Handout: Bash

Table of Contents

Welcome to GenEpi-BioTrain Training in Genomic Epidemiology and Public Health Bioinformatics "Bridging the Gap" - Third edition (TB and AMR)! This course will equip you with the basic skills needed to navigate Unix environments, automate tasks, and harness its capabilities to advance your understanding and practice in bioinformatics. This second segment will be introducing bash and focusing on its automation prospects.

Bash

Bash (Bourne Again Shell) is **a command-line interpreter that extends Unix by using its powerful commands** for automation, data processing, and workflow management. Widely used in bioinformatics, Bash enables researchers to manipulate files**, construct pipelines**, and integrate Unix tools efficiently.

Creating a bash script

To initiate a simple script, launch any text editor, such as nano or vim. For the purpose of these exercises, we will opt for nano. Here is an example of how to create and open a script called scriptname.sh:

```
nano scriptname.sh                                                    Unix
```

This action creates and/or opens a text document, allowing you to write content similarly to any other document. The ".sh" extension denotes that the file contains a script written in the Bash programming language.

In the very bottom of the nano text editor, you will see a section of key options, including saving changes (Ctrl + O), exiting (Ctrl + X), etc. To perform one of these options, press Ctrl and the letter associated with the option. You can save the script when exiting.

```
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Non-executable comments

Comments are essential for script documentation. They start with a hashtag (#) and are non-executable. Everything after # will be a comment:

```
# This entire line is a comment and cannot be executed                Bash
```

Shebangs

The shebang (#!/bin/bash) at the beginning of a script informs the system to use Bash as the interpreter. **This is crucial for script execution**. Here is an example:

```
#!/bin/bash                                                           Bash
```

Printing

Use the **echo** command to print messages to the terminal. Here is an example:

```bash
#!/bin/bash

echo Hello World
echo 'Hello World'
echo "Hello World"
```

This script will print Hello World to the terminal three times. It is worth noting that the sentence can be enclosed using single quotes ('), double quotes ("), and even without using quotes. Nevertheless, using quotations is advisable.

Running bash scripts

To run a bash script, you simply add **bash** in front of the script name in the terminal. Here is an example:

```unix
bash scriptname.sh
```

To make the script executable, run the following command in the terminal (further elaborated in the permission section):

```unix
chmod 755 scriptname.sh
```

Now, to run it, you simply type **./** in front of the script in the terminal, assuming you are in the same directory as the script:

```unix
./scriptname.sh
```

If the script is not located in your current directory, you can specify the path:

```unix
./path/to/scriptname.sh
```

Permissions

File permissions determine who can read, write, and execute a file. Here is a brief overview:

- **Read** (r): Users with read permission can view the contents of the file and list directory contents if the file is a directory.
- **Write** (w): Users with write permission can modify the file's contents, delete it, or rename it. For directories, write permission allows users to create, delete, and rename files within the directory.
- **Execute** (x): Users with execute permission can execute the file as a program or script. For directories, execute permission allows users to access the directory's contents and navigate through it.

File permissions are typically represented by three sets of three characters:

- The first set indicates permissions for the file owner.
- The second set indicates permissions for the group that the file belongs to.
- The third set indicates permissions for other users not in the file's group.

For example, the permission string rwxrw-r-- means:

- The file owner has read, write, and execution permission.
- The group has read and write permission.
- Other users have read permission.

File permissions can be viewed and modified using the ls -l command to display permissions and the chmod command to change them. Here is a list of mode parameters used for modifying file permissions:

#	Permission
0	None
1	Execute only
2	Write only
3	Write and execute
4	Read only
5	Read and execute
6	Read and write
7	Read, write, and execute

To modify permissions, use the **chmod** command along with the modes from the table above. For example, in order to change permissions of a file so it matches the example above (rwxrw-r--), you can write the following in the terminal:

```
chmod 764 scriptname.sh                                    Unix
```

Variables

Variables are used to store data such as numbers, strings, directories, and arrays. These can be used for later reference or manipulation within a bash script. They are flexible and can be assigned, reassigned, and used in various contexts. You must assign the variable before using it; otherwise, the interpreter does not know what to do with the variable. Some key points about variables in Bash:

- **Declaration**: Variables are declared by assigning a value to them, without specifying a data type.
- **Naming Convention**: Variable names are **case-sensitive** and can consist of letters, numbers, and underscores but cannot start with a number.
- **Accessing Variables**: To access the value stored in a variable, prepend the variable name with a dollar sign ($).

- **Assigning Values**: Values are assigned to variables using the operator (=). No spaces are allowed around the operator when assigning values.
- **Quoting**: It is good practice to quote variables containing white space.

Here is an example of a bash script with assigned variables:

```bash
#!/bin/bash

pet="polar bear"
edu=astrophysicist
wish=baker
col=blue

echo "Mia wants another $pet"
echo "Mia studied to be an $edu but really wanted to be a $wish"
```

The output when running the script above:

```unix
./scriptname.sh
output:  Mia wants another polar bear
         Mia studied to be an astrophysicist but really wanted to be a baker
```

Notice how the variable blue is absent from the output.

Positional arguments

Positional arguments refer to the parameters passed to a script when it is executed. These arguments are accessible within the script using special variables known as positional parameters. These are represented by: $1, $2, $3, ..., where $1 represents the first argument, $2 the second, and so on. For example, the following bash script:

```bash
#!/bin/bash

echo "I love $1, but only on $2"
echo "All arguments passed: $@"
```

Would have the following output:

```unix
./scriptname.sh vegemite Tuesdays
output:  I love vegemite, but only on Tuesdays
         All arguments passed: vegemite Tuesdays
```

Word count

In Unix, **wc** stands for "word count". It is a command-line utility used to count the number of lines, words, and characters in a file or standard input stream. When used without any options or arguments, **wc** counts the number of lines, words, and characters in the specified file(s). For example, to count the number of lines, words, and characters in the script created above (scriptname.sh), you can use the command:

```
wc scriptname.sh                                                    Bash
output  4    15    100 scriptname.sh


wc -l scriptname.sh
output  4    scriptname.sh
```

Relational operators

Relational operators are used to compare values or expressions and determine their relationship. These operators help in making decisions within scripts based on the comparison results. Here is a list of relational operators:

Operator	Explanation
-eq	is equal to
-ne	is not equal to
-gt	is greater than
-ge	is greater than or equal to
-lt	is less than
-le	is less than or equal to

If statements

An **if statement** in Bash allows commands to run only when a specific condition is true. It evaluates a condition and, if met, executes the corresponding commands. If the condition is false, the commands are skipped. Here is a brief explanation of how it works:

- **Syntax**: The basic syntax of an if statement in Bash is as follows:

```
if [ <condition> ]; then                                            bash
        # commands to execute if the condition is true
        <statement>
fi
```

- **Condition**: The <condition> is an expression that determines whether a statement is true or false. It is typically enclosed within square brackets [ ].
- **Commands**: The commands to execute (<statement>) if the condition evaluates to **true** are placed between the **then** keyword and the **fi** keyword. These commands can be simple or complex, including multiple lines of code.

Here is an example:

```bash
#!/bin/bash
if [ 2 -lt 4 ]; then
        echo "2 is indeed less than 4"
fi

if [ $1 -lt 12 ]; then
        echo "We have less than 12 days until summer! "
fi

cake=1
if [ $cake -eq 1 ]; then
        echo "there is only one cake left :( "
fi
```

Note: tab is used to set the indentation before the <statement(s)>
Note: space is used to add whitespace after the if statement as well as between the brackets and the <condition>

The output when executing the script above:

```unix
./script.sh 11
output:  2 is indeed less than 4
                We have less than 12 days until summer!
                There is only one cake left :(

./script.sh 20
output:  2 is indeed less than 4
                There is only one cake left :(
```

Note: Running the script with an input of 20 produces only two outputs. This happens because the if condition (20 < 12) is false, so the print command inside the if block does not execute.

Elif and else statements

You can extend an if statement using **elif** (short for "else if") and **else** to handle multiple conditions and define a fallback action if none are met. Unlike **if**, you can include multiple **elif** statements in sequence, but only one else. The basic syntax is:

```bash
if [ <condition1> ]; then
        # commands to execute if condition1 is true
        <statement1>
elif [ <condition2> ]; then
        # commands to execute if condition1 is false and condition2 is true
        <statement2>
else
        # commands to execute if none of the conditions are true
        <statement3>
fi
```

Here is an example:

```bash
#!/bin/bash

if [ "$1" -gt "$2" ]; then
   echo "$1 is greater than $2"
elif [ "$1" -lt "$2" ]; then
   echo "$1 is less than $2"
else
   echo "$1 is equal to $2"
fi
```

The output when executing the script above:

```unix
./script.sh 5 3
output:  5 is greater than 3

./script.sh 3 5
output:  3 is less than 5

./script.sh 5 5
output:  5 is equal to 5
```

For loops

A **for loop** is used to iterate over a list of items and perform a set of commands for each item in the list. Here is a brief explanation of how for loops work:

- **Syntax**: The basic syntax of a *for loop* is as follows:

```bash
#!/bin/bash

for <item> in <iterable>; do
        # commands to execute for each <item>
        <statement>
done
```

- **iterable**: The <iterable> is a collection of items separated by spaces (list). These items can be strings, numbers, file names, or any other type of data.
- **Item**: The <item> is a variable that represents each element in the <iterable> during each iteration of the loop. You can use this variable to reference the current item being processed.
- **Statement**: The <statement(s)> to execute for each item in the iterable are placed between the **do** and **done** keywords. These commands can be simple or complex, including multiple lines of code.

Here is an example:

```bash
#!/bin/bash

for animal in bear hippo snake werewolf; do
   echo "The $animal is a cutiepie!"
done
```

The output when executing the script above:

```Unix
./script
output:  The bear is a cutiepie!
         The hippo is a cutiepie!
         The snake is a cutiepie!
         The werewolf is a cutiepie!
```

Note: It does not matter what you call the variable <item>. The result would be the same if you called it *cat* instead of *fruit*, as long as you changed the name inside the for-loop as well. Here is another example that would result in the same output:

```bash
#!/bin/bash

for fruit in bear hippo snake werewolf; do
   echo "The $fruit is a cutiepie"
done
```

Looping over files

Looping over files in Bash involves iterating through a list of files or directories and performing actions on each. You can use wildcards like * to match multiple files or specify patterns (e.g., *.txt for text files) to filter specific types. This allows you to automate tasks such as renaming, moving, or processing multiple files efficiently:

```Bash
#!/bin/bash

for file in /path/to/dir/*; do
        echo "Processing file: $file"
done
```

Looping over arrays

An array is a data structure that stores multiple values under a single variable name. In Bash, arrays are defined using parentheses () and accessed using their index number. You can iterate over all elements of an array using a for loop. For example:

```Bash
#!/bin/bash

# Declare array
cutePetArray=("bear" "hippo" "snake" "werewolf")

for cutiepie in ${cutePetArray[@]}; do
        echo "The $cutiepie is a cutiepie! "
done
```

Note: Using a for loop over an array is the same as usual, with an added '@' to ensure it loops over all things in the array, not just the first string.

Output when running the script above:

```
./script                                                    Unix
output:  The bear is a cutiepie!
         The hippo is a cutiepie!
         The snake is a cutiepie!
         The werewolf is a cutiepie!
```

you can access individual elements of an array using their index number, with zero-based indexing (i.e., the first element is at index 0, the second at 1, etc.):

```
#!/bin/bash                                                 Bash

# Declare array
cutePetArray=("bear" "hippo" "snake" "werewolf")

echo "The ${cutePetArray [2]} is a cutiepie!"
echo "The ${cutePetArray [0]} is a cutiepie!"
```

Note: Curly braces {} are required when accessing an array element, or Bash will not interpret it correctly.

Output when running the script above:

```
./script                                                    Unix
output:  The snake is a cutiepie!
         The bear is a cutiepie!
```