Chapter 5. Flow Control / Conditions & loops

if constructs

The most compact syntax of the **if** command is:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

Combining expressions

Operation	Effect
[! EXPR]	True if **EXPR** is false.
[(EXPR)]	Returns the value of **EXPR**. This may be used to override the normalprecedence of operators.
[EXPR1 -a EXPR2]	True if both **EXPR1** and **EXPR2** are true.
[EXPR1 -o EXPR2]	True if either **EXPR1** or **EXPR2** is true.

Example

```
####True If /var/log/messages file exists #######

 # if [ -f /var/log/messages ]
 > then
 > echo "/var/log/messages exists"
 > fi
 /var/log/messages exists

####True If /var/log/non-existing file dose not exists ####

# if [ ! -f /var/log/non-existing ]; then echo "File is not here"; fi
File is not here
```

Expressions used with if

Primary	Meaning
[-a FILE]	True if FILEexists.
[-b FILE]	True if FILEexists and is a block-special file.
[-c FILE]	True if FILEexists and is a character-special file.
[-d FILE]	True if FILEexists and is a directory.
[-e FILE]	True if FILEexists.
[-f FILE]	True if FILEexists and is a regular file.

Primary	**Meaning**
[-g FILE]	True if FILEexists and its SGID bit is set.
[-h FILE]	True if FILEexists and is a symbolic link.
[-k FILE]	True if FILEexists and its sticky bit is set.
[-p FILE]	True if FILEexists and is a named pipe (FIFO).
[-r FILE]	True if FILEexists and is readable.
[-s FILE]	True if FILEexists and has a size greater than zero.
[-t FD]	True if file descriptor FDis open and refers to a terminal.
[-u FILE]	True if FILEexists and its SUID (set user ID) bit is set.
[-w FILE]	True if FILEexists and is writable.
[-x FILE]	True if FILEexists and is executable.
[-O FILE]	True if FILEexists and is owned by the effective user ID.
[-G FILE]	True if FILEexists and is owned by the effective group ID.
[-L FILE]	True if FILEexists and is a symbolic link.
[-N FILE]	True if FILEexists and has been modified since it was last read.
[-S FILE]	True if FILEexists and is a socket.
[FILE1 -nt FILE2]	True if FILE1has been changed more recently than FILE2, or if FILE1 exists and FILE2does not.
[FILE1 -ot FILE2]	True if FILE1is older than FILE2, or is FILE2exists and FILE1 does not.
[FILE1 -ef FILE2]	True if FILE1and FILE2refer to the same device and inode numbers.
[-o OPTIONNAME]	True if shell option "OPTIONNAME" is enabled.
[-z STRING]	True of the length if "STRING" is zero.
[-n STRING] or [STRING]	True if the length of "STRING" is non-zero.
[STRING1 == STRING2]	True if the strings are equal. "=" may be used instead of "==" for strictPOSIX compliance.
[STRING1 != STRING2]	True if the strings are not equal.
[STRING1 < STRING2]	True if "STRING1" sorts before "STRING2" lexicographically in thecurrent locale.
[STRING1 > STRING2]	True if "STRING1" sorts after "STRING2" lexicographically in thecurrent locale.
[ARG1 OP ARG2]	"OP" is one of -eq, -ne, -lt, -le, -gtor -ge. These arithmetic binaryoperators return true if "ARG1" is equal to, not equal to, less than, lessthan or equal to, greater than,

Testing exit status

```
# if [ $? -eq 0 ]
>   then
>   echo 'That was a good job!'
>   fi
That was a good job!

# if ! grep vidya test
>   then
>   echo "name : vidya not exists in file test"
>   fi
name : vidya not exists in file test

# if grep sandip test
>   then
>   echo "name : sandip exists in file test"
>   fi
sandip Wikram subhash
Wikram sandip subhash
Wikram2 1sandip subhash1
Wikram2 1sandip subhash1
Wikram sandip subhash
Wikram2 1sandip subhash1
Wikram2 1sandip subhash1
name : sandip exists in file test
```

Particularly for the numerical

```
# num=50
# if [ $num -gt 20 ]
>    then
>    echo "num is greater than 20"
>    fi
num is greater than 20

# if [ $num > 20 ]; then echo "num is greater than 20"; fi
num is greater than 20
```

-gt greater than
-lt less than
-eq equal to
-le less than or equal to
-ge greater than or equal to

For string comparison

```
# if [ sandip == sandip ]; then echo "strings are equal"; fi
strings are equal

# if [ sandip -eq sandip ]; then echo "strings are equal"; fi
-bash: [: sandip: integer expression expected

# if [ "sandip raaje" == "sandip raaje" ]; then echo "strings are
equal"; fi
strings are equal

# if [ "sandip raaje" != "sandip" ]; then echo "strings are not
equal"; fi
strings are not equal

# if [[ "sandip raaje" == sandip* ]]; then echo "strings are equal";
fi
strings are equal
```

if/then/else constructs

The most compact syntax of the if command is:

if TEST-COMMANDS
then
do this and this
else
do that and that
fi

examples

```
# if [[ "sandip raaje" == sandip* ]]
>     then
>     echo "strings matches"
>     else
>     echo "strings dont matches"
>     fi
strings matches
```

if/then/elif/else constructs

```
# name1=satya
# if [ "$name1" == Wikram ]
>     then
>     echo "this is Wikram"
>     elif [ "$name1" == sandip ]
>     then
>     echo "this is sandip"
>     else
>     echo "this is neither sandip or Wikram"
>     fi
this is neither sandip or Wikram
```

Nested if statements

```
#!/bin/bash
# This script will test if we're in a leap year or not.
year=`date +%Y`
if [ $[$year % 400] -eq "0" ]
   then
        echo "This is a leap year. February has 29 days."
elif [ $[$year % 4] -eq 0 ]
   then
   if [ $[$year % 100] -ne 0 ]
      then
        echo "This is a leap year, February has 29 days."
   else
        echo "This is not a leap year. February has 28 days."
   fi
else
   echo "This is not a leap year. February has 28 days."
Fi
```

Logical Operators

AND

```
if [ condition ] && [ condition ]
if command && [ condition ]
if [ $STATUS -ne 200 ] -a [[ "$STRING" != "$VALUE" ]]
if [ "${STATUS}" -ne 100 -a "${STRING}" = "${VALUE}" ]
if [ "${STATUS}" -ne 100 ] && [ "${STRING}" = "${VALUE}" ]
```

OR

```
if [ $STATUS -ne 200 ] -o [[ "$STRING" != "$VALUE" ]];
```

Composite

```
if ([ $num -ge 10 ] && [ $num -le 20 ]) || ([ $num -ge 100 ] && [ $num
-le 200 ])
```

example

```
#!/bin/bash
# We will check if the number is even and also divisible by 10.
num=50

if [ $((num % 2)) == 0 ] && [ $((num % 10)) == 0 ];
then
    echo "$num is even and also divisible 10."
Fi
```

Loops

for: the for loop is ideal for working with arguments on the command line and with sets of files (e.g., all files in a given directory).

```
for name [in list]
do
statements that can use
$name...
done
```

example

```
IFS=:
for dir in $PATH
do
ls -ld $dir
done
```

while: A while loop is a statement that iterates over a block of code till the condition specified is evaluated to false.

Some good examples are here : https://www.geeksforgeeks.org/bash-scripting-while-loop/

```
while [ condition ];
do
    # statements
    # commands
done

while:
do
  echo "An Infinite loop"
  # statements
  # commands
done
```

example

```
# counter=0

# while [ $counter -le 5 ] # Waiting to be false
> do
> echo Counter: $counter
> ((counter++))
> done
```

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

until: The until loop is used to execute a given set of commands as long as the given condition evaluates to false.

```
until [CONDITION]
do
  [COMMANDS]
Done
```

example

```
# until [ $counter -gt 5 ] # Waiting to be true
> do
>   echo Counter: $counter
>   ((counter++))
> done
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

case: Bash shell case statement is like switch statement in C. It can be used to test simple values like integers and characters. bash's case construct lets you test strings against patterns that can contain wildcard characters.

```
case expression
in
pattern1
)
statements ;;
pattern2
)
statements ;;
...
esac
```

example

```
#!/bin/bash
echo "Which color do you like best?"
echo "1 - Blue"
echo "2 - Red"
echo "3 - Yellow"
echo "4 - Green"
echo "5 - Orange"
read color;
case $color in
  1) echo "Blue is a primary color.";;
  2) echo "Red is a primary color.";;
  3) echo "Yellow is a primary color.";;
  4) echo "Green is a secondary color.";;
  5) echo "Orange is a secondary color.";;
  *) echo "This color is not available. Please choose a different
one.";;
esac
```

Can you use numerical ranges in case?

Hint: it works with pattern matching

select: Select command is used in scripts for ***menu creation***. Different types of menu generation tasks can be implemented with the select command such as: creating menu-based director list, creating a menu from file content etc. By default this is infinite loop until you place break condition within iteration.

```
select name [ in data_list ]
do
statement1
Statement2
Statement3
done
```

example

```
#!/bin/bash
echo "Which Operating System do you like?"

# Operating system names are used here as a data source
select os in Ubuntu LinuxMint Windows8 Windows7 WindowsXP
do
if  [ ! -z $os ]
then
  echo "Its $os"
else
  echo "OS not found"
  break
fi

done
```

Bonus stuff

sleep: sleep is a command-line utility that allows you to suspend the calling process for a specified time. In other words, the sleep command pauses the execution of the next command for a given number of seconds.

s - seconds (default)
m - minutes
h - hours
d - days

Docs : https://linuxize.com/post/how-to-use-linux-sleep-command-to-pause-a-bash-script/

example

```
#!/bin/bash
# start time
date +"%H:%M:%S"

# sleep for 5 seconds
sleep 5

# end time
date +"%H:%M:%S"
```

example

```
sleep 2m 30s
sleep 0.5
```

wait: The bash wait command is a Shell command that waits for background running processes to complete and returns the exit status. Unlike the sleep command, which waits for a specified time, the wait command waits for all or specific background tasks to finish

Docs: https://linuxize.com/post/bash-wait/

example

```
#!/bin/bash
sleep 10 &
sleep 15 &
sleep 25 &

wait
echo "I was waiting for pre-sleeping child processes."
```

exec: The command following exec replaces the current shell. This means no subshell is created and the current process is replaced with this new command.

Docs : https://www.baeldung.com/linux/exec-command-in-shell-script

example

```
#!/bin/bash
script_log="/tmp/log_`date +%F`.log"
exec 1>>$script_log
exec 2>&1
echo "This will be written into log file rather than terminal.."
echo "This too.."
```

Let's now run an example to read from the file we created:

```
# cat input_csv
SNo,Quantity,Price,Value
1,2,20,40
2,5,10,50
3,1,70,70
```

```
#!/bin/bash
exec < input_csv
read row1
echo -n "The contents of first line are: "
echo $row1
echo -n "The contents of second line are: "
read row2
echo $row2
```

trap:

Trap allows you to catch signals and execute code when they occur. Signals are asynchronous notifications that are sent to your script when certain events occur.
Bash also provides a psuedo-signal called "EXIT", which is executed when your script exits; this can be used to make sure that your script executes some cleanup on exit.

Bash trap Syntax & Options
-p - Displays signal commands.
-l - Prints a list of all the signals and their numbers.
trap [-options] “piece of code” [signal name or value]

Docs: https://www.geeksforgeeks.org/shell-scripting-bash-trap-command/

Example

```
trap "echo Hello World" SIGINT
```

Example

```
trap "echo The script is terminated; exit" SIGINT
while true
do
    echo Test
    sleep 1
done
```

Example

```
$ cat trap_ex_00.sh
tempfile=/tmp/tmpdata
rm -f $tempfile

$ sh -x trap_ex_00.sh
+ tempfile=/tmp/tmpdata
+ rm -f /tmp/tmpdata
rm: cannot remove '/tmp/tmpdata': Is a directory
$ echo $?
1
```

Example

```
$ cat trap_ex_01.sh
tempfile=/tmp/tmpdata
trap "rm -f $tempfile" EXIT


$ sh -x trap_ex_01.sh
+ tempfile=/tmp/tmpdata
+ trap 'rm -f /tmp/tmpdata' EXIT
+ rm -f /tmp/tmpdata
rm: cannot remove '/tmp/tmpdata': Is a directory
$ echo $?
0
```