### How to Create Your First Bash Script

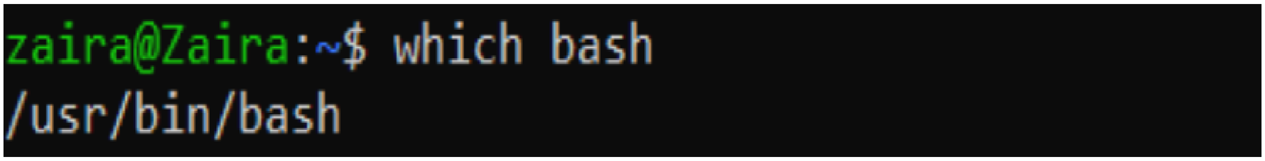
Let's create a simple script in bash that outputs `Hello World`.

### Create a file named hello_world.sh

```
touch hello_world.sh
```

### Find the path to your bash shell.

```
which bash
```



```
zaira@Zaira:~$ which bash
/usr/bin/bash
```

In my case, the path is `/usr/bin/bash` and I will include this in the shebang.

### Write the command.

We will `echo` "hello world" to the console.

Our script will look something like this:

```
#! /usr/bin/bash
echo "Hello World"
```

Edit the file `hello_world.sh` using a text editor of your choice and add the above lines in it.

**Provide execution rights to your user.**

Modify the file permissions and allow execution of the script by using the command below:

```
chmod u+x hello_world.sh
```

chmod modifies the existing rights of a file for a particular user. We are adding +x to user u.

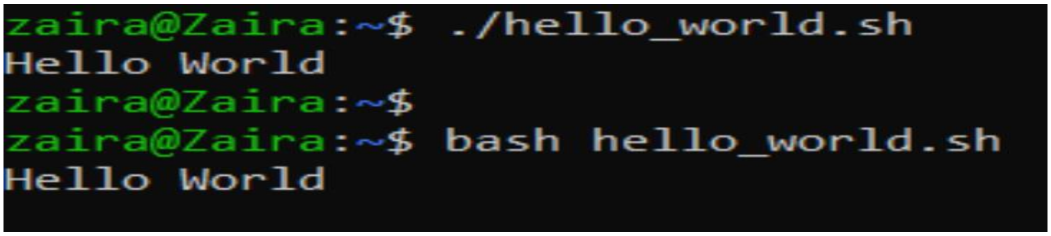
**Run the script.**

You can run the script in the following ways:

```
./hello_world.sh
```

```
bash hello_world.sh.
```

### Here's the output:



```
zaira@Zaira:~$ ./hello_world.sh
Hello World
zaira@Zaira:~$
zaira@Zaira:~$ bash hello_world.sh
Hello World
```

*Two ways to run scripts*

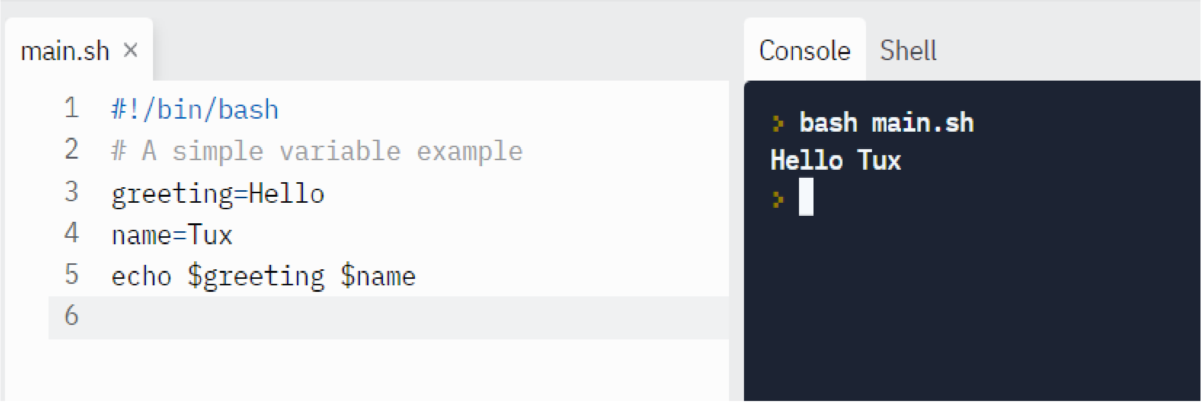
### The Basic Syntax of Bash Scripting

Just like any other programming language, bash scripting follows a set of rules to create programs understandable by the computer. In this section, we will study the syntax of bash scripting.

### How to define variables

We can define a variable by using the syntax `variable_name=value`. To get the value of the variable, add $ before the variable.

```
#!/bin/bash
# A simple variable example
greeting=Hello
name=Tux
echo $greeting $name
```



```
#!/bin/bash
# A simple variable example
greeting=Hello
name=Tux
echo $greeting $name
```

```
› bash main.sh
Hello Tux
›
```

Tux is also the name of the Linux mascot, the penguin.

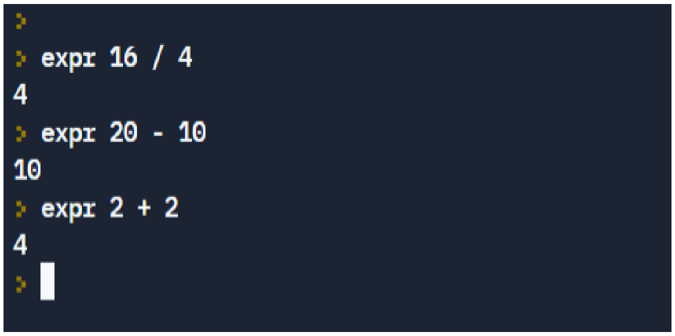
*Hi, I am Tux.*

**Arithmetic Expressions**

Below are the operators supported by bash for mathematical calculations:

| Operator | Usage |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| ** | exponentiation |
| % | modulus |

Let's run a few examples.



```
>
> expr 16 / 4
4
> expr 20 - 10
10
> expr 2 + 2
4
>
```

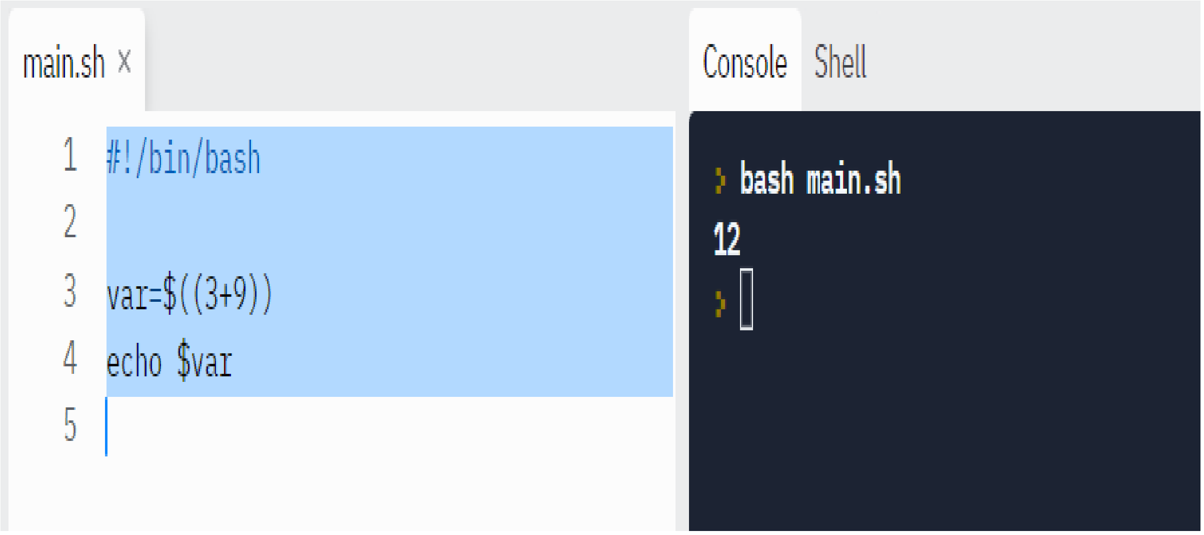
*Note the spaces, these are part of the syntax*

Numerical expressions can also be calculated and stored in a variable using the syntax below:

```
var=$((expression))
```

Let's try an example.

```
#!/bin/bash

var=$((3+9))
echo $var
```



```
#!/bin/bash

var=$((3+9))
echo $var
```

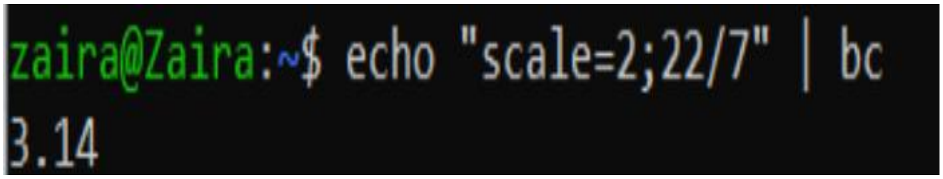
```
› bash main.sh
12
›
```

Fractions are not correctly calculated using the above methods and truncated.

For **decimal calculations**, we can use `bc` command to get the output to a particular number of decimal places. `bc` (Bash Calculator) is a command line calculator that supports calculation up to a certain number of decimal points.

```
echo "scale=2;22/7" | bc
```

Where `scale` defines the number of decimal places required in the output.



```
zaira@Zaira:~$ echo "scale=2;22/7" | bc
3.14
```

*Getting output to 2 decimal places*

### How to read user input

Sometimes you'll need to gather user input and perform relevant operations.

In bash, we can take user input using the `read` command.

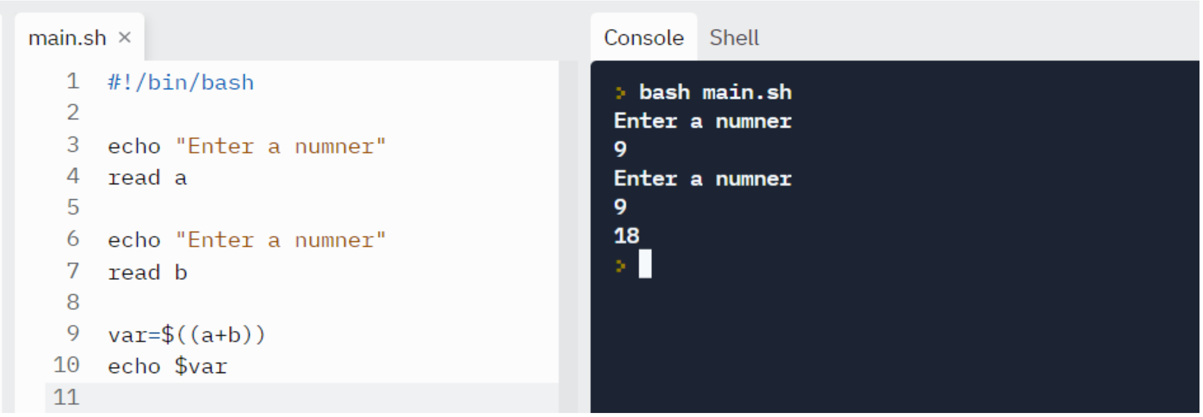
```
read variable_name
```

To prompt the user with a custom message, use the `-p` flag.

```
read -p "Enter your age" variable_name
```

## Example:

```
#!/bin/bash

echo "Enter a numner"
read a

echo "Enter a numner"
read b

var=$((a+b))
echo $var
```



```bash
#!/bin/bash

echo "Enter a numner"
read a

echo "Enter a numner"
read b

var=$((a+b))
echo $var
```

```
> bash main.sh
Enter a numner
9
Enter a numner
9
18
> 
```

### Numeric Comparison logical operators

Comparison is used to check if statements evaluate to `true` or `false`. We can use the below shown operators to compare two statements:

| Operation | Syntax | Explanation |
|---|---|---|
| Equality | num1 -eq num2 | is num1 equal to num2 |
| Greater than equal to | num1 -ge num2 | is num1 greater than equal to num2 |
| Greater than | num1 -gt num2 | is num1 greater than num2 |
| Less than equal to | num1 -le num2 | is num1 less than equal to num2 |
| Less than | num1 -lt num2 | is num1 less than num2 |
| Not Equal to | num1 -ne num2 | is num1 not equal to num2 |

## Syntax:

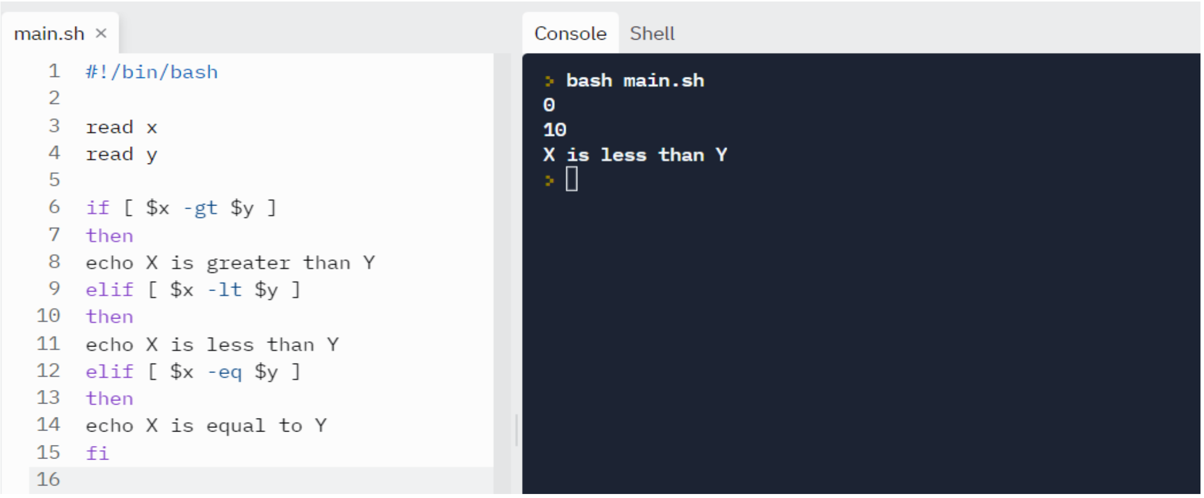
```
if [ conditions ]
    then
         commands
fi
```

## Example:

Let's compare two numbers and find their relationship:

```
read x
read y

if [ $x -gt $y ]
then
echo X is greater than Y
elif [ $x -lt $y ]
then
echo X is less than Y
elif [ $x -eq $y ]
then
echo X is equal to Y
fi
```

## Output:



```
#!/bin/bash

read x
read y

if [ $x -gt $y ]
then
echo X is greater than Y
elif [ $x -lt $y ]
then
echo X is less than Y
elif [ $x -eq $y ]
then
echo X is equal to Y
fi

```

```
> bash main.sh
0
10
X is less than Y
> 
```

### Conditional Statements (Decision Making)

Conditions are expressions that evaluate to a boolean expression (`true` or `false`). To check conditions, we can use `if`, `if-else`, `if-elif-else` and nested conditionals.

The structure of conditional statements is as follows:

- `if...then...fi` statements
- `if...then...else...fi` statements
- `if..elif..else..fi`
- `if..then..else..if..then..fi..fi..` (Nested Conditionals)

**Syntax:**

```
if [[ condition ]]
then
    statement
elif [[ condition ]]; then
    statement
else
    do this by default
fi
```

To create meaningful comparisons, we can use AND `-a` and OR `-o` as well.

The below statement translates to: If a is greater than 40 and b is less than 6.

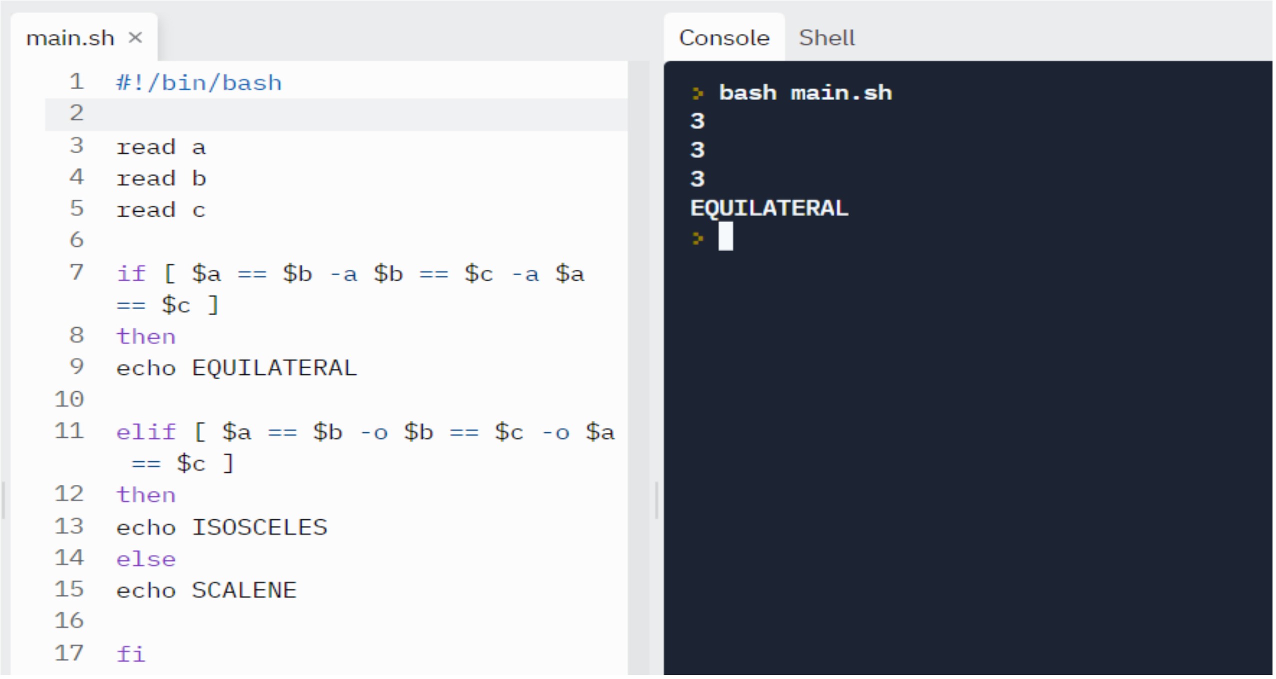
```
if [ $a -gt 40 -a $b -lt 6 ]
```

**Example:** Let's find the triangle type by reading the lengths of its sides.

```
read a
read b
read c

if [ $a == $b -a $b == $c -a $a == $c ]
then
echo EQUILATERAL

elif [ $a == $b -o $b == $c -o $a == $c ]
then
echo ISOSCELES
else
echo SCALENE

fi
```

**Output:**

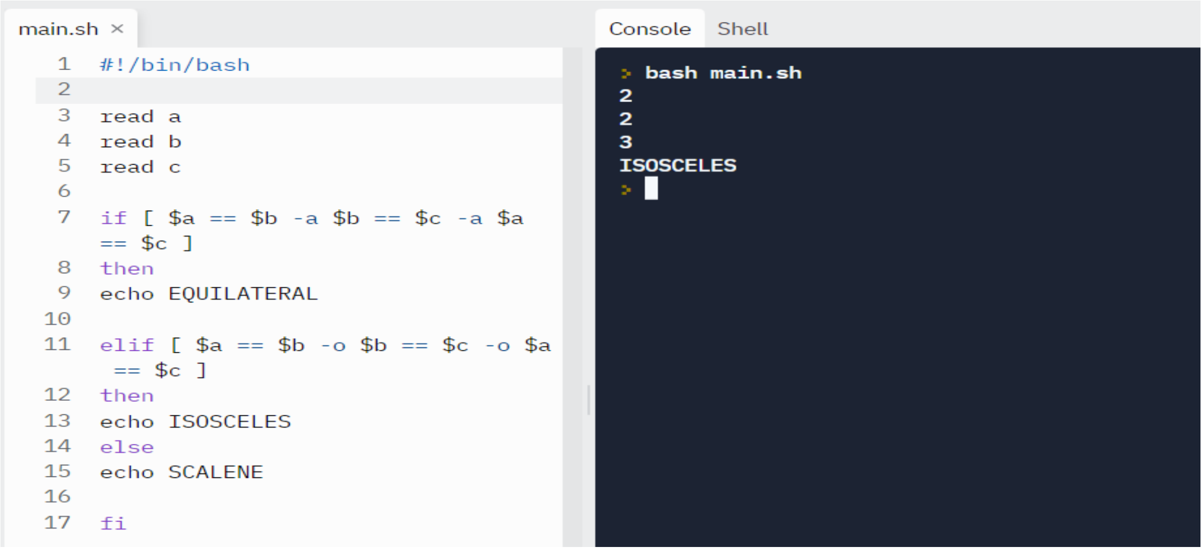
Test case #1



```bash
#!/bin/bash

read a
read b
read c

if [ $a == $b -a $b == $c -a $a == $c ]
then
echo EQUILATERAL

elif [ $a == $b -o $b == $c -o $a == $c ]
then
echo ISOSCELES
else
echo SCALENE

fi
```

```
> bash main.sh
3
3
3
EQUILATERAL
> 
```

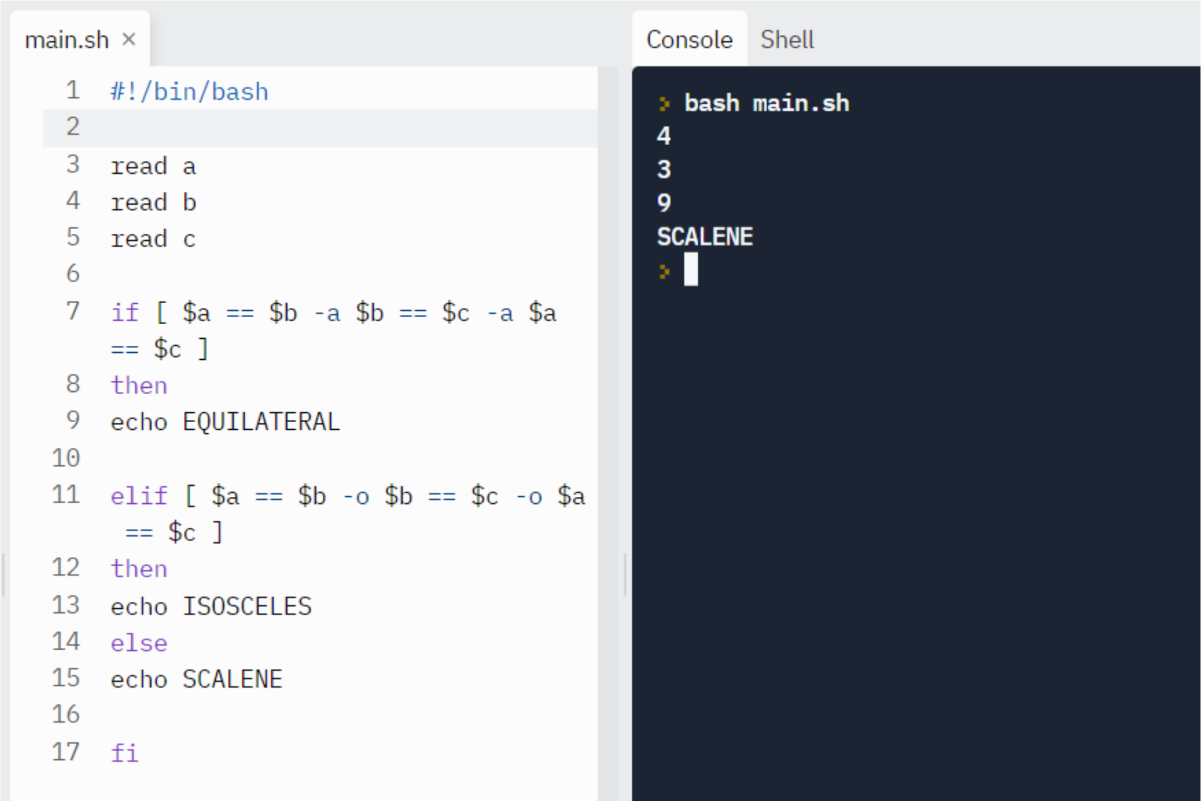
## Test case #2



```bash
#!/bin/bash

read a
read b
read c

if [ $a == $b -a $b == $c -a $a == $c ]
then
echo EQUILATERAL

elif [ $a == $b -o $b == $c -o $a == $c ]
then
echo ISOSCELES
else
echo SCALENE

fi
```

```
> bash main.sh
2
2
3
ISOSCELES
> 
```

## Test case #3



```bash
#!/bin/bash

read a
read b
read c

if [ $a == $b -a $b == $c -a $a == $c ]
then
echo EQUILATERAL

elif [ $a == $b -o $b == $c -o $a == $c ]
then
echo ISOSCELES
else
echo SCALENE

fi
```

```
› bash main.sh
4
3
9
SCALENE
›
```

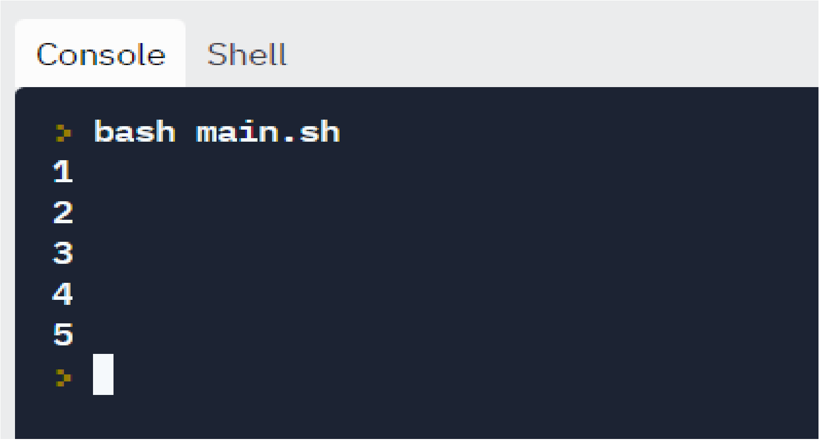
### Looping and skipping

For loops allow you to execute statements a specific number of times.

### Looping with numbers:

In the example below, the loop will iterate 5 times.

```bash
#!/bin/bash

for i in {1..5}
do
    echo $i
done
```

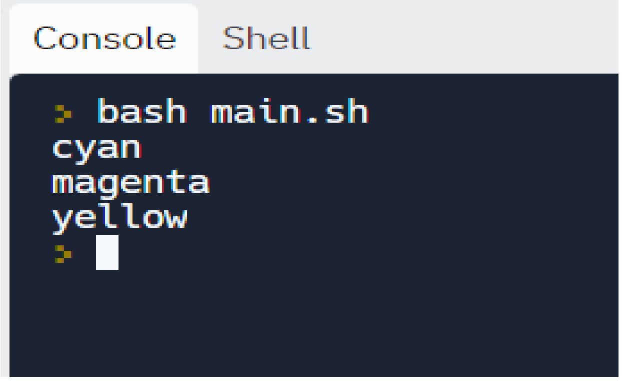


```
> bash main.sh
1
2
3
4
5
> 
```

**Looping with strings:**

We can loop through strings as well.

```
#!/bin/bash

for X in cyan magenta yellow
do
     echo $X
done
```



```
> bash main.sh
cyan
magenta
yellow
> 
```

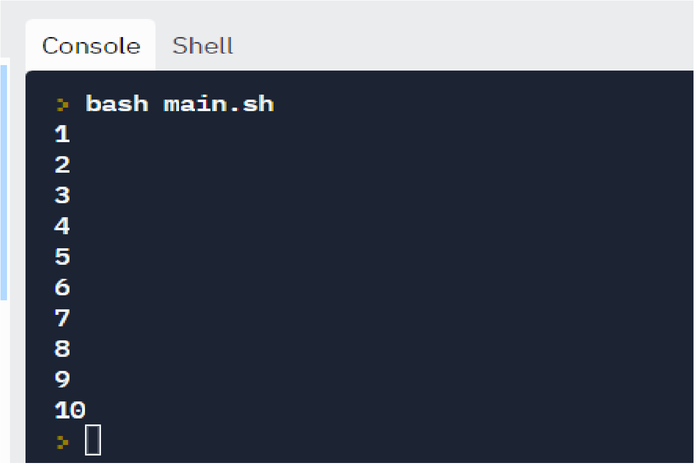
**While loop**

While loops check for a condition and loop until the condition remains `true`. We need to provide a counter statement that increments the counter to control loop execution.

In the example below, `(( i += 1 ))` is the counter statement that increments the value of `i`.

## Example:

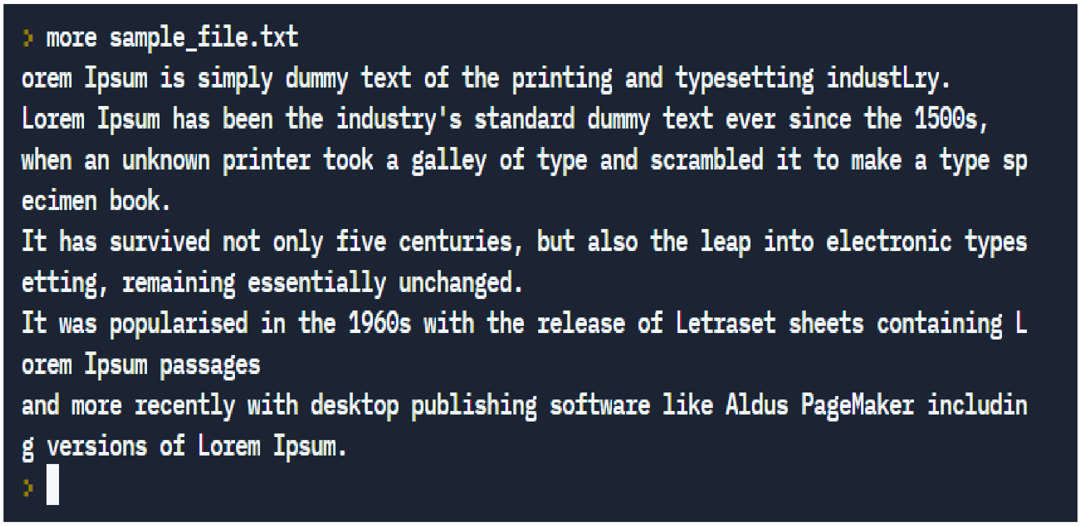
```
#!/bin/bash
i=1
while [[ $i -le 10 ]] ; do
   echo "$i"
  (( i += 1 ))
done
```



```
> bash main.sh
1
2
3
4
5
6
7
8
9
10
> 
```

### Reading files

Suppose we have a file `sample_file.txt` as shown below:

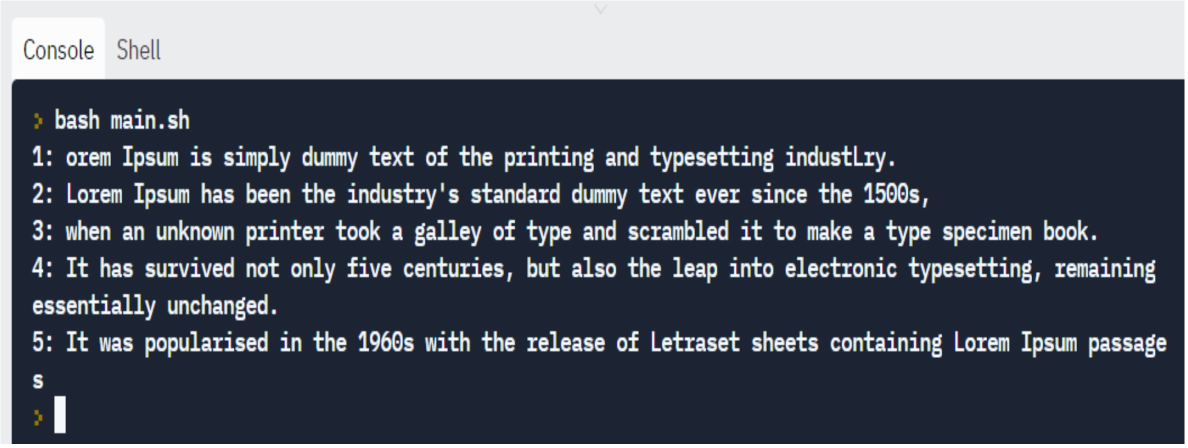


```
› more sample_file.txt
orem Ipsum is simply dummy text of the printing and typesetting industLry.
Lorem Ipsum has been the industry's standard dummy text ever since the 1500s,
when an unknown printer took a galley of type and scrambled it to make a type sp
ecimen book.
It has survived not only five centuries, but also the leap into electronic types
etting, remaining essentially unchanged.
It was popularised in the 1960s with the release of Letraset sheets containing L
orem Ipsum passages
and more recently with desktop publishing software like Aldus PageMaker includin
g versions of Lorem Ipsum.
› 
```

We can read the file line by line and print the output on the screen.

```
#!/bin/bash

LINE=1

while read -r CURRENT_LINE
    do
        echo "$LINE: $CURRENT_LINE"
    ((LINE++))
done < "sample_file.txt"
```

## Output:



```
› bash main.sh
1: orem Ipsum is simply dummy text of the printing and typesetting industLry.
2: Lorem Ipsum has been the industry's standard dummy text ever since the 1500s,
3: when an unknown printer took a galley of type and scrambled it to make a type specimen book.
4: It has survived not only five centuries, but also the leap into electronic typesetting, remaining
essentially unchanged.
5: It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passage
s
› 
```

*Lines with line number printed*

### How to execute commands with back ticks

If you need to include the output of a complex command in your script, you can write the statement inside back ticks.

**Syntax:**

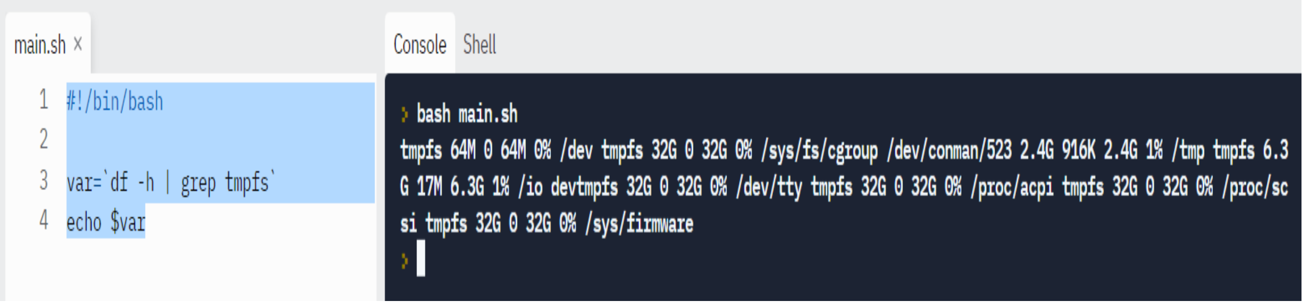
var= commands

**Example**: Suppose we want to get the output of a list of mountpoints with `tmpfs` in their name. We can craft a statement like this: `df -h | grep tmpfs`.

To include it in the bash script, we can enclose it in back ticks.

```
#!/bin/bash

var=`df -h | grep tmpfs`
echo $var
```

Output:



```
#!/bin/bash

var=`df -h | grep tmpfs`
echo $var
```

```
› bash main.sh
tmpfs 64M 0 64M 0% /dev tmpfs 32G 0 32G 0% /sys/fs/cgroup /dev/conman/523 2.4G 916K 2.4G 1% /tmp tmpfs 6.3
G 17M 6.3G 1% /io devtmpfs 32G 0 32G 0% /dev/tty tmpfs 32G 0 32G 0% /proc/acpi tmpfs 32G 0 32G 0% /proc/sc
si tmpfs 32G 0 32G 0% /sys/firmware
›
```

### How to get arguments for scripts from the command line

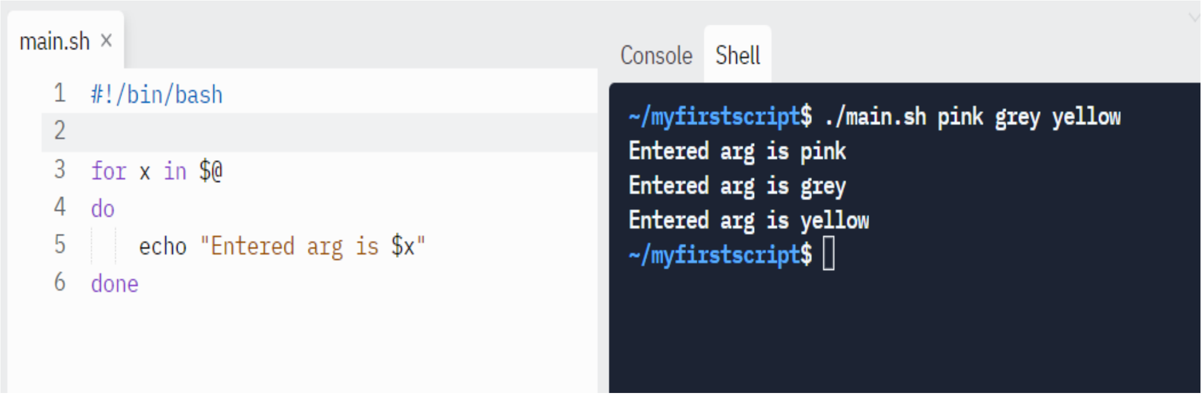
It is possible to give arguments to the script on execution.

$@ represents the position of the parameters, starting from one.

```
#!/bin/bash

for x in $@
do
    echo "Entered arg is $x"
done
```

Run it like this:

```
./script arg1 arg2
```



```
#!/bin/bash

for x in $@
do
    echo "Entered arg is $x"
done
```

```
~/myfirstscript$ ./main.sh pink grey yellow
Entered arg is pink
Entered arg is grey
Entered arg is yellow
~/myfirstscript$ 
```

### How to Automate Scripts by Scheduling via cron Jobs

Cron is a job scheduling utility present in Unix like systems. You can schedule jobs to execute daily, weekly, monthly or in a specific time of the day. Automation in Linux heavily relies on cron jobs.

Below is the syntax to schedule crons:

```
# Cron job example
* * * * * sh /path/to/script.sh
```

Here, * represents minute(s) hour(s) day(s) month(s) weekday(s), respectively.

Below are some examples of scheduling cron jobs.

| SCHEDULE | SCHEDULED VALUE |
|---|---|
| 5 0 * 8 * | At 00:05 in August. |
| 5 4 * * 6 | At 04:05 on Saturday. |

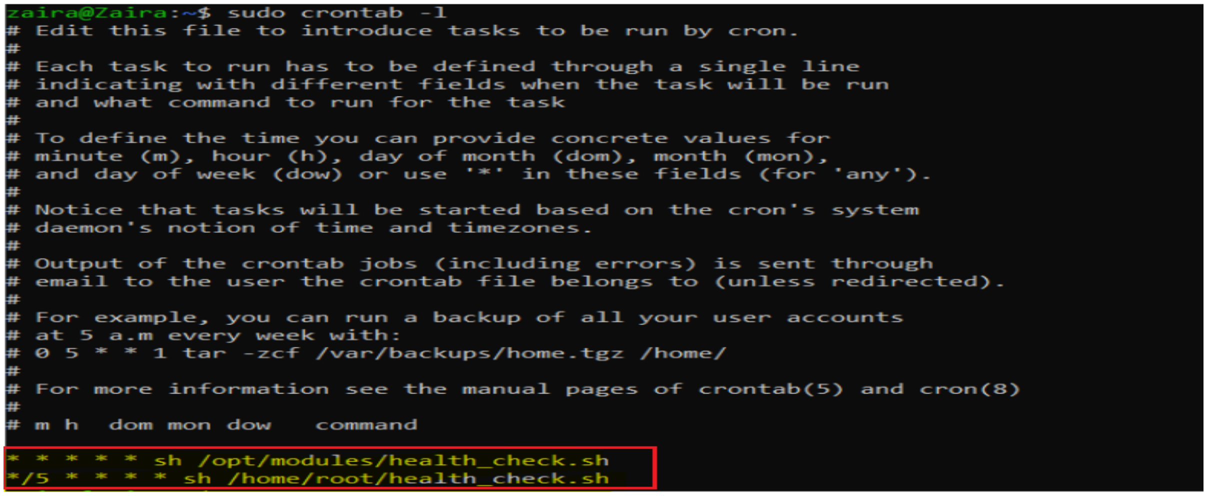
| | |
|---|---|
| 0 22 * * 1-5 | At 22:00 on every day-of-week from Monday through Friday. |

You can learn about cron in detail in this blog post.

### How to Check Existing Scripts in a System

#### Using crontab

`crontab -l` lists the already scheduled scripts for a particular user.



```
zaira@Zaira:~$ sudo crontab -l
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command

* * * * * sh /opt/modules/health_check.sh
*/5 * * * * sh /home/root/health_check.sh
```

*My scheduled scripts*

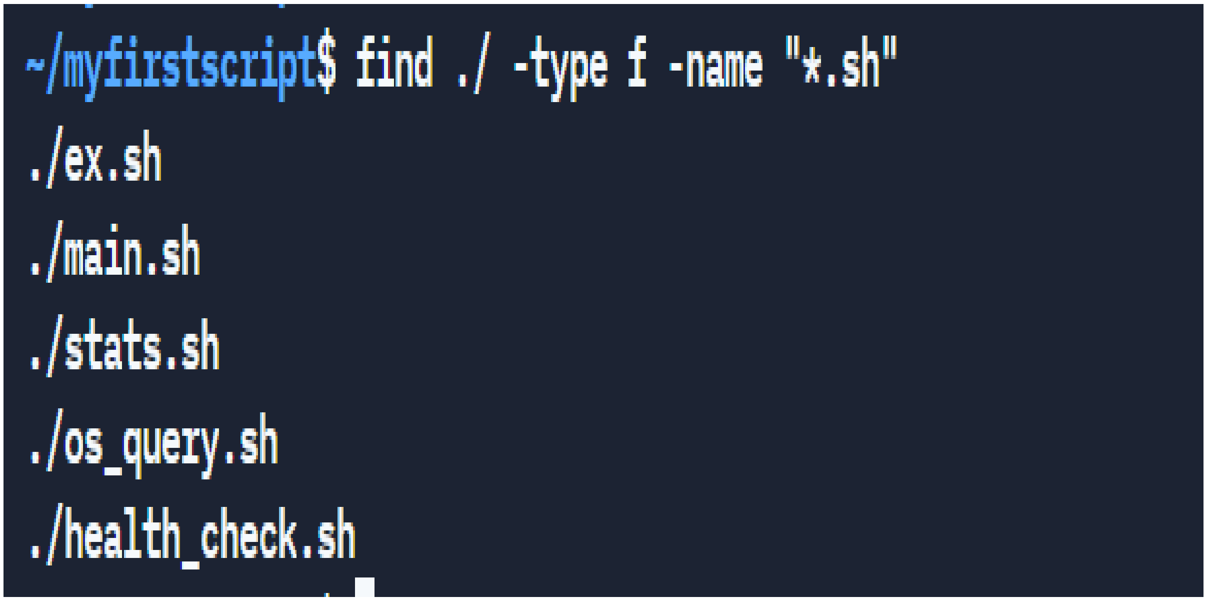
#### Using the find command

The `find` command helps to locate files based on certain patterns. As most of the scripts end with `.sh`, we can use the find script like this:

```
find . -type f -name "*.sh"
```

Where,

- `.` represents the current directory. You can change the path accordingly.
- `-type f` indicates that the file type we are looking for is a text based file.
- `*.sh` tells to match all files ending with `.sh`.



```
~/myfirstscript$ find ./ -type f -name "*.sh"
./ex.sh
./main.sh
./stats.sh
./os_query.sh
./health_check.sh
```