

Module 3

Introduction to Computer Programming

Introduction

Recent advances in high-throughput technology have transformed modern biology into an area overflowing with large datasets. These datasets are pushing the limits of desktop applications, and we are now finding that using an Excel spreadsheet is simply not enough. In addition, we often find ourselves in situations where we have to repeat the same task or procedure for every item in our dataset. This repetitive process can often be tedious and time consuming. By learning basic programming we can write small programs, called scripts that allow us to easily manage these large datasets and to automate repetitive tasks. Learning to program can be a daunting task, but it is also extremely worthwhile. Not only will you improve your research, you will also learn new concepts and have a lot of fun!

Aims

The aim of this module is to present you with an introduction to programming, guiding you through useful Linux commands and essential programming concepts. The first part of this module introduces you to more advanced Linux concepts, illustrating these with meaningful examples and exercises. The second part of the module introduces the notion of shell scripting and demonstrates how to save useful Linux commands for future use so that they can be used over and over again.

Many of the examples and exercises throughout this module are designed with sequence manipulation tasks in mind, and at the end of it we hope you will be able to write some small scripts to help you with your research. Like all modules, 'if you don't understand, please ask'. No-one is going to become a programming expert in a few hours; the overall purpose of the module is to give you a taste of what writing small scripts can do to automate and accelerate your analysis.

If at the end of this module you would like to learn more about programming, we have provided a list of useful resources for further reading.

Advanced Linux

Increasingly, the output of biological research exists as *in silico* data, usually in the form of large text files. Linux is particularly suitable for working with such files as it has several powerful and flexible commands that can be used to process and analyse these data. One advantage of learning how to use Linux is that many commands can be combined in an almost unlimited fashion. So, if you can learn just six Linux commands, you will be able to do a lot more than just six things.

You have already been introduced to some basic Linux commands including `ls`, `pwd` and `cp`. Linux contains hundreds of commands, but, to conduct your analysis, you will probably only need around 10 to achieve most of what you want to do. In the following exercises we will introduce you to more Linux commands and provide examples of how they can be used in bioinformatic analyses.

Exercises : More Linux commands

To begin: open a terminal window, move into the `Module_3_Programming` directory, then the `Linux` directory (`cd Module_3_Programming/Linux`) and follow the instructions below.

BED files

We will be using a BED file in the examples that follow. A BED file (.bed) is a tab-delimited text file that defines a set of features. To see the format of a BED file you can view it by running:

```
cd Module_3_Programming/Linux
```

```
less Pfalciparum.bed
```

BED lines have three required fields and nine additional optional fields. The first three required BED fields are:

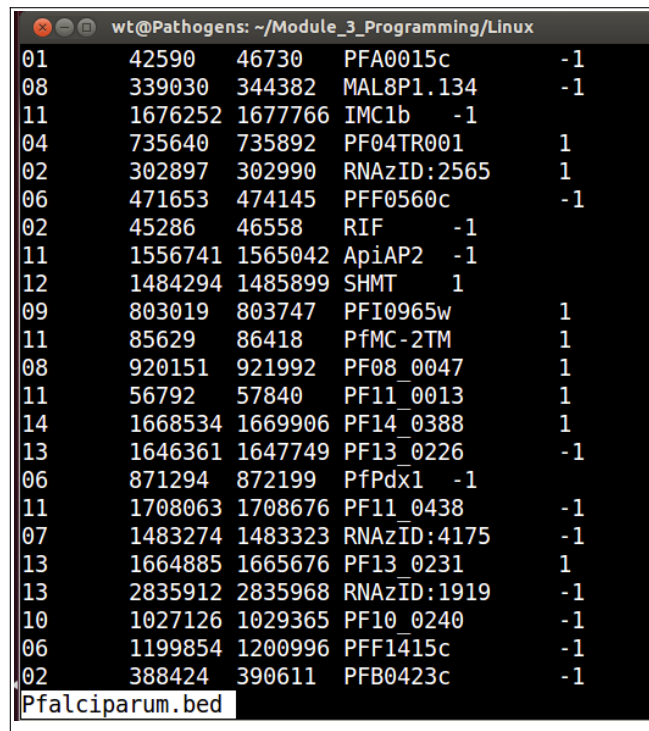
- **chrom** - The name of the chromosome or scaffold
- **chromStart** - The starting position of the feature in the chromosome or scaffold. The first base in a chromosome is numbered 0.
- **chromEnd** - The ending position of the feature in the chromosome or scaffold.

Other additional optional BED fields include name, score, and strand. For more information on BED files see: <http://genome.ucsc.edu/FAQ/FAQformat.html#format1>

Getting help man

To obtain further information on any of the Linux commands listed below you can use the `man` command. For example, to get a full description and examples of how to use the `sort` command type the following in a terminal window.

```
man sort
```



Chromosome	Start	End	Gene Name	Strand
01	42590	46730	PFA0015c	-1
08	339030	344382	MAL8P1.134	-1
11	1676252	1677766	IMC1b	-1
04	735640	735892	PF04TR001	1
02	302897	302990	RNAzID:2565	1
06	471653	474145	PFF0560c	-1
02	45286	46558	RIF	-1
11	1556741	1565042	ApiAP2	-1
12	1484294	1485899	SHMT	1
09	803019	803747	PFI0965w	1
11	85629	86418	PfMC-2TM	1
08	920151	921992	PF08_0047	1
11	56792	57840	PF11_0013	1
14	1668534	1669906	PF14_0388	1
13	1646361	1647749	PF13_0226	-1
06	871294	872199	PfPdx1	-1
11	1708063	1708676	PF11_0438	-1
07	1483274	1483323	RNAzID:4175	-1
13	1664885	1665676	PF13_0231	1
13	2835912	2835968	RNAzID:1919	-1
10	1027126	1029365	PF10_0240	-1
06	1199854	1200996	PFF1415c	-1
02	388424	390611	PFB0423c	-1
Pfalciparum.bed				

Figure 1 The first part of the Pfalciparum.bed file

sort - Sort values in a file or piped input

This command lets you sort the contents of the input. When you sort the input, lines with identical content end up next to each other in the output, which can then be fed to `uniq` (see below) to count the number of unique lines in the input.

To sort the contents of the BED file type the following command:

```
sort Pfalciparum.bed [ENTER]
```

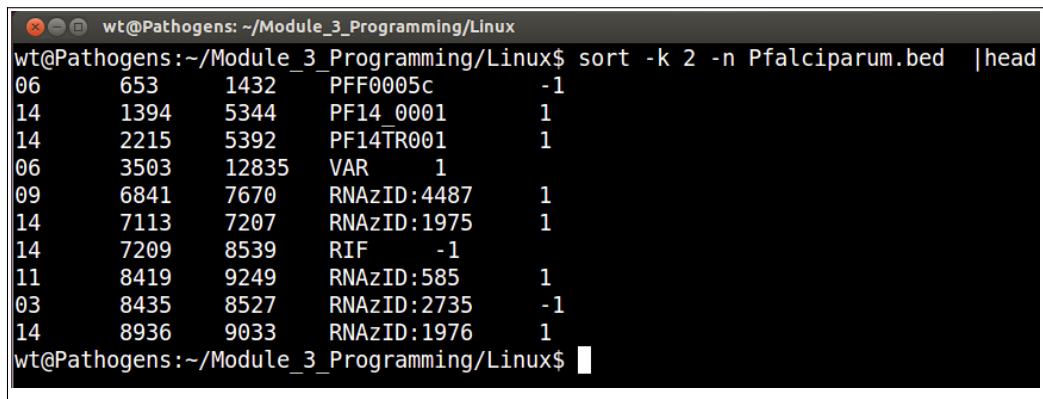
To sort the contents of the BED file on position type the following command:

```
sort -k 2 -n Pfalciparum.bed [ENTER]
```

The sort command can sort by multiple columns e.g. 1st column and then 2nd column by specifying successive `-k` parameters in the command. Modify the previous command to sort the BED file on chromosome and then gene position.

What does the `-n` option do?

Hint: use the command `man sort`.



```

wt@Pathogens: ~/Module_3_Programming/Linux
wt@Pathogens:~/Module_3_Programming/Linux$ sort -k 2 -n Pfalciparum.bed | head
06      653      1432    PFF0005c      -1
14     1394     5344    PF14_0001       1
14     2215     5392    PF14TR001       1
06     3503     12835    VAR           1
09     6841     7670    RNAzID:4487     1
14     7113     7207    RNAzID:1975     1
14     7209     8539    RIF           -1
11     8419     9249    RNAzID:585      1
03     8435     8527    RNAzID:2735    -1
14     8936     9033    RNAzID:1976     1
wt@Pathogens:~/Module_3_Programming/Linux$

```

Figure 2 Sorting on a column. Since there is a lot of output head is used to return the 1st 10 lines.

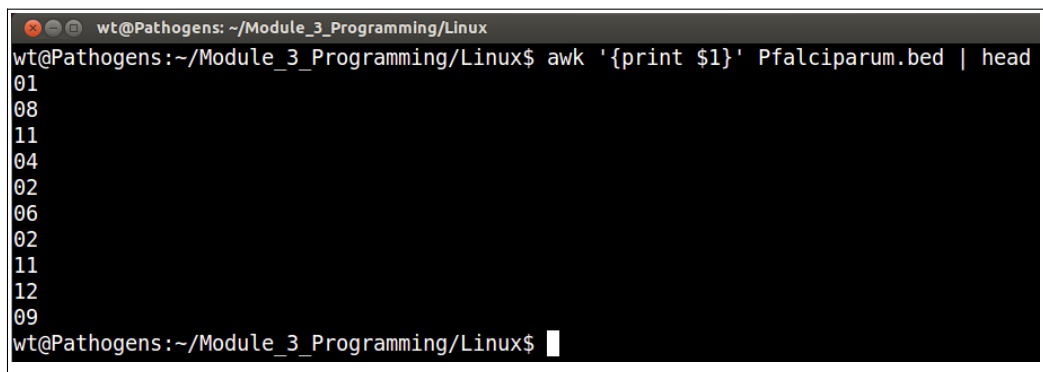
awk - Pattern scanning and processing language

This command lets you manipulate the input text, making it very useful for chopping out the bits of a file that your interested in and outputting them to another file or command.

To print out the first column of the BED file, enter the following command:

```
awk '{print $1}' Pfalciparum.bed [ENTER]
```

This is a very powerful and complex command, and is often used in conjunction with sed which will be talked about later.



```

wt@Pathogens: ~/Module_3_Programming/Linux
wt@Pathogens:~/Module_3_Programming/Linux$ awk '{print $1}' Pfalciparum.bed | head
01
08
11
04
02
06
02
11
12
09
wt@Pathogens:~/Module_3_Programming/Linux$

```

Figure 3 Extracting the first column with awk. Since there is a lot of output head is used to return the 1st 10 lines.

uniq - extract unique lines from a file or piped input

The **uniq** command is usually used in combination with **sort** to count unique values in the input.

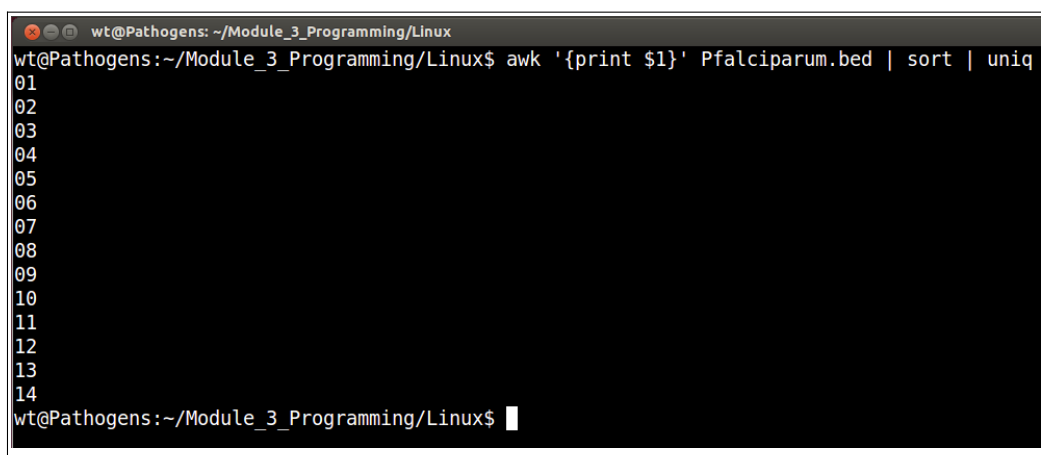
To get the list of chromosomes in the BED file type the following command.

```
awk '{print $1}' Pfalciparum.bed | sort | uniq [ENTER]
```

How many chromosomes are there?

Now modify the previous command to count the number of features per chromosome.

Hint: use the **man** command to look at what are the options for the **uniq** command.



```
wt@Pathogens: ~/Module_3_Programming/Linux
wt@Pathogens:~/Module_3_Programming/Linux$ awk '{print $1}' Pfalciparum.bed | sort | uniq
01
02
03
04
05
06
07
08
09
10
11
12
13
14
wt@Pathogens:~/Module_3_Programming/Linux$
```

Figure 4 Counting the chromosomes in the BED file

find - Finds files matching an expression

The `find` command will search the directory and all sub directories and return a list of files. It can filter the files for you if you tell it the name of the file your searching for. A `*` denotes a wildcard which can stand for any character(s).

To find all files in the current directory and all sub directories:

```
find . [ENTER]
```

To find all fastq files in the current directory and all sub directories:

```
find . -name "*.fastq" [ENTER]
```

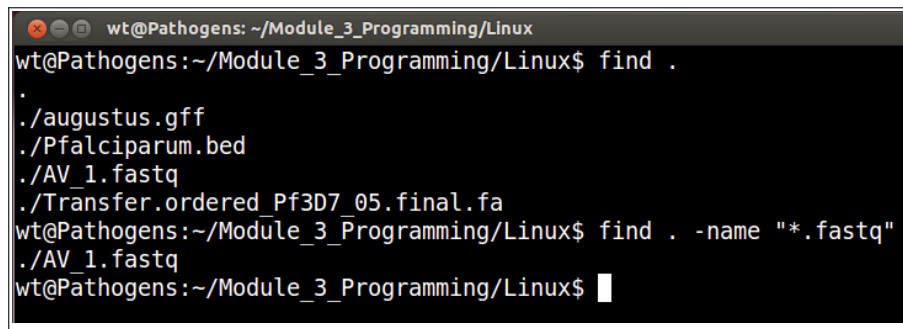
Modify the command above to find all bed files in the current directory.

Use the `find` command to find all fastq files in the `Module_3_Programming` directory.

The `find` command is very powerful. It may be used to execute other Unix commands on the files it finds using the command line option `-exec`. Type this command:

```
find . -name "*.fastq" -exec cp {} .. \; [ENTER]
```

Now look at the content of the parent directory for fastq files.

A terminal window with a black background and white text. The title bar shows 'wt@Pathogens: ~/Module_3_Programming/Linux'. The first command is 'find .' which lists several files: './augustus.gff', './Pfalciiparum.bed', './AV_1.fastq', and './Transfer.ordered_Pf3D7_05.final.fa'. The second command is 'find . -name "*.fastq"' which returns './AV_1.fastq'. The prompt is at the end of the second command line.

```
wt@Pathogens: ~/Module_3_Programming/Linux
wt@Pathogens:~/Module_3_Programming/Linux$ find .
.
./augustus.gff
./Pfalciiparum.bed
./AV_1.fastq
./Transfer.ordered_Pf3D7_05.final.fa
wt@Pathogens:~/Module_3_Programming/Linux$ find . -name "*.fastq"
./AV_1.fastq
wt@Pathogens:~/Module_3_Programming/Linux$
```

Figure 5 Finding files in the current directory

Introduction to Regular Expressions. A regular expression, often called a pattern, is an expression that matches strings of text, such as particular characters, words, or patterns of characters. Common abbreviations for "regular expression" include `regex` and `regexp`.

Matching operators:

`.` match any character

`[]` match one character found within the brackets

Positional flags:

`^` match only at beginning of line. **{\bf NOTE:}** inside square brackets this means match anything except the characters

`$` match only at end of line

Quantifiers:

`+` one or more times

`*` zero or more times

`?` zero or one time

Here are some examples:

`colou?r` matches both "color" and "colour".

`ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.

`ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".

`[abc]` matches either "a" or "b" or "c".

`[^abc]` matches anything but "a" or "b" or "c".

Additional reading: http://en.wikipedia.org/wiki/Regular_expression

grep - For searching text files for character strings and regular expressions

grep is a command line tool which searches the content of text files for regular expressions and text strings. At its most simple, grep can be used to search for a simple string of characters:

```
grep PF11 Pfalciparum.bed [ENTER]
```

grep has many command line options. Some examples include **-c** which counts number of matches, **-i** which ignores case of the matched string and **-v** which inverts the match, returning lines which don't include the matched string.

First modify the above command to count number of matches to PF11. Next, modify it to find all lines not containing the text PF11.

As mentioned, one of the most powerful features of grep is its use of regular expression operators, flags and quantifiers.

Using the information found in **Introduction to Regular Expressions**, search for features in the bed file on chromosomes 10, 11 and 12 only.

```

wt@Pathogens: ~/Module_3_Programming/Linux
11 1522356 1524343 PF11_0397 -1
12 1646601 1647140 PFL1910c -1
11 1248308 1250230 PF11_0331 -1
12 725834 729830 PFL0895c -1
12 1864218 1866650 PFL2135c -1
10 1040287 1045002 PF10_0243 -1
    42992 43250 RNAzID:597 1
    2211527 2212538 stevor 1
10 994247 996430 PFL1175W 1
12 787744 795219 PF10_0188 -1
12 26321 27687 RIF -1
11 329867 335601 Ap1AP2 -1
11 1569601 1573186 PF11TR008 1
10 63153 64033 PFJ13 1
11 1449687 1451447 PF11_0380 -1
11 1446486 1448502 PF11_0379 1
10 1378756 1379292 PF10_0337 1
11 1681865 1690081 PF11_0433 -1
11 1063668 1064549 PF11_0284 -1
10 1030042 1031532 PF10_0241 1
11 1598547 1599629 PF11_0412 -1
11 1323041 1324312 PFPDI-11 -1
wt@Pathogens:~/Module_3_Programming/Linux$ grep -c ^1[012] Pfalciparum.bed
1626
wt@Pathogens:~/Module_3_Programming/Linux$

```

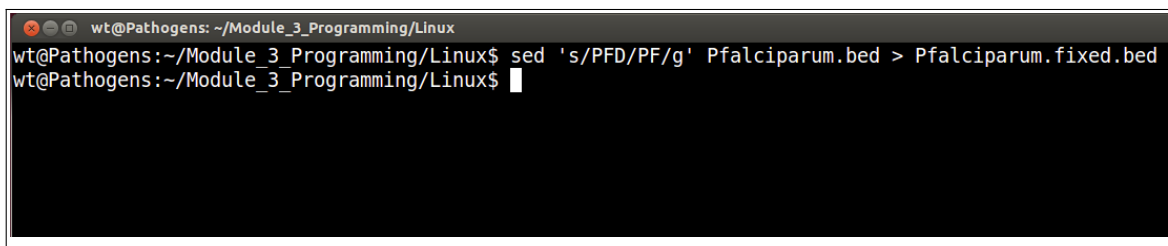
Figure 6 Final part of last grep exercise followed by count of lines found

sed - For filtering and transforming text

sed is a powerful command line tool for editing the contents of a file and outputting the modified contents to another file. For example, you can find all occurrences of a particular string of text on each line of a file and replace them with another string of text. For example, we have found a mistake in the feature names in our BED file where we need to replace all feature names that begin with PFD with PF. To do this use the command:

```
sed 's/PFD/PF/g' Pfalciparum.bed > Pfalciparum.fixed.bed [ENTER]
```

Now modify the command to change all 'VAR' features to be called 'VAR_gene' and all 'RIF' features to be called 'RIFIN_gene'. Write the final output to a file called Pfalciparum.modified.bed. How many features have changed?

A terminal window screenshot showing the execution of the sed command. The terminal title is 'wt@Pathogens: ~/Module_3_Programming/Linux'. The command entered is 'sed 's/PFD/PF/g' Pfalciparum.bed > Pfalciparum.fixed.bed'. The prompt changes from 'wt@Pathogens:~/Module_3_Programming/Linux\$' to 'wt@Pathogens:~/Module_3_Programming/Linux\$' after the command is executed, with a cursor at the end of the second line.

```
wt@Pathogens: ~/Module_3_Programming/Linux
wt@Pathogens:~/Module_3_Programming/Linux$ sed 's/PFD/PF/g' Pfalciparum.bed > Pfalciparum.fixed.bed
wt@Pathogens:~/Module_3_Programming/Linux$
```

Figure 7 Using sed to find and replace text in a file

Conclusion

The commands we have just seen can process vast amounts of information in a very short amount of time. They can be joined together to manipulate data and calculate results. Bioinformatics software often produces vast quantities of output results and these commands will help you filter things down to a more manageable level so that you can get meaningful findings out the other end. Learning how to use this small set of commands will save you a substantial amount of time.

What is a computer program?

A computer program is a sequence of written statements or instructions that can be understood by a computer in order to perform an overall task. A program must be written in a specific language (called a programming language) that is understood by the computer.

Many programming languages exist e.g. Perl, Python, Java, C++. It is not important which language you learn first as once you become familiar with one programming language, it is much easier to learn others. There is often a distinction between interpreted (e.g. Perl and Python) and compiled (e.g. C++ and Java) languages. People often call programs written in an interpreted language 'scripts'. All you need to know is that a script is just a program and scripting is just programming. In the remainder of this module we will introduce you to the notion of scripting using shell scripting.

Shell scripting

If you have a set of useful Linux commands that you want to use over and over again on different datasets, how do you do this without having to type the same commands over and over again? The command line has its own built-in programming language and can be used to create a "shell script".

To create a shell script, create a plain text file and add a series of Linux commands to the file and then treat that file as if it was any other Linux command or program. When you want to repeatedly execute the series of commands for multiple datasets, the shell script can be used to automate the task and save you lots of time.

There are several shell programs on a Linux system that can be used to execute shell scripts. These include ksh, tcsh and bash. You do not need to worry about this for now, you just need to know that for the remainder of this module you will use bash (Bourne Again SHell) and the shell scripts that you write will be called bash scripts.

Writing shell scripts

Any text editor can be used to write your script, but just remember that a word processor application (like Microsoft Word or LibreOffice) is **not** a text editor. A text editor doesn't allow you to format text (e.g. bold, italics, font sizes) and produces files in a plain (non-proprietary) format (.txt) that is readable by any computer.

Some freely available text editors

- Linux: vi, vim, nano, gedit, emacs
- Windows: notepad, Textpad, PSPad, Notepad++
- Mac OSX: vi, vim, nano, gedit, emacs, TextWrangler, TextEdit

```

wt@Pathogens: ~/.Module_3_Programming/Linux
wt@Pathogens: ~/.Module_3_Programming/Linux$ chmod +x sed_script.sh
wt@Pathogens: ~/.Module_3_Programming/Linux$ ./sed_script.sh
wt@Pathogens: ~/.Module_3_Programming/Linux$

sed_script.sh (~/.Module_3_Programming/Linux) - gedit
#!/bin/bash
input_filename=Pfalcciparum.bed
output_filename=Pfalcciparum.fixed.bed
sed 's/PFD/PF/g' $input_filename > $output_filename

```

Figure 8 A basic shell script which does the same thing as the previous example. The colours are added by the text editor to make it easier for a person to read and understand the script. The colours aren't used by the computer.

The basic structure of a shell script is shown in Figure 8. It is essentially just a list of Linux commands which are the individual instructions for the computer to follow. When creating a shell script it is standard practice to save it as a .sh file instead of a .txt file.

Running shell scripts

In order to get the computer to follow the instructions in a script you must execute (i.e. run) the script. In Linux, text files can be executed (i.e. run) as programs, provided they contain instructions in some language **and** the very first line of the text file starts with **#!** (referred to as shebang) followed by the path to a program that can understand (interpret) the instructions.

To run (execute) a text file you must give it execution privileges using `chmod`:

```
chmod +x sed_script.sh
```

and then execute it from the command line:

```
./sed_script.sh
```

Exercise : Hello World!

First let us look at a basic shell script. Navigate to the `Module_3_Programming` directory and then to the `BASH` directory and using your preferred text editor open the file `hello.sh`. You should see the shell script shown in Figure 9.

```
1  #!/bin/bash
3  #print to the screen
4  echo "Hello world!"
```

Figure 9 Hello world script

This is a simple shell script which prints the text "Hello world!" to the screen.

- **Line 1** tells the computer which program to use to execute this file, in this case it is the **bash** program. Every bash shell script that you write will always begin with the text `#!/bin/bash`.
- **Line 3** is a comment and acts as a note to yourself about what a line of code does. It is always good practice to add explanatory comments. In shell scripts, comments start with a `#` which tells the computer to ignore everything on this line.
- **Line 4** contains the Linux command `echo` which just prints text to the screen.

In a terminal window, type the following commands to give the `hello.sh` script execution privileges and run the script.

```
chmod +x hello.sh
```

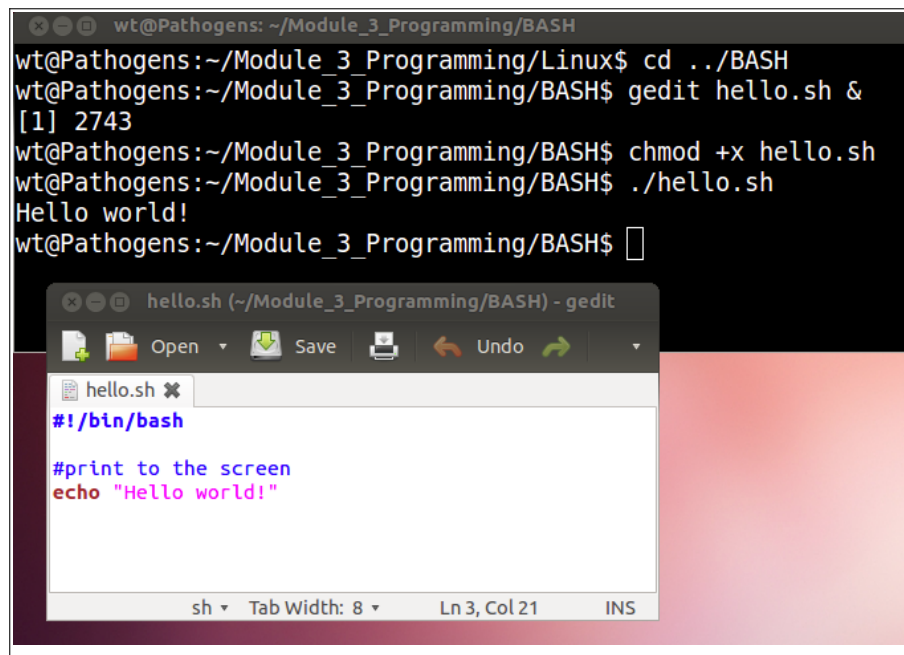
```
./hello.sh
```

```
Hello world!
```

Modify the script `hello.sh` so that it prints your name to the screen. Save this script as `my-name.sh`, give it execution privileges and then run it.

Congratulations, you have just written your first shell script!

Now it is time to make some Linux shell scripts that might actually be useful.



The image shows a terminal window and a gedit editor. The terminal window, titled 'wt@Pathogens: ~/Module_3_Programming/BASH', displays the following commands and output:

```
wt@Pathogens:~/Module_3_Programming/Linux$ cd ../BASH
wt@Pathogens:~/Module_3_Programming/BASH$ gedit hello.sh &
[1] 2743
wt@Pathogens:~/Module_3_Programming/BASH$ chmod +x hello.sh
wt@Pathogens:~/Module_3_Programming/BASH$ ./hello.sh
Hello world!
wt@Pathogens:~/Module_3_Programming/BASH$
```

The gedit editor, titled 'hello.sh (~/.Module_3_Programming/BASH) - gedit', shows the contents of the script:

```
#!/bin/bash

#print to the screen
echo "Hello world!"
```

The editor's status bar at the bottom indicates 'sh', 'Tab Width: 8', 'Ln 3, Col 21', and 'INS'.

Figure 10 Hello world script

Exercise : BWA mapping script

One common task in bioinformatics is to take raw reads from a sequencing machine and align them to a reference sequence (called mapping). This task requires a number of different commands to be run, with the `bwa` command performing the alignment of the sequences. If you have several different files (e.g. from different samples) of sequence data to analyse this can be quite time consuming. Therefore in this exercise we will create a shell script that can be used over and over again to map different samples (also known as lanes) of sequence data. Essentially this involves taking in 3 files and producing a single BAM file as output.

Navigate to the `Module_3_Programming` directory and then to the `BASH` directory and using your preferred text editor open the file `map_lanes.sh`. You should see the shell script shown in Figure 11.

```

1  #!/bin/bash

3  #read in values from command line
4  fastq1=$1
5  fastq2=$2
6  ref=$3
7  output=$4

9  #index the reference file
10 bwa index $ref

12 #map the sequence data
13 bwa aln $ref $fastq1 > F.sai
14 bwa aln $ref $fastq2 > R.sai
15 bwa sampe -a 700 $ref F.sai R.sai $fastq1 $fastq2 > $output.sam

17 #create a sorted and indexed bam file
18 samtools view -b -S $output.sam > $output.tmp.bam
19 samtools sort $output.tmp.bam $output
20 samtools index $output.bam

```

Figure 11 A shell script to map sequence data with bwa

This script performs the following set of standard mapping tasks:

- **Line 1** tells the computer which program to use to execute this file, in this case it is the **bash** program.
- **Lines 4-7** reads in the values passed to the script from the command line. These values are called command line arguments. We will discuss these in more detail later.
- **Line 10** indexes the reference file.
- **Lines 13-14** aligns the fastq files to the reference genome.
- **Line 15** extracts alignments from bwa's proprietary binary .sai file to a .sam file.
- **Line 18** converts the .sam file into a .bam file using samtools.
- **Lines 19-20** sorts and indexes the .bam file so that it can be viewed with Artemis.

In a terminal window, make the `map_lanes.sh` script executable and run it using the following commands:

```
chmod +x map_lanes.sh
```

```
./map_lanes.sh NV_1.fastq NV_2.fastq L2_cat.fasta NV
```

Please note that this script will run for several minutes, so please be patient. Lots of information about the progress of the mapping will be printed to the screen, but its rare you'd ever need to look at it. The data is from a *Chlamydia trachomatis* sample. While we wait let us learn more about variables and command line arguments.

Variables

In bash scripting, as in any scripting language, you use containers called variables to store data, change it, and access it later. New variables can be created like this:

```
name=value
```

In a bash script, you must do it exactly like this, with no spaces on either side of the equals sign, the variable name must contain only alphanumeric characters and underscores and it cannot start with a numeric character. Accessing the values stored in a variable can be done like this:

```
$name
```

In the `map_lanes.sh` script we create four different variables and use them to store the values that are passed to the script from the command line.

```
fastq1=$1  
fastq2=$2  
ref=$3  
output=$4
```

Later in the script we access the values stored in these variables. For example, we index the reference genome by passing the value that is stored in the `ref` variable to the `bwa index` command.

```
bwa index $ref
```


Command Line Arguments

Since we want to use the `map_lanes.sh` script on different datasets, it takes some arguments on the command line telling it what to work on. These arguments are:

- Name of the input fastq files
- Name of the reference file to use
- A prefix to use when writing output files (e.g. `<prefix>.bam`).

Remember we have run the `map_lanes.sh` script with the following command line arguments

```
$ ./map_lanes.sh NV_1.fastq NV_2.fastq L2_cat.fasta NV
```

A shell script can have any number of command line arguments which can be accessed in the script using the variables `$0`, `$1`, `$2`, `$3`, `$4`, `$5` etc.

- The variable `$0` is the script's name, when run with the command above this variable will contain the value `./map_lanes.sh`
- The variable `$1` is the first argument passed to the script, when run with the command above this variable will contain the value `"NV_1.fastq"`
- The variable `$2` is the second argument passed to the script, when run with the command above this variable will contain the value `"NV_2.fastq"`
- The variable `$3` is the third argument passed to the script, when run with the command above this variable will contain the value `"L2_cat.fasta"`
- The variable `$4` is the fourth argument passed to the script, when run with the command above this variable will contain the value `"NV"`
- The total number of arguments is stored in `$#`.

When the `map_lanes.sh` script is finished running, type `ls` to see the contents of the directory. You should see a new file called `NV.bam` which contains the results of mapping the files `NV_1.fastq` and `NV_2.fastq` to the `L2_cat.fasta` reference sequence.

Why is the file called `NV.bam`?

Now use the `map_lanes.sh` script to map the files `AV_1.fastq` and `AV_2.fastq` to the `L2_cat.fasta` reference sequence.

Exercise : BWA mapping script - what could go wrong?

Try running the `map_lanes.sh` script with the following command line arguments:

```
$ ./map_lanes.sh NV_1.fastq NV_2.fastq L2.fasta NV2
```

Did the script run successfully? If not, why not?

Often, the difference between a good script and a poor script is assessed in terms of the robustness of the script. That is, the ability of the script to handle situations in which something goes wrong. In this case, does the `map_lanes.sh` script handle the situation where a file supplied by the user does not exist?

In this example we will look at improving the robustness of the `map_lanes.sh` script by adding some argument and error checking to the script. Navigate to the `Module_3_Programming` directory and then to the `BASH` directory and using your preferred text editor open the file `map_lanes_validate_inputs.sh`. You should see the shell script shown in Figure 12.

```

1  #!/bin/bash

3  #read in values from command line
4  fastq1=$1
5  fastq2=$2
6  ref=$3
7  output=$4

9  #check the correct number of arguments are passed to the script
10 if [ $# != 4 ]; then
11     echo "Usage: $0 fastq1 fastq2 reference out_prefix"
12     exit
13 fi

15 #check the fastq and reference files passed to the script exist
16 if [ ! -f $fastq1 ] || [ ! -f $fastq2 ] || [ ! -f $ref ]; then
17     echo "Error: One of the input files does not exist"
18     exit
19 fi

21 #index the reference file
22 bwa index $ref

24 #map the sequence data
25 bwa aln $ref $fastq1 > F.sai
26 bwa aln $ref $fastq2 > R.sai
27 bwa sampe -a 700 $ref F.sai R.sai $fastq1 $fastq2 > $output.sam

29 #create a sorted and indexed bam file
30 samtools view -b -S $output.sam > $output.tmp.bam
31 samtools sort $output.tmp.bam $output
32 samtools index $output.bam

```

Figure 12 Error checking in a shell script

This script performs some checks on the values passed to it from the command line and then performs a set of standard mapping tasks:

- **Lines 1-7** tell the computer which program to use to execute this file and reads in the values passed to the script from the command line.
- **Lines 10-13** checks that the correct number of command line arguments have been passed to the script. The lines say if the number of command line arguments passed to the script is NOT EQUAL TO 4, print a message to the screen telling the user what the correct usage is and exit the script.
- **Lines 16-19** checks that all the files passed to the script exist. The lines say if the first fastq file does not exist OR the second fastq file does not exist OR the reference file does not exist, print an error message to the screen and exit the script.
- **Lines 22-32** perform a set of standard mapping tasks.

Please note:

`##` is the number of command line arguments

`!=` means NOT EQUAL TO

`$0` is the name of the script

`!` is the NOT operator

`-f` checks if a file exists

`||` means OR

Modify the script to check to see if the output file already exists. If it does exist, print a warning message and exit from the script.

Decision Statements

Sometimes you will want to perform different tasks depending on whether a condition is true or false. In bash this can be achieved with the keyword, `if`. The `if` statement consists of a condition that is evaluated, and a block of code that is run if the condition evaluates to true as shown in Figure 13.

```
1 if [ CONDITION ]; then
2   # instructions to follow if condition is true
3 fi
```

Figure 13 BASH if statement

Loops

In bioinformatics we often have to perform the same action/analysis multiple times. For example, its quite common to multiplex a 96 well plate of samples into a single Illumina lane, so to analyse your data you'll need to run the same commands on all 96 sets of sequencing data. Rather than running a script over and over again, you can use a loop. It will keep running a set of commands until a condition is met, for example, loop over all files in a directory and run the commands on each file.

In bash this can be achieved with the keyword `FOR`. The `FOR` statement consists of a list and a variable name, then a block of commands to run. In the example in Figure 14, the `ls` command is run to get a list of files in the current directory. Each file is then taken in turn and is assigned to the variable `i`. The block of code is then run, and `$i` contains the name of the file. Here we just print out the filename, but you can use any command.

```
1  FOR i in $( ls ); DO
2      echo $i
3  DONE
```

Figure 14 Bash `FOR` statement

Exercise : Mapping to Multiple references

Modify the script `map_lanes_validate_inputs.sh` from the previous exercise so that it takes in 2 fastq files and a directory containing references as input, and maps the fastq files to each reference. The references are contained in the sub directory called `references`. It is sometimes necessary to map to multiple different references, because you may not know in advance which reference best represents the underlying genome of the sample you've sequenced.

Save this script as `multiple_mappings.sh`, make it executable and run it with the following command line arguments:

```
./multiple_mappings.sh NV_1.fastq NV_2.fastq references
```

Hint: There are 3 references in the directory, so 3 BAM files should be produced as output.

Useful resources

Although we are not teaching Perl as part of this course, we have included a copy of the e-book "Beginning Perl for Bioinformatics" as a pdf which is a good starting point for anyone who wants to delve further into programming and teach themselves Perl. This book is stored as a pdf file on the USB disk and is called "Beginning_Perl_for_Bioinformatics.pdf".

Some further useful resources include:

Books

- "Unix in a Nutshell" by Arnold Robbins
- "Learning the Bash Shell" by Bill Rosenblatt and Cameron Newham
- "Learning Perl" by Randall Schwartz
- "Mastering Perl for Bioinformatics" by James Tisdall
- "Perl programming for biologists" by D. Curtis Jamison
- "Mastering Regular Expressions" by Jeffrey Friedl

Online resources

- <http://www.perl.com/pub/a/2000/10/begperl1.html>
- <http://www.bioperl.org>
- <http://search.cpan.org>

We hope you have fun learning to program!