outcome

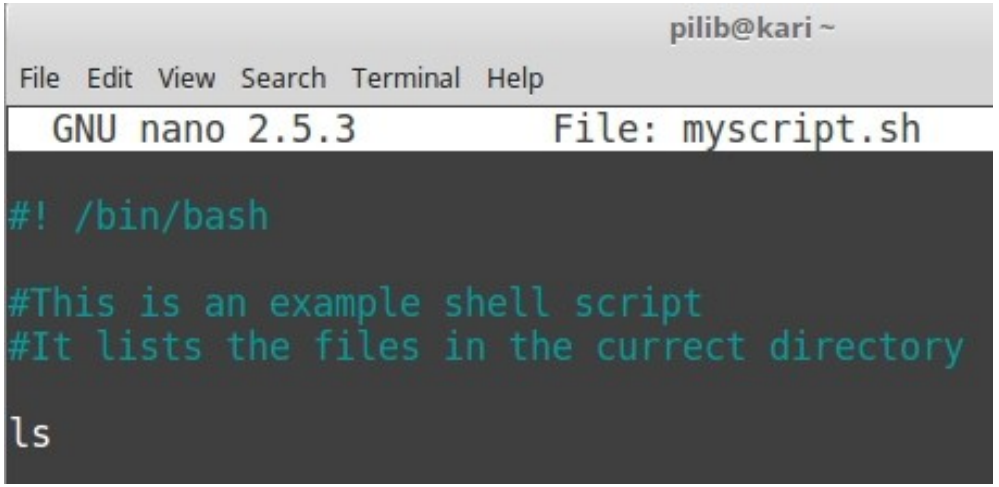
On successful completion of this module, students should be able to:

- **Use Docker, (bio)conda, and git to create reproducible analysis environments and generate reproducible results**
- **Use the Linux command line environment including access/use of a High-Performance Compute (HPC) cluster**
- Write Rmarkdown documents to generate reproducible research reports
- Analyse gene expression microarrays in order to identify differentially expressed genes, enriched GO terms, pathways, and gene sets
- Develop simple Shiny applications

What is a Bash Script?

Simple text file (create with nano, vi, vim, emacs etc.) that contains list of shell commands to be executed

- Starts with **#! /bin/bash**
- Any line starting with a # is a comment and is ignored by the interpreter
- Make executable with **chmod a+x myscript.sh**
- Execute (in cwd) with **./myscript.sh**



Variables

Temporary store for a piece of information

- Assigned values with **= (be careful of spaces, env!)**
- Names can include **letters, numbers** or **underscores** (should not start with number)
- Access values with **$**VARNAME
- **echo** command to print to **stdout (double quotes allow substitution)**

```
#! /bin/bash

#This is an example shell script
#It assigns values to two variables
#and prints them to stdout

SAMPLE_TYPE="HNSCC_Tumour_Recurrence"
SAMPLE_NUM=26

echo $SAMPLE_TYPE
echo $SAMPLE_NUM
```

Variables

Temporary store for a piece of information

- Assigned values with **= (be careful of spaces, env!)**
- Names can include **letters, numbers** or **underscores** (should not start with number).
- Access values with **$**VARNAME
- **echo** command to print to **stdout (double quotes allow substitution)**

```
#! /bin/bash

#This is an example shell script
#It assigns values to two variables
#and prints them to stdout

SAMPLE_TYPE="HNSCC_Tumour_Recurre
SAMPLE_NUM=26

echo $SAMPLE_TYPE
echo $SAMPLE_NUM
```

```
#! /bin/bash

#This is an example shell script
#It assigns values to two variables
#and prints them to stdout

SAMPLE_TYPE="HNSCC_Tumour_Recurrence"
SAMPLE_NUM=26

echo "Processing $SAMPLE_NUM samples of type: $SAMPLE_TYPE"
```

Variables – Command Substitution

We can use built-in shell commands to assign values to variables using **VARNAME=$(command*)***

```
#! /bin/bash

#This is an example shell script
#It assigns values to a variable
#based on command output

MYDIR=$(pwd)
FILES=$(ls *.fq | wc -l)
```

Variables – Command Substitution

We can use built-in shell commands to assign values to variables using **VARNAME=$(command*)***

```
#! /bin/bash

#This is an example shell script
#It assigns values to a variable
#based on command output

MYDIR=$(pwd)
FILES=$(ls *.fq | wc -l)

echo "Working in directory: $MYDIR"
echo "Found $FILES fastq files to process..."
```

```
pilib@kari ~ $ ./myscript2.sh
Working in directory: /home/pilib
Found 12 fastq files to process...
```

Variables – Command Substitution

This can also be useful for finding the location of executables needed for your pipeline:

```
#! /bin/bash

#This is an example shell script
#It assigns values to variables
#based on command output

MYDIR=$(pwd)
FILES=$(ls *.fq | wc -l)
PYPATH=$(which python)
MULTIQC_PATH=$(which multiqc)

echo "Working in directory: $MYDIR"
echo "Found $FILES fastq files to process..."
echo "Processing with: "
$PYPATH --version
$MULTIQC_PATH --version
```

```
pilib@kari ~ $ ./myscript2.sh
Working in directory: /home/pilib
Found 12 fastq files to process...
Processing with:
Python 3.6.3 :: Anaconda custom (64-bit)
multiqc, version 1.1
```

Flow Control – If Statements

What happens if a file we expect to process doesn't exist or a program (or particular version) isn't installed? We can check using i**f statements**.

```
if [ condition1 ]; then
    statement1
fi
```

The code within the **statement block is <u>only</u> executed if the condition / logical expression holds true.**

There are a number of **built-in logical operators** to test **arithmetic** and **string** values as well as the **existence** of and **access** to **files**.

Flow Control – If Statements

Operator	Description
! EXPRESSION	The EXPRESSION is false.
-n STRING	The length of STRING is greater than zero.
-z STRING	The lengh of STRING is zero (ie it is empty).
STRING1 = STRING2	STRING1 is equal to STRING2
STRING1 != STRING2	STRING1 is not equal to STRING2
INTEGER1 -eq INTEGER2	INTEGER1 is numerically equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is numerically greater than INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is numerically less than INTEGER2
-d FILE	FILE exists and is a directory.
-e FILE	FILE exists.
-r FILE	FILE exists and the read permission is granted.
-s FILE	FILE exists and it's size is greater than zero (ie. it is not empty).
-w FILE	FILE exists and the write permission is granted.
-x FILE	FILE exists and the execute permission is granted.

Flow Control – If Statements

```
#! /bin/bash

a=10
b=20

if [ $a -eq $b ]; then
   echo "a is equal to b"
fi

if [ ! $a -eq $b ]; then
   echo "a is not equal to b"
fi
```

```
MYDIR=$(pwd)
FILES=$(ls *.fq | wc -l)
PYPATH=$(which python)

if [ -z $PYPATH ]; then
        echo "Error. Python not found...aborting"
fi
```

Flow Control – If Statements

We can also specify what code to execute if the condition does not hold true by adding an **else** block to the if statement:

```
if [ condition1 ]; then
    statement1
else
    statement2
fi
```

As the conditions are mutually exclusive (can't be both true and false at the same time) **only one set of statements gets executed**, the other branch of the if statement is ignored**.**

Flow Control – If Statements

```
MYDIR=$(pwd)
FILES=$(ls *.fq | wc -l)
PYPATH=$(which python)

if [ -z $PYPATH ]; then
        echo "Error. Python not found...aborting"
else
        echo "Working in directory: $MYDIR"
        echo "Found $FILES fastq files to process..."
        echo "Processing with: "
        $PYPATH --version
fi
```

Flow Control – If Statements

With can test arbitrarily complex conditions by extending the if..else..fi structure with **elif** blocks or by creating **nested statements.**

```
if [ condition1 ]; then
     statement1
elif [ condition2 ]; then
    statement2
elif [ condition3 ]; then
    statement3
else
    statement4
fi
```

```
if [ condition1 ]; then
    if [condition2]; then
        statement1
    else
        statement2
    fi
else
     statement3
fi
```

Flow Control – If Statements

Conditions can also be combined using logical ‘and’ operator **&&** and logical ‘or’ operator **||** if placed within **double square brackets (beware operator precedence)**

```
#! /bin/bash
a=16
b=10
c=4

#Assuming no equal numbers here
if [[ $a -gt $b && $a -gt $c ]]; then
   echo "a is the biggest number"
elif [[ $a -lt $b && $a -lt $c ]]; then
   echo "a is the smallest number"
else
   echo "a is the middle number"
   if [ $b -gt $c ]; then
        echo "b is the biggest, c is the smallest"
   else
        echo "c is the smallest, b is the biggest"
   fi
fi
```

User Input

Sometimes, rather than hardcoding paths/file names, we want to let the user enter this information at run time. One way to do this is with the read command.

read varname

read –p "prompt" varname

read –sp "prompt" varname

```
#! /bin/bash

read -p "Enter your username: " USER
read -sp "Enter your password: " PASS
echo

if [[ $USER = "pilib" && $PASS = "secret" ]]; then
        echo "Welcome $USER"
else
        echo "Incorrect user/password combination"
fi
```

```
pilib@kari ~ $ ./read.sh
Enter your username: pilib
Enter your password:
Welcome pilib
```

User Input

Read can also read in multiple variables on the same line, e.g.

`read var1 var2 var3 ...`

```
#! /bin/bash

echo "Enter reference genome, read file1, read file2: "
read REF PAIR1 PAIR2

if [ ! -r $REF ]; then
        echo "Genome file is not readable..."
        exit 1
fi

if [ ! -r $PAIR1 ]; then
        echo "Pair file 1 is not readable..."
        exit 1
fi

if [ ! -r $PAIR2 ]; then
        echo "Pair file 2 is not readable..."
        exit 1
fi
```

Exercise 1

Command Line Arguments

Users can also pass **arguments** to the script when executing it. These are **automatically** assigned to **positional variables $1 to $9** (the **name of the script** is **$0).** There are also a number of other **special shell variables:**

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$@	All positional parameters, “$@” is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

Arrays - Basics

Arrays are a collection of variables. Unlike other languages, bash doesn't require arrays to be of a specific type. Arrays are declared with **declare -a** and **initialised** with **parentheses:**

```
declare -a ARR
ARR=() #empty array
ARR=(13 “a” 1)
```

Access individual array elements using **curly braces and square brackets** with **index.** Using @ instead of a specific index **lists all** of the array values**.**

Multiple elements within a **range (array slice)** can be accessed by adding start and end indices after ARR[@]. Note that arrays are **zero-based**

```
echo “First item: ${ARR[0]}”
echo “All items: ${ARR[@]}”
echo “Items 2 & 3: ${ARR[@]:1:2}
```

Arrays – Modifying, Deleting, and Appending

Individual array elements can be **modified** using square brackets**:**

```
ARR[2]=8
```

To **remove** items from an array, use **unset**

```
unset ARR[2]
```

Individual or multiple elements can be **appended** to an array using **+=**
${#ARR[@]} will give current number of elements **(array size)**

```
echo “Array size is: ${#ARR[@]}
ARR+=(2 “more” “elements”)
echo “Array size now: ${#ARR[@]}
```

Arrays - Example

```
#! /bin/bash

declare -a ARR
ARR=(12 "bwt" "mm9.fa" "reads.fa")

echo ${ARR[@]}
echo "There are ${#ARR[@]} elements in this array"
echo "The first element is: ${ARR[0]}"
echo "The Second and third elements are: ${ARR[@]:1:2}"

echo "Modifying the first element..."
ARR[0]=8
echo "The first element is now ${ARR[0]}"
echo ${ARR[@]}

echo "Appending multiple values to the array..."
ARR+=("these" "values" "are" "being"  "appended")
echo ${ARR[@]}

echo "Removing item 3 from the array..."
unset ARR[2]
echo ${ARR[@]}
```

For Loops - Basics

For loops can be used to iterate over a series of **elements in a sequence or array**, **words in a string, files in a list**. The general syntax is:

```
for variable in argument-list
  do
       commands
  done
```

```
for i in `seq 1 10`
do
        echo $i
done
```

```
for i in 1 3 7 12 14
do
        echo $i
done
```

```
for i in $@
do
        echo $i
done
```

```
read -p "Please enter a string: " mystr

for word in $mystr
do
        echo $word
done
```

```
for file in *.fq
do
        fastqc $file
done
```

For Loops – Alternative Loop Control

Loop iterations can also be specified using an initial value for the loop-control variable, test expression, and increment (or modifying) value:

```
for (( init_value; condition; modifier ));
do
        commands
done
```

```
#! /bin/bash

array=("mm9" "bowtie2" 8 "DEseq2")
arraylength=${#array[@]}

for (( i=0; i<${arraylength}; i++ ));
do
    echo "Param $(( $i + 1 )) :  ${array[$i]}"
done
```

While and Until Loops

Two other loop forms that execute based on a logical test:

```
while [ expression ]
  do
       command-list
  done
```

```
until [ expression ]
  do
       command-list
  done
```

```
sum=0
counter=1
while [ $counter -le 10 ]
do
    echo "Counter is: $counter"
    sum=$((sum + counter))
    echo "New sum is: $sum"
    ((counter++))
done
```

```
sum=0
counter=1
until [ $counter -gt 10 ]
do
    echo "Counter is: $counter"
    sum=$((sum + counter))
    echo "New sum is: $sum"
    ((counter++))
done
```

Exercises 2 & 3

Functions

Useful to define your own functions for commonly used blocks of code:

```
function-name () {
   statements
}
```

```
#! /bin/bash

testfile() {
  if [ $# -gt 0 ]; then
     if [[ -f $1 && -r $1 ]]; then
        echo "$1 is a readable file"
     else
        echo "$1 is not a readable file"
     fi
  fi
}
```

String Manipulation

and ## strip from front of variable

- **${var#Pattern}** Removes shortest pattern match
- **${var##Pattern}** Remove longest pattern match

What will the following code produce?

```
file=/home/pilib/data/VEC45_R1_fastq.tar.gz
echo ${file#/*/}
echo ${file##/*/}
```

String Manipulation

and ## strip from front of variable

- **${var#Pattern}** Removes shortest pattern match
- **${var##Pattern}** Remove longest pattern match

What will the following code produce?

```
file=/home/pilib/data/VEC45_R1_fastq.tar.gz
echo ${file#/*/}
echo ${file##/*/}
```

```
pilib/data/VEC45_R1_fastq.tar.gz
VEC45_R1_fastq.tar.gz
```

String Manipulation

% and **%%** strip from end of variable

- **${var%Pattern}** Removes shortest pattern match
- **${var%%Pattern}** Remove longest pattern match

What will the following code produce?

```
file=/home/pilib/data/VEC45_R1_fastq.tar.gz
echo ${file%.*}
echo ${file%%.*}
```

String Manipulation

% and **%%** strip from end of variable

- **${var%Pattern}** Removes shortest pattern match
- **${var%%Pattern}** Remove longest pattern match

What will the following code produce?

```
file=/home/pilib/data/VEC45_R1_fastq.tar.gz
echo ${file%.*}
echo ${file%%.*}
```

```
/home/pilib/data/VEC45_R1_fastq.tar
/home/pilib/data/VEC45_R1_fastq
```

String Manipulation

% and **%%** strip from end of variable

- Fast rename of all files

```
#!/bin/bash

for p in /data/project1/scripts/perl/*.perl
do
        mv "$p" "${p%.perl}.pl"
done
```

String Manipulation

Pattern replacement **/** and **//**

- **${var/pattern/string}** Replace first occurrence only
- **${var//pattern/string}** Replace all occurrences

```
## list of samples
## (only paired reads, must follow _1.*/_2.* file naming convention)
reads1=(${FASTQLOC}/*_1.*)
reads1=("${reads1[@]##*/}")
reads2=("${reads1[@]/_1./_2.}")
```

Regular Expressions - Metacharacters

Can use with **grep, sed, awk, or expr** commands also **=~** operator

Operator	Effect
.	Matches any single character.
?	The preceding item is optional and will be matched, at most, once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{N}	The preceding item is matched exactly N times.
{N,}	The preceding item is matched N or more times.
{N,M}	The preceding item is matched at least N times, but not more than M times.
-	represents the range if it's not first or last in a list or the ending point of a range in a list.
^	Matches the empty string at the beginning of a line; also represents the characters not in the range of a list.
$	Matches the empty string at the end of a line.
\b	Matches the empty string at the edge of a word.
\B	Matches the empty string provided it's not at the edge of a word.
\<	Match the empty string at the beginning of word.
\>	Match the empty string at the end of word.

Regular Expressions – POSIX Classes

POSIX class	similar to	meaning
[:upper:]	[A-Z]	uppercase letters
[:lower:]	[a-z]	lowercase letters
[:alpha:]	[A-Za-z]	upper- and lowercase letters
[:digit:]	[0-9]	digits
[:xdigit:]	[0-9A-Fa-f]	hexadecimal digits
[:alnum:]	[A-Za-z0-9]	digits, upper- and lowercase letters
[:punct:]		punctuation (all graphic characters except letters and digits)
[:blank:]	[\t]	space and TAB characters only
[:space:]	[\t\n\r\f\v]	blank (whitespace) characters
[:cntrl:]		control characters

Regular Expressions – Examples

```
#! /bin/bash

read -p "Please enter a positive integer:" mystr

if [[ $mystr =~ ^[0-9]+$ ]]; then
    echo "You entered a valid number"
else
    echo "You entered an invalid number"
fi
```