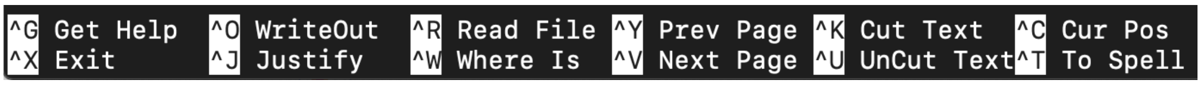
Using nano

If you want to create a script using Nano or edit a script, go on the terminal and write:
nano scriptname.sh

This will open a sh file with the name "scriptname."
This is a text document where you can write like any other document.

Nano has many functions. To perform an option, press control and the given letter associated with the option. Exit is ctrl + X. 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
```

Comments

If you want to add comments to a script without the script running them as code, add a #. Everything after # will be a comment,

Example:

```
# This and the rest of this line is now a comment
```

Shebang

A shebang is the character sequence of #! at the beginning of the script. It tells the operating system the path to interpret the file.
If one uses bash, you would have the shebang as well as the path to the bash binary:
#!/bin/bash

Remember: **All bash scripts need to start with the above shebang line.**

Echo

Echo is a command used to print the output to the terminal.

If you want to print the sentence "Hello World"
You can use the following commands:

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

Using quotations is safer and is the recommended route, especially if the print statement has quotes.

If you want an empty line, have an echo stand alone:
echo

Running bash scripts

To run a bash script, you can say:
`bash scriptname.sh`

If you want to make the script executable, run this command on the file:
`chmod 755 scriptname.sh`

From then on, you can run the script using:
`./scriptname.sh`

Making files executable makes them easier for automation and ensures that you don't have to provide an interpreter.

Variable

Variables are a good way of defining values that needs to change between running scripts. They are also brilliant if you use the same value multiple times in one script. You must assign the variable before using it; otherwise, the interpreter does not know what to do with the variable.

Variables can be used to set values but also define directories.

```
pet=”polar bear”
edu=astrophysicist
wish=baker
dir=/path/to/dir/

echo “Mia wants another $pet”
output: Mia wants another polar bear

echo “Mia studied to be an $edu but really wanted to be a $wish”
output: Mia studied to be an astrophysicist but really wanted to be a
baker
```

If you want to save the result of a command and store it in a variable:
`var=$(command)`

This variable can be used in echo. You can thereby print the output of a command:
`echo “There are $var sequences”`

Positional arguments

Instead of assigning the variable in the script, one can supply the variable in the command line. You do this by positional arguments.

$1 (first positional argument)
$2 (second positional argument)

Here is an example:

```
# If the script scriptname.sh includes the line:                     Bash
echo "I love $1 but only on $2"

# You can supply the positional arguments while executing the script. Linux
./scriptname.sh vegemite Tuesdays
output: I love vegemite, but only on Tuesdays
```

Word count – Only counting lines

As previously learned in the Linux section, you can use 'wc' to count words and lines. If you want to count how many sequences there are in a fasta file, combining grep and wc through pipe | would be good. If you only want to count the lines with the fasta header (marked by '>'), then add -l (l for lines) to the wc command:

```
wc -l
```

Integer (number) operator

These operators can be used to compare integers (numbers with no decimal). You would use them in *if statements*.

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* is a programming conditional statement that performs an action when proven true. The general *if statement* is as below;

```
if [ <expression> ]                                    bash
then
        <statement>
fi
```

The <expression> is what it will check and see if it is true.
If true, it will run the <statement> between *then* and *fi*.
fi marks the end of the *if statement*.
Use ‘tab’ to set the indentation before the <statement>.

It is essential to note the whitespace around the [ ] brackets.
A white space must exist between *if* and the first bracket ‘[‘.
There also needs to be a whitespace between the inside of the bracket and the <statement> on both sides ([<statement>]).

Here is an example:

```bash
if [ $1 -lt 12 ]                                             Bash
then
	echo “We have less than 12 days to Christmas!”
fi
```

Elif and else statements

There is also the possibility to add extra *if statements*; these are called *elif* (else if). Unlike *if statements*, you can have many of these following each other. You can also add an *else statement*. This triggers if neither *if* nor *elif* is TRUE.

```bash
if [ <expression1> ]                                         Bash
then
	<statement1>
elif [ <expression2> ]
then
	<statement2>
else
	<statement3>
fi
```

For loops

A loop is an iteration statement. It is used to repeat a process until a particular situation is reached. For *for loops* specifically, you repeat a process over all items <var> in each list <iterable>.

```bash
for <var> in <iterable>                                      Bash
do
	<statement>
done

# Example
for filename in /a/path/to/dir/*
do
	echo “I guess $filename is here”
done
```

Note that you didn't have to call it **filename** but could have called it cats or whatever pleases you. The **filename** in the *for loop* can now be used as a variable within the *for loop*.

For loops ends with a *done* instead of *fi* as seen in *if statements*.

Looping over specific files in directories

If you want to loop over specific files in a dictionary, specify it in the 'in <iterable>' part of the *for loop*. Here you would use the wildcard (*) character so that it would find all files of the specific type.

For example, if you want to loop over txt files in the directory books, use the following command:

```
for textfile in /books/*.txt
```

Arrays and looping over them

An array is like a list of items. You declare an array like you would assign a variable. You give it a name by using a '=' with no whitespace around it, enclose the array in parenthesis and type in the strings with quotations.

```
Array=("string" "string2" "string")
```

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.

Here is an example:

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

for cutiepie in {cutePetArray[@]}
do
      echo "A $cutiepie is a cutiepie!"
done
output:      A polar bear is a cutiepie!
             A guinea pig is a cutiepie!
             A snake is a cutiepie!
             A Loch Ness Monster is a cutiepie!
```