If Statements!

Decisions, decisions.

Introduction

Bash if statements are very useful. In this section of our Bash Scripting Tutorial you will learn the ways you may use if statements in your Bash scripts to help automate tasks.

If statements (and, closely related, **case** statements) allow us to make decisions in our Bash scripts. They allow us to decide whether or not to run a piece of code based upon conditions that we may set. If statements, combined with loops (which we'll look at in the next section) allow us to make much more complex scripts which may solve larger tasks.

Like what we have looked at in previous sections, their syntax is very specific so stay on top of all the little details.

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

```
1.  #!/bin/bash
2.  # Basic if statement
3.
4.  if [ $1 -gt 100 ]
5.  then
6.     echo Hey that\'s a large number.
7.     pwd
8.  fi
9.
10. 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 6 -** The backslash (\ ) in front of the single quote (') is needed as the single quote has a special meaning for bash and we don't want that special meaning. The backslash escapes the special meaning to make it a normal plain single quote again.

- **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.

```
./if_example.sh 15
Fri 22 Jan 2:32:52 2021
./if_example.sh 150
Hey that's a large number.
/home/ryan/bin
Fri 22 Jan 2:32:52 2021

```

It is always good practice to test your scripts with input that covers the different scenarios that are possible.

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 [ ] 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.

```
1.  test 001 = 1
2.  echo $?
3.  1
4.  test 001 -eq 1
5.  echo $?
6.  0
7.  touch myfile
8.  test -s myfile
9.  echo $?
10. 1
11. ls /etc > myfile
12. test -s myfile
13. echo $?
14. 0
15.
```

Let's break it down:

- **Line 1 -** Perform a string based comparison. Test doesn't print the result so instead we check it's exit status which is what we will do on the next line.
- **Line 2 -** The variable **$?** holds the exit status of the previously run command (in this case test). 0 means TRUE (or success). 1 = FALSE (or failure).
- **Line 4 -** This time we are performing a numerical comparison.
- **Line 7 -** Create a new blank file **myfile** (assuming that myfile doesn't already exist).
- **Line 8** - Is the size of **myfile** greater than zero?
- **Line 11 -** Redirect some content into myfile so it's size is greater than zero.
- **Line 12 -** Test the size of **myfile** again. This time it is TRUE.

Indenting

You'll notice that in the **if** statement above we indented the commands that were run if the statement was true. This is referred to as indenting and is an important part of writing good, clean code (in any language, not just Bash scripts). The aim is to improve readability and make it harder for us to make simple, silly mistakes. There aren't any rules regarding indenting in Bash so you may indent or not indent however you like and your scripts will still run exactly the same. I would highly recommend you do indent your code however (especially as your scripts get larger) otherwise you will find it increasingly difficult to see the structure in your scripts.

Nested If statements

Talking of indenting. Here's a perfect example of when it makes life easier for you. You may have as many **if** statements as necessary inside your script. It is also possible to have an if statement inside of another if statement. For example, we may want to analyse a number given on the command line like so:

nested_if.sh

```
1.  #!/bin/bash
2.  # Nested if statements
3.
4.  if [ $1 -gt 100 ]
5.  then
6.     echo Hey that\'s a large number.
7.
8.     if (( $1 % 2 == 0 ))
9.     then
10.       echo And is also an even number.
11.    fi
12. fi
```

Let's break it down:

- **Line 4 -** Perform the following, only if the first command line argument is greater than 100.
- **Line 8 -** This is a light variation on the **if** statement. If we would like to check an expression then we may use the double brackets just like we did for variables.
- **Line 10 -** Only gets run if both if statements are true.

> Yo dawg, I herd you like **if** statements so I put an **if**statement inside your **if** statement.
>
> **Xzibit**

(Xzibit didn't actually say that but I'm sure he would have, had he hosted Pimp My Bash Script.)

> You can nest as many if statements as you like but as a general rule of thumb if you need to nest more than 3 levels deep you should probably have a think about reorganising your logic.

If Else

Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the **else** mechanism.

```
if [ <some test> ]
then
    <commands>
else
    <other commands>
fi
```

Now we could easily read from a file if it is supplied as a command line argument, **else** read from STDIN.

else.sh

```
1.  #!/bin/bash
2.  # else example
3.
4.  if [ $# -eq 1 ]
5.  then
6.     nl $1
7.  else
8.     nl /dev/stdin
9.  fi
```

If Elif Else

Sometimes we may have a series of conditions that may lead to different paths.

```
if [ <some test> ]
then
    <commands>
elif [ <some test> ]
then
    <different commands>
else
    <other commands>
fi
```

For example it may be the case that if you are 18 or over you may go to the party. If you aren't but you have a letter from your parents you may go but must be back before midnight. Otherwise you cannot go.

if_elif.sh

```
1.  #!/bin/bash
2.  # elif statements
3.
4.  if [ $1 -ge 18 ]
5.  then
6.     echo You may go to the party.
7.  elif [ $2 == 'yes' ]
8.  then
9.     echo You may go to the party but be back before
    midnight.
10. else
11.    echo You may not go to the party.
12. fi
```

You can have as many elif branches as you like. The final else is also optional.

Boolean Operations

Sometimes we only want to do something if multiple conditions are met. Other times we would like to perform the action if one of several condition is met. We can accommodate these with **boolean operators**.

- **and** - &&
- **or** - ||

For instance maybe we only want to perform an operation if the file is readable **and** has a size greater than zero.

and.sh

```
1.  #!/bin/bash
2.  # and example
3.
4.  if [ -r $1 ] && [ -s $1 ]
5.  then
6.     echo This file is useful.
7.  fi
```

Maybe we would like to perform something slightly different if the user is either bob or andy.

or.sh

```
1.  #!/bin/bash
2.  # or example
3.
4.  if [ $USER == 'bob' ] || [ $USER == 'andy' ]
5.  then
6.     ls -alh
7.  else
8.     ls
9.  fi
```

Case Statements

Sometimes we may wish to take different paths based upon a variable matching a series of patterns. We could use a series of **if** and **elif**statements but that would soon grow to be unweildly.

Fortunately there is a **case** statement which can make things cleaner. It's a little hard to explain so here are some examples to illustrate:

```
case <variable> in
<pattern 1>)
    <commands>
    ;;
<pattern 2>)
    <other commands>
    ;;
esac
```

Here is a basic example:

case.sh

```
#!/bin/bash
# case example

case $1 in
   start)
      echo starting
      ;;
   stop)
      echo stoping
      ;;
   restart)
      echo restarting
      ;;
   *)
      echo don\'t know
      ;;
esac
```

Let's break it down:

- **Line 4 -** This line begins the **case**mechanism.
- **Line 5** - If $1 is equal to 'start' then perform the subsequent actions. the) signifies the end of the pattern.
- **Line 7** - We identify the end of this set of statements with a double semi-colon (;;). Following this is the next case to consider.
- **Line 14** - Remember that the test for each case is a pattern. The *represents any number of any character. It is essentialy a catch all if for if none of the other cases match. It is not necessary but is often used.
- **Line 17** - esac is case backwards and indicates we are at the end of the case statement. Any other statements after this will be executed normally.

```
1. ./case.sh start
2. starting
3. ./case.sh restart
4. restarting
5. ./case.sh blah
6. don't know
7.
```

Now let's look at a slightly more complex example where patterns are used a bit more.

disk_useage.sh

```
1. #!/bin/bash
2. # Print a message about disk useage.
3.
4. space_free=$( df -h | awk '{ print $5 }' | sort -n |
   tail -n 1 | sed 's/%//' )
5.
6. case $space_free in
7.    [1-5]*)
```

```
 8.       echo Plenty of disk space available
 9.       ;;
10.    [6-7]*)
11.       echo There could be a problem in the near future
12.       ;;
13.    8*)
14.       echo Maybe we should look at clearing out old
    files
15.       ;;
16.    9*)
17.       echo We could have a serious problem on our hands
    soon
18.       ;;
19.    *)
20.       echo Something is not quite right here
21.       ;;
22. esac
```

Summary

if
Perform a set of commands if a test is true.

else
If the test is not true then perform a different set of commands.

elif
If the previous test returned false then try this one.

&&
Perform the and operation.

||
Perform the or operation.

case
Choose a set of commands to execute depending on a string matching a particular pattern.

Indenting
Indenting makes your code much easier to read. It get's increasingly important as your Bash scripts get longer.

Planning
Now that your scripts are getting a little more complex you will probably want to spend a little bit of time thinking about how you structure them before diving in.