Bash Scripting

1.Introduction

The following pages are intended to give you a solid foundation in how to write Bash scripts, to get the computer to do complex, repetitive tasks for you. You won't be a bash guru at the end but you will be well on your way and armed with the right knowledge and skills to get you there if that's what you want (which you should).

Bash scripts are used by Systems Administrators, Programmers, Network Engineers, Scientists and just about anyone else who uses a Linux/ Unix system regularly. No matter what you do or what your general level of computer proficiency is, you can generally find a way to use Bash scripting to make your life easier.

What are scripts anyways?

Think of a script for a play, or a movie, or a TV show. The script tells the actors what they should say and do. A script for a computer tells the computer what it should do or say. In the context of Bash scripts we are telling the Bash shell what it should do.
A Bash script is a plain text file which contains a series of commands. These commands are a mixture of commands we would normally type ouselves on the command line (such as **ls** or **cp**for example) and commands we could type on the command line but generally wouldn't (you'll discover these over the next few pages). An important point to remember though is:

Anything you can run normally on the command line can be put into a script and it will do exactly the same thing. Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.

First Script

As we discussed earlier that script is a normal text file with commands in it.
We will open a file vi editor and add some commands in it.
It is convention to give files that are Bash scripts an extension of **.sh** (print.sh for example)

```
$ vi print.sh

#!/bin/bash
# A sample Bash script
echo Hello World!
```

Let's break it down:
Line 1 - Is what's referred to as the **shebang**.
This is the first line of the script above. The hash exclamation mark (#!) character sequence is referred to as the Shebang. Following it is the path to the interpreter (or program) that should be used to run (or interpret) the rest of the lines in the text file. (For Bash scripts it will be the path to Bash, but there are many other types of scripts and they each have their own interpreter.)

Line 2- This is a comment. Anything after **#** is not executed. It is for our reference only.
Line 3- Is the command echo which will print a message to the screen. You can type this command yourself on the command line and it will behave exactly the same.

How do we run them?

Running a Bash script is fairly easy. Another term you may come across is **executing** the script (which means the same thing). Before we can execute a script it must have the execute permission set (for safety reasons this persmission is generally not set by default). If you forget to grant this permission before running the script you'll just get an error message telling you as such and no harm will be done.

```
imran@DevOps:.../bash$ ./print.sh
bash: ./print.sh: Permission denied
imran@DevOps:.../bash$ ls -l
total 4
-rw-rw-r-- 1 imran imran 53 Oct 21 17:33 print.sh
imran@DevOps:.../bash$ chmod 755 print.sh
imran@DevOps:.../bash$ ls -l
total 4
-rwxr-xr-x 1 imran imran 53 Oct 21 17:33 print.sh
imran@DevOps:.../bash$ ./print.sh
Hello World!
```

2.Variables!

Temporary stores of information

How do they Work?

A variable is a temporary store for a piece of information. There are two actions we may perform for variables:

- Setting a value for a variable.
- Reading the value for a variable.

To read the variable we then place its name (preceded by a $ sign) anywhere in the script we would like.

```
imran@DevOps:.../bash$ VAR1=123
imran@DevOps:.../bash$ echo $VAR1
123
```

Command line arguments

When we run a program on the command line you would be familiar with supplying arguments after it to control its behaviour. For instance we could run the command **ls -l /etc**. **-l** and **/etc** are both command line arguments to the command **ls**. We can do similar with our bash scripts. To do this we use the variables **$1** to represent the first command line argument, **$2** to represent the second command line argument and so on. These are automatically set by the system when we run our script so all we need to do is refer to them.
Let's look at an example.

```
#!/bin/bash
# A simple copy script
cp $1 $2
# Let's verify the copy worked
echo Details for $2
ls –lh $2
```

Let's break it down:

- **Line 4** - run the command **cp** with the first command line argument as the source and the second command line argument as the destination.
- **Line 8** - run the command **echo** to print a message.

- **Line 9** - After the copy has completed, run the command **ls** for the destination just to verify it worked. We have included the options **l** to show us extra information and **h** to make the size human readable so we may verify it copied correctly.

Other Special Variables

There are a few other variables that the system sets for you to use as well.

- **$0** - The name of the Bash script.
- **$1** - **$9** - The first 9 arguments to the Bash script. (As mentioned above.)
- **$#** - How many arguments were passed to the Bash script.
- **$@** - All the arguments supplied to the Bash script.
- **$?** - The exit status of the most recently run process.
- **$$** - The process ID of the current script.
- **$USER** - The username of the user running the script.
- **$HOSTNAME** - The hostname of the machine the script is running on.
- **$SECONDS** - The number of seconds since the script was started.
- **$RANDOM** - Returns a different random number each time is it referred to.
- **$LINENO** - Returns the current line number in the Bash script.

Setting your own varibles

```
imran@DevOps:~/.../bash_scripts$ cat 1_print.sh
intA=20
floatB=20.20
stringA="first_string"
DIR_PATH="/tmp"

echo
echo "##########################"
echo "Value of integer A is $intA"
echo "##########################"
```

```
echo "Value of Float B is $intB"

echo "########################"
echo "Value of string A is $stringA"

echo "########################"
echo "Directory path is $DIR_PATH"

echo "########################"
echo "Content of TMP directory."
echo "########################"
ls $DIR_PATH
```

Quotes

In the example above we kept things nice and simple. The variables only had to store a single word. When we want variables to store more complex values however, we need to make use of quotes. This is because under normal circumstances Bash uses a space to determine separate items.
imran@DevOps:.../bash$ myvar=Hello World
World: command not found
imran@DevOps:.../bash$
When we enclose our content in quotes we are indicating to Bash that the contents should be considered as a single item. You may use single quotes (') or double quotes (").

Single quotes will treat every character literally.
Double quotes will allow you to do substitution (that is include variables within the setting of the value).

```
imran@DevOps:.../bash$ myvar='Hello World'
imran@DevOps:.../bash$ echo $myvar
Hello World
imran@DevOps:.../bash$ newvar="More $myvar"
imran@DevOps:.../bash$ echo $newvar
More Hello World
imran@DevOps:.../bash$ newvar='More $myvar'
imran@DevOps:.../bash$ echo $newvar
More $myvar
imran@DevOps:.../bash$
```

Command Substitution

Command substitution allows us to take the output of a command or program (what would normally be printed to the screen) and save it as the value of a variable. To do this we place it within brackets, preceded by a $ sign.

```
imran@DevOps:.../bash$ myvar=$( ls /etc | wc -l )
imran@DevOps:.../bash$ echo There are $myvar entries in the directory /etc
There are 249 entries in the directory /etc
```

Exporting Variables

Variable defined in the script leave with it and dies after the script dies or completes. If we want to define a variable that is accessible to all the scripts from your current we need to export it.
Export a variable from bash shell as mentioned below.

```
imran@DevOps:.../bash$ var1=foo
imran@DevOps:.../bash$ echo $var1
foo
imran@DevOps:.../bash$ export var1
```

Create a script which prints exported and local variable

```
imran@DevOps:.../bash$ vi script1.sh
#!/bin/bash
# demonstrate variable scope

var2=foobar

echo "Printing exported varible from bash shell"

echo $var1

echo "Printing varible defined in the script"
echo $var2
```

Execute the script to see the results

```
imran@DevOps:.../bash$ ./script1.sh
```

Printing exported varible from bash shell
`foo`
Printing varible defined in the script
`foobar`

Summary

$1, $2, ...
The first, second, etc command line arguments to the script.

variable=value
To set a value for a variable. Remember, no spaces on either side of =

Quotes " '
Double will do variable substitution, single will not.

variable=$(command)

Save the output of a command into a variable

export var1

Make the variable var1 available to child processes.

3.User Input!

Let's make our scripts interactive.

Ask the User for Input

If we would like to ask the user for input then we use a command called **read**. This command takes the input and will save it into a variable.
read var1

Let's look at a simple example:

```
#!/bin/bash
# Ask the user for their name
echo Hello, who am I talking to?
read varname
echo It\'s nice to meet you $varname
```

Let's break it down:

- **Line 3 -** Print a message asking the user for input.
- **Line 4 -** Run the command **read** and save the users response into the variable **varname**
- **Line 5 – echo** another message just to verify the read command worked. Note: I had to put a backslash (\ ) in front of the ' so that it was escaped.

You are able to alter the behaviour of **read** with a variety of command line options. (See the man page for read to see all of them.) Two commonly used options however are **-p** which allows you to specify a prompt and **-s** which makes the input silent. This can make it easy to ask for a username and password combination like the example below:

Login.sh

```
#!/bin/bash
# Ask the user for login details
read -p 'Username: ' uservar
read -sp 'Password: ' passvar
echo
echo Thankyou $uservar we now have your login details
```

So far we have looked at a single word as input. We can do more than that however.

cars.sh

```
#!/bin/bash
# Demonstrate how read actually works
echo What cars do you like?
read car1 car2 car3
```

```
    echo Your first car was: $car1
    echo Your second car was: $car2
    echo Your third car was: $car3
```

```
imran@DevOps:.../bash$ ./cars.sh
What cars do you like?
Porche Ford Toyota
Your first car was: Porche
Your second car was: Ford
Your third car was: Toyota
```

4. If Statements!

Decisions, decisions.

Basic If Statements

A basic if statement effectively says, **if** a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions. If follows the format below:

---

if [<some test>]
then
<commands>
fi

Anything between **then** and **fi** (if backwards) will be executed only if the test (between the square brackets) is true.

Let's look at a simple example:

if_example.sh

```
#!/bin/bash
# Basic if statement
if [ $1 -gt 100 ]
then
  echo Hey thats a large number.
  pwd
fi
date
```

Let's break it down:

- **Line 4** - Let's see if the first command line argument is greater than 100
- **Line 6 and 7** - Will only get run if the test on line 4 returns true. You can have as many commands here as you like.
- **Line 8** - fi signals the end of the if statement. All commands after this will be run as normal.
- **Line 10** - Because this command is outside the if statement it will be run regardless of the outcome of the if statement.

```
imran@DevOps:.../bash$ ./if_example.sh 150
Hey thats a large number.
```

```
/tmp/bash
Sun Oct 30 16:19:28 IST 2016
```

Sample if/else condition script

```
#!/bin/bash

intA=20
intB=30
if [ intA==intB ];
then
  echo "intA is not equal to intB"
else
  echo "Equal"
fi

if [ -f hosts ];
then
  echo "File exists!"
else
  echo "Does not exist"
fi
```

Test

The square brackets ([ ]) in the **if** statement above are actually a reference to the command **test**. This means that all of the operators that test allows may be used here as well. Look up the man page for test to see all of the possible operators (there are quite a few) but some of the more common ones are listed below.

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.

A few points to note:

- **=** is slightly different to **-eq**. [001 = 1] will return false as = does a string comparison (ie. character for character the same) whereas -eq does a numerical comparison meaning [001 -eq 1] will return true.

- When we refer to FILE above we are actually meaning a path. Remember that a path may be absolute or relative and may refer to a file or a directory.

- Because [5 is just a reference to the command **test** we may experiment and trouble shoot with test on the command line to make sure our understanding of its behaviour is correct.

5. Loops!

Round and round we go.

Loops allow us to take a series of commands and keep re-running them until a particular situation is reached. They are useful for automating repetitive tasks.
The **for** loop is a little bit different to the previous two loops. What it does is say for each of the items in a given list, perform the given set of commands. It has the following syntax.

---

for var in <list>
do
<commands>
done

The for loop will take each item in the list (in order, one after the other), assign that item as the value of the variable **var**, execute the commands between do and done then go back to the top, grab the next item in the list and repeat over.

The list is defined as a series of strings, separated by spaces.

`For loops iterate for as many arguments given:`

`Example:`

The contents of $Variable is printed three times.

```
#!/bin/bash
for Variable in {1..3}
do
    echo "$Variable"
done
```

Or write it the "traditional for loop" way:

```
for ((a=1; a <= 3; a++))
do
    echo $a
done
```

They can also be used to act on files..
This will run the command 'cat' on file1 and file2

```
for Variable in file1 file2
do
    cat "$Variable"
done
```

or the output from a command

This will cat the output from ls.

```
for Output in $(ls)
do
    cat "$Output"
done
```

Printing list of host ip addresses from hosts file.

Create a file named hosts

```
$ vi hosts
192.168.1.10
192.168.1.11
192.168.1.12
192.168.1.13
```

```
#!/bin/bash
for i in `cat hosts`;do
  echo "Printing list of hosts."
  echo $i
done
```

6. Real time use cases

Backup script

Being working with systems you may need to take backup of files, directories, log files etc. Below mentione scenario shows you how you can automate the backup procedures.

In this example we will create a file and mention the name of log files that needs to be backup up with tar command. Before taking backup our script will also tell us if the log file exists or not. It will skip the backup procedure if the file does not exists. After all there is no point running backup command if the log file does not exist.
Create directory for storing log files

```
$ mkdir -p /tmp/scripts/logs
```

Put some files in the logs directory.

```
$ cd /tmp/scripts/logs
$ touch ansible.log  apache.log  mysql.log  nagios.log
```

You can choose to put some content in the log files, touch will just create empty files.

```
$ cd /tmp/scripts
```

Create a file where you place the name of the files that you want to backup.

```
$ vi backup_files.txt
apache.log
mysql.log
nagios.log
ansible.log
chef.log
```

There is one extra filename chef.log which is not present in our logs directory /tmp/scripts/logs.
We will see how we will handle it in our script

```
$ vi backup.sh
#!/bin/bash

LOG_DIR='/tmp/scripts/logs'
BACKUP_DIR='/tmp/scripts/logs_backup'

mkdir -p $BACKUP_DIR
for i in `cat backup_files.txt`; do
  if [ -f $LOG_DIR/$i ];
  then
```

```
    echo "Copying $i to logs_backup directory."
    cp $LOG_DIR/$i $BACKUP_DIR
  else
    echo "$i log file does exist, skipping."
  fi
done

echo
echo

echo "Zipping log files"
tar -czvf logs_backup.tgz logs_backup
echo
echo
echo "Backup completed successfully."
```

Running command on remote servers/nodes

Sometime we need to run a command or set of commands on multiple nodes/server.
We use ssh to login to these nodes and run that command individually and manually on all the nodes/servers.
But its a very time consuming and mundane work if you have to do it many times.
We will write a bash script to do that.

For this exercise we will choose to run “yum install httpd” on three nodes.
Assumtions:
1. Three centos vm’s
2. Vm’s have internet connection to download and install softwares.
3. All vm’s have same username to connect to

We will create a file named “hosts-dev” and add ip address of all three nodes in that.

```
$ vi hosts-dev
192.168.2.5
192.168.2.6
192.168.2.7
```

We will write a for loop which will read the ip address from the hosts file and run yum install httpd command over ssh. Install package has to be

```
$ vi install.sh
#!/bin/bash
for hosts in `cat hosts-dev`
do
  ssh vagrant@$hosts sudo yum install httpd -y
done
```

Lets break it down

Line 2: for loop will run over the content on hosts-dev file one by one, which are the ipadresses of the vm's. Notice we have used backticks `` and not single quotes '' to read the hosts-dev file (`cat hosts-dev`).
Line 4: we are running `sudo yum install httpd -y` command over ssh. $hosts variable will hold the ip address and will establish ssh connection with vagrant user(ssh vagrant@$hosts).
This loop will run untill we exaust all the entries in the hosts-dev file, if you have lets say 50 nodes you can add ip address of all the 50 nodes in this file.

Every time it logs into the vm's/sever/nodes it will ask you a password, it will be painfull if you have lot of servers that you manage to enter password manually. In the next section we will deal with this issue by doing ssh key exchange.

7. How To Set Up SSH Keys

About SSH Keys

SSH keys provide a more secure way of logging into a virtual private server with SSH than using a password alone. While a password can eventually be cracked with a brute force attack, SSH keys are nearly impossible to decipher by brute force alone. Generating a key pair provides you with two long string of characters: a public and a private key. You can place the public key on any server, and then unlock it by connecting to it with a client that already has the private key. When the two match up, the system unlocks without the need for a password. You can increase security even more by protecting the private key with a passphrase.

Step One—Create the RSA Key Pair

The first step is to create the key pair on the client machine (there is a good chance that this will just be your computer):

```
$ ssh-keygen -t rsa
```

Step Two—Store the Keys and Passphrase

Once you have entered the Gen Key command, you will get a few more questions:

Enter file in which to save the key (/home/demo/.ssh/id_rsa):

You can press enter here, saving the file to the user home (in this case, my example user is called demo).

Enter passphrase (empty for no passphrase):

It's up to you whether you want to use a passphrase. Entering a passphrase does have its benefits: the security of a key, no matter how encrypted, still depends on the fact that it is not visible to anyone else. Should a passphrase-protected private key fall into an unauthorized users possession, they will be unable to log in to its associated accounts until they figure out the passphrase, buying the hacked user some extra time. The only downside, of course, to having a passphrase, is then having to type it in each time you use the Key Pair.

The entire key generation process looks like this:

```
$ ssh-keygen -t rsa

Generating public/private rsa key pair.

Enter file in which to save the key (/home/demo/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:
```

```
Your identification has been saved in /home/demo/.ssh/id_rsa.

Your public key has been saved in /home/demo/.ssh/id_rsa.pub.

The key fingerprint is:

4a:dd:0a:c6:35:4e:3f:ed:27:38:8c:74:44:4d:93:67 demo@a

The key's randomart image is:

+--[ RSA 2048]----+

|          .oo.   |

|         .  o.E  |

|        + .  o   |

|     . = = .     |

|      = S = .    |

|     o + = +     |

|      . o + o .  |

|           . o   |

|                 |

+-----------------+
```

The public key is now located in /home/demo/.ssh/id_rsa.pub The private key (identification) is now located in /home/demo/.ssh/id_rsa

Step Three—Copy the Public Key

Once the key pair is generated, it's time to place the public key on the virtual server that we want to use.

You can copy the public key into the new machine's authorized_keys file with the ssh-copy-id command. Make sure to replace the example username and IP address below.

```
$ ssh-copy-id user@192.168.2.5
```

Alternatively, you can paste in the keys using SSH:

```
$ cat ~/.ssh/id_rsa.pub | ssh user@192.168.2.5 "mkdir -p ~/.ssh && cat >>
~/.ssh/authorized_keys"
```

No matter which command you chose, you should see something like:

```
The authenticity of host '192.168.2.5 (192.168.2.5)' can't be established.RSA
key fingerprint is b1:2d:33:67:ce:35:4d:5f:f3:a8:cd:c0:c4:48:86:12.

Are you sure you want to continue connecting (yes/no)? Yes

Warning: Permanently added '12.34.56.78' (RSA) to the list of known hosts.

user@12.34.56.78's password:

Now try logging into the machine, with "ssh 'user@12.34.56.78'", and check in:



  ~/.ssh/authorized_keys



to make sure we haven't added extra keys that you weren't expecting.
```

Now you can go ahead and log into user@192.168.2.5 and you will not be prompted for a password. However, if you set a passphrase, you will be asked to enter the passphrase at that time (and whenever else you log in in the future).

Quick reference for Bash

Below mentioned script put all the major **bash scripts syntax** we learned in one script.

```
#!/bin/bash
# First line of the script is shebang which tells the system how
to execute
# the script: http://en.wikipedia.org/wiki/Shebang_(Unix)
# As you already figured, comments start with #. Shebang is also a
comment.

# Simple hello world example:
echo Hello world!

# Each command starts on a new line, or after semicolon:
echo 'This is the first line'; echo 'This is the second line'
```

```
# Declaring a variable looks like this:
Variable="Some string"

# But not like this:
Variable = "Some string"
# Bash will decide that Variable is a command it must execute and
give an error
# because it can't be found.

# Or like this:
Variable= 'Some string'
# Bash will decide that 'Some string' is a command it must execute
and give an
# error because it can't be found. (In this case the 'Variable='
part is seen
# as a variable assignment valid only for the scope of the 'Some
string'
# command.)

# Using the variable:
echo $Variable
echo "$Variable"
echo '$Variable'
# When you use the variable itself — assign it, export it, or else
— you write
# its name without $. If you want to use the variable's value, you
should use $.
# Note that ' (single quote) won't expand the variables!

# String substitution in variables
echo ${Variable/Some/A}
# This will substitute the first occurrence of "Some" with "A"

# Substring from a variable
Length=7
echo ${Variable:0:Length}
# This will return only the first 7 characters of the value

# Default value for variable
```

```
echo ${Foo:-"DefaultValueIfFooIsMissingOrEmpty"}
# This works for null (Foo=) and empty string (Foo=""); zero
(Foo=0) returns 0.
# Note that it only returns default value and doesn't change
variable value.

# Builtin variables:
# There are some useful builtin variables, like
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
echo "All arguments passed to script: $@"
echo "Script's arguments separated into different variables: $1
$2..."

# Reading a value from input:
echo "What's your name?"
read Name # Note that we didn't need to declare a new variable
echo Hello, $Name!

# We have the usual if structure:
# use 'man test' for more info about conditionals
if [ $Name != $USER ]
then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# NOTE: if $Name is empty, bash sees the above condition as:
if [ != $USER ]
# which is invalid syntax
# so the "safe" way to use potentially empty variables in bash is:
if [ "$Name" != $USER ] ...
# which, when $Name is empty, is seen by bash as:
if [ "" != $USER ] ...
# which works as expected

# There is also conditional execution
```

```
echo "Always executed" || echo "Only executed if first command
fails"
echo "Always executed" && echo "Only executed if first command
does NOT fail"

# To use && and || with if statements, you need multiple pairs of
square brackets:
if [ "$Name" == "Steve" ] && [ "$Age" -eq 15 ]
then
    echo "This will run if $Name is Steve AND $Age is 15."
fi

if [ "$Name" == "Daniya" ] || [ "$Name" == "Zach" ]
then
    echo "This will run if $Name is Daniya OR Zach."
fi

# Expressions are denoted with the following format:
echo $(( 10 + 5 ))

# Unlike other programming languages, bash is a shell so it works
in the context
# of a current directory. You can list files and directories in
the current
# directory with the ls command:
ls

# These commands have options that control their execution:
ls -l # Lists every file and directory on a separate line

# Results of the previous command can be passed to the next
command as input.
# grep command filters the input with provided patterns. That's
how we can list
# .txt files in the current directory:
ls -l | grep "\.txt"

# You can redirect command input and output (stdin, stdout, and
stderr).
```

```
# Read from stdin until ^EOF$ and overwrite hello.py with the
lines
# between "EOF":
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF

# Run hello.py with various stdin, stdout, and stderr
redirections:
python hello.py < "input.in"
python hello.py > "output.out"
python hello.py 2> "error.err"
python hello.py > "output-and-error.log" 2>&1
python hello.py > /dev/null 2>&1
# The output error will overwrite the file if it exists,
# if you want to append instead, use ">>":
python hello.py >> "output.out" 2>> "error.err"



# Commands can be substituted within other commands using $( ):
# The following command displays the number of files and
directories in the
# current directory.
echo "There are $(ls | wc -l) items here."

# The same can be done using backticks `` but they can't be nested
- the preferred way
# is to use $( ).
echo "There are `ls | wc -l` items here."

# Bash uses a case statement that works similarly to switch in
Java and C++:
```

```
case "$Variable" in
    #List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";;
esac

# for loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done

# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done

# They can also be used to act on files..
# This will run the command 'cat' on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command
# This will cat the output from ls.
for Output in $(ls)
do
    cat "$Output"
done

# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done
```

```
# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt
# prints first 10 lines of file.txt
head -n 10 file.txt
# sort file.txt's lines
sort file.txt
# report or omit repeated lines, with -d it reports them
uniq -d file.txt
# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt
# replaces every occurrence of 'okay' with 'great' in file.txt,
(regex compatible)
sed -i 's/okay/great/g' file.txt
# print to stdout all lines of file.txt which match some regex
# The example prints lines which begin with "foo" and end in "bar"
grep "^foo.*bar$" file.txt
# pass the option "-c" to instead print the number of lines
matching the regex
grep -c "^foo.*bar$" file.txt
# if you literally want to search for the string,
# and not the regex, use fgrep (or grep -F)
fgrep "^foo.*bar$" file.txt
```