SHELL SCRIPTING

UNIX shell program interprets user commands which are directly entered by the user or which can be read from a file called the shell script or shell programming.

*Shell scrips are interpreted not compiled.

```
root@vm:/home/lucky# cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/bin/tmux
/usr/bin/screen
```

Bash stands for bourne again shell

Bash is an improved version of bourne shell(sh) and now-a-days it is standard GNU shell which is intuitive and flexible

To know where bash is located

```
root@vm:/home/lucky# which bash
/usr/bin/bash
```

#! is called shebang or hashbang

```
root@vm:/home/lucky# cat >> hello.sh
#! /usr/bin/bash
echo "hello world"
^C

root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash
echo "hello world"
root@vm:/home/lucky# ./hello.sh
hello world
root@vm:/home/lucky# chmod u-x hello.sh
root@vm:/home/lucky# ls -l
total 4
-rw-r--r-- 1 root root 36 Sep 10 07:50 hello.sh
root@vm:/home/lucky# ./hello.sh
bash: ./hello.sh: Permission denied
root@vm:/home/lucky# chmod u+x hello.sh
root@vm:/home/lucky# ./hello.sh
hello world
```

Commands are the lines of code which are not executed by our script helpful to know about our code.#

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash
#this is a comment
echo "hello world" # this is  a comment
root@vm:/home/lucky# ./hello.sh
hello world
```

Variables are containers which store data inside it.

In the Unix system, there are 2 types of data

1. System variable

2. User-defined variable

System variables are created and maintained by linux or unix operating system.

These are the predefined variable which are defined by your operating system.

These are defined in capital cases.

User defined variables are the variables which are defined by user.

Generally these are in lowercase letters…but there is no rule to written in lowercase…we can write in uppercase as well

****System variable demo****

```
root@vm:/home/lucky# cat hello.sh
#!/usr/bin/bash
echo "hello world"




echo $BASH
echo $BASH_VERSION
echo $HOME
echo $PWD

root@vm:/home/lucky# ./hello.sh
hello world
/usr/bin/bash
5.0.17(1)-release
/root
/home/lucky
```

User variable demo

```
root@vm:/home/lucky# cat hello.sh
#!/usr/bin/bash
echo "hello world"




echo $BASH
echo $BASH_VERSION
echo $HOME
echo $PWD

name=sravani
echo the name is $name


root@vm:/home/lucky# ./hello.sh
hello world
/usr/bin/bash
5.0.17(1)-release
/root
/home/lucky
the name is sravani
```

```
root@vm:/home/lucky# cat hello.sh
#!/usr/bin/bash
echo "hello world"




echo shell name is  $BASH
echo our version is $BASH_VERSION
echo our home directory is $HOME
echo our current working directory is $PWD

name=sravani
echo the name is $name

10val =10
echo $10val


root@vm:/home/lucky# ./hello.sh
hello world
shell name is /usr/bin/bash
our version is 5.0.17(1)-release
our home directory is /root
our current working directory is /home/lucky
the name is sravani
./hello.sh: line 15: 10val: command not found
0val
```

Here there is an error … Starting numerical cannot be taken as a variable name..

```
lucky@vm1:~$ sudo su
root@vm1:/home/lucky# touch hello.sh
root@vm1:/home/lucky# vi hello.sh
"hello.sh" 4L, 34C written
root@vm1:/home/lucky# ./hello.sh
bash: ./hello.sh: Permission denied
root@vm1:/home/lucky# chmod u+x hello.sh
root@vm1:/home/lucky# ./hello.sh
10
```

How to take input from the user????

Actually to take any input from the user ..use this command

Read variable name

Whatever value will be entered from the user ..that value will be stored in that variable name.

```
root@vm1:/home/lucky# cat hello.sh
#!/usr/bin/bash
val=10
echo $val


echo "Entered name:"
read name
echo "Entered name:"$name

root@vm1:/home/lucky# ./hello.sh
10
Entered name:
lucky
Entered name:lucky
```

How to take multiple inputs from the user ??

```
root@vm1:/home/lucky# cat hello.sh
#!/usr/bin/bash
val=10
echo $val


echo "Enter names:"
read name1 name2 name3
echo "Entered names:"$name1 $name2 $name3


root@vm1:/home/lucky# ./hello.sh
10
Enter names:
l r v
Entered names:l r v
```

How to take user input in the same line??

```
root@vm1:/home/lucky# vi hello.sh
root@vm1:/home/lucky# "hello.sh" 6L, 76C written
root@vm1:/home/lucky# cat hello.sh
#!/usr/bin/bash

read -p  "username:" user_val
echo "username:$user_val"


root@vm1:/home/lucky# ./hello.sh
username:lucky
username:lucky
```

Ya…if user is entering some passwords we don't want show that typing password to the people…what is user entering…

So we use the command -s

S for silent

```
root@vm1:/home/lucky# cat hello.sh
#!/usr/bin/bash

read -p  "username:" user_val
read -sp "password:" pwd
echo "username:$user_val"
echo "password:$pwd"


root@vm1:/home/lucky# ./hello.sh
username:luky
password:username:luky
password:123
```

So the password which is entering by the user is not visible to the outside.

But we encounter an issue with this…username is also printing in same line..to resolve this…just add one echo under the password.

```
root@vm1:/home/lucky# cat hello.sh
#!/usr/bin/bash

read -p  "username:" user_val
read -sp "password:" pwd
echo
echo "username:$user_val"
echo "password:$pwd"


root@vm1:/home/lucky# ./hello.sh
username:lucky
password:
username:lucky
password:123
```

If you want to take a bunch of inputs from the user…we can use array

a is used to indicate array…let's see what happens

```
root@vm1:/home/lucky# cat hello.sh
#!/usr/bin/bash

echo "Enter names:  "
read -a name
echo "names:" ${name[0]} ${name[1]} ${name[2]}

root@vm1:/home/lucky# ./hello.sh
Enter names:
q w e
names: q w e
```

If you don't take any variable for taking input ……..

An automatic system variable "REPLY" will be assigned ….

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo "names:"
read
echo "names:"$REPLY





root@vm1:/home/lucky# ./hello.sh
names:
qwerty
names:qwerty
```

How to pass arguments to the bash script?

```
root@vm1:/home/lucky# vi hello.sh
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo $1 $2 $3





root@vm1:/home/lucky# ./hello.sh

root@vm1:/home/lucky# ./hello.sh q w e
q w e
```


```
root@vm1:/home/lucky# vi hello.sh
root@vm1:/home/lucky# ./hello.sh q w e
./hello.sh q w e
```

Another way of parsing into an array…

How to parse arguments into an array

$@ stores arguments into an array

```
test@test:~/Desktop$ ./hello.sh Mark Tom John
Mark Tom John  > echo $1 $2 $3
test@test:~/Desktop$ ./hello.sh Mark Tom John
./hello.sh Mark Tom John  > echo $1 $2 $3
test@test:~/Desktop$
```

```
hello.sh
#! /bin/bash

echo $0 $1 $2 $3  ' > echo $1 $2 $3'

args=("$@")

echo ${args[0]} ${args[1]} ${args[2]} ${args[3]}
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

v="$@"
echo ${v[0]} ${v[1]} ${v[2]}





root@vm1:/home/lucky# ./hello.sh 1 2 3
1 2 3
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

v="$@"
echo $@





root@vm1:/home/lucky# ./hello.sh m n o
m n o
```

To print the no.of arguments passed to the shell…. $#

```
root@vm1:/home/lucky# vi hello.sh
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

v="$@"
echo $#





root@vm1:/home/lucky# ./hello.sh m n q
3
```

Conditional statements

```
integer comparison

-eq - is equal to - if [ "$a" -eq "$b" ]
-ne - is not equal to - if [ "$a" -ne "$b" ]
-gt - is greater than - if [ "$a" -gt "$b" ]
-ge - is greater than or equal to - if [ "$a" -ge "$b" ]
-lt - is less than - if [ "$a" -lt "$b" ]
-le - is less than or equal to - if [ "$a" -le "$b" ]
< - is less than  - (("$a" < "$b"))
<= - is less than or equal to - (("$a" <= "$b"))
> - is greater than - (("$a" > "$b"))
>= - is greater than or equal to  - (("$a" >= "$b"))

string comparison
= - is equal to - if [ "$a" = "$b" ]
== - is equal to - if [ "$a" == "$b" ]
!= - is not equal to - if [ "$a" != "$b" ]
< - is less than, in ASCII alphabetical order - if [[ "$a" < "$b" ]
> - is greater than, in ASCII alphabetical order - if [[ "$a" > "$b
-z - string is null, that is, has zero length
```

->Symbols are like(>,<,<=,>=) ….round brackets(()) are used …two sets of round brackets are used.

->For letters (eq,ne,ge,gt,lt,le)…square brackets[] are used….

-> for string comparison's , [] square bracket is used..

*Basic syntax of if statement

```
if [ condition ]
then
  statement
fi
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

c=10

if  [  $c -eq 10 ]
then
        echo "condition is true"
fi





root@vm1:/home/lucky# ./hello.sh
condition is true
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

c=10

if  [  $c -ne 100  ]
then
        echo "condition is true"
fi





root@vm1:/home/lucky# ./hello.sh
condition is true
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

c=10

if  [  $c -gt 9  ]
then
        echo "condition is true"
fi





root@vm1:/home/lucky# ./hello.sh
condition is true
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

c=10

if  [  $c -lt 9  ]
then
        echo "condition is true"
fi





root@vm1:/home/lucky# ./hello.sh
root@vm1:/home/lucky#
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

c=10

if  [  $c -ge 9  ]
then
        echo "condition is true"
fi





root@vm1:/home/lucky# ./hello.sh
condition is true
```

```
root@vm1:/home/lucky# cat hello.sh
#! /usr/bin/bash

c=0

if  [  $c -le 9  ]
then
        echo "condition is true"
fi





root@vm1:/home/lucky# ./hello.sh
condition is true
```

```
#! /bin/bash

word=abc

if  [ $word = "abc" ]
then
  echo "condition is true"
fi
```

```
#! /bin/bash

word=a

if  [[ $word < "b" ]]
then
  echo "condition is true"
else
  echo "condition is false"
fi
```

```
word=a

if  [[ $word == "b" ]]
then
  echo "condition b is true"
elif
then
  echo "condition a is true"
else
  echo "condition is false"
fi
```

*****FILE TEST OPERATORS*****

If you want to place the cursor in the same line put \c ….it will print \c as it there… but along with \c we need to put -e…-e enables the functionality of \c …

To check whether the file exists or not a special flag is there

Command is -e filename

```
root@vm:/home/lucky# cat hello.sh
echo  -e "Enter the file name: \c"
read file_name

if [ -e $file_name ]
then
        echo "file found"
else
        echo "file not found"
fi

root@vm:/home/lucky# ./hello.sh
Enter the file name: hello.sh
file found
root@vm:/home/lucky# ./hello.sh
Enter the file name: kushi
file not found
```

-e for file exists or not

-f is for file exists and file is a regular file or not

-d is whether that directories exists or not

For block special file we use flag -b

For character special file , we use flag -c

Flag which checks the file empty or not (-s)

```
root@vm:/home/lucky# cat hello.sh
echo  -e "Enter the file name: \c"
read file_name

if [ -d $file_name ]
then
        echo "file found"
else
        echo "file not found"
fi

root@vm:/home/lucky# ./hello.sh
Enter the file name: h
file found
```

```
root@vm:/home/lucky# cat hello.sh
echo  -e "Enter the file name: \c"
read file_name

if [ -s $file_name ]
then
        echo "not empty"
else
        echo "empty"
fi

root@vm:/home/lucky# ./hello.sh
Enter the file name: 4.txt
empty
```

```
root@vm:/home/lucky# ./hello.sh
Enter the file name: hello.sh
not empty
```

-w checks whether the file has write permission or not..

How to append output to the end of text file

```
root@vm:/home/lucky# cat hello.sh
echo  -e "Enter the file name: \c"
read file_name

if [ -f $file_name ]
then
        if [ -w ]
        then
                echo "write some text here ...to quit cntrl+d"
                cat>>$file_name
        fi
else
        echo "file do not have write permissions"
fi
```

```
root@vm:/home/lucky# touch d
root@vm:/home/lucky# ./hello.sh
Enter the file name: d
write some text here ...to quit cntrl+d
hi there
root@vm:/home/lucky# cat hello.sh
echo  -e "Enter the file name: \c"
read file_name

if [ -f $file_name ]
then
        if [ -w ]
        then
                echo "write some text here ...to quit cntrl+d"
                cat>>$file_name
        fi
else
        echo "file do not have write permissions"
fi
```

-a is used for and operator

```
age=25

if [ "$age" -gt 18 ] && [ "$age" -lt 30]
then
  echo "valid age"
  else
  echo "age not valid"
fi
```

```
#! /bin/bash

age=50

if [ "$age" -gt 18 -a "$age" -lt 30 ]
then
  echo "valid age"
  else
  echo "age not valid"
fi
```

```
#! /bin/bash

age=60

if [[ "$age" -gt 18 && "$age" -lt 30 ]]
then
  echo "valid age"
  else
  echo "age not valid"
fi
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

age=60

if [ "$age" -gt 18 ] || [ "$age" -lt 30 ]
then
  echo "valid age"
else
        echo "not valid"
fi

root@vm:/home/lucky# ./hello.sh
valid age
```

```
#! /bin/bash

age=25

if [ "$age" -eq 18 -o "$age" -eq 30 ]
then
  echo "valid age"
  else
  echo "age not valid"
fi
```

```
#! /bin/bash

age=25

if [[ "$age" -eq 18 || "$age" -eq 30 ]]
then
  echo "valid age"
  else
  echo "age not valid"
fi
```

```
root@vm:/home/lucky# vi hello.sh
root@vm:/home/lucky# ./hello.sh
1+1
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

a=20
b=5

echo $((a+b))
echo $((a-b))
echo $((a*b))
echo $((a/b))
echo $((a%b))

root@vm:/home/lucky# ./hello.sh
25
15
100
4
0
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

a=20
b=5

echo $(expr $a + $b )
echo $(expr $a - $b )
echo $(expr $a * $b )
echo $(expr $a / $b )
echo $(expr $a % $b )

root@vm:/home/lucky# ./hello.sh
25
15
expr: syntax error: unexpected argument ‘4.txt’

4
0
```

For expr method ..* symbol is not skipped we won’t get the expected output

But to resolve this error /* back slash in front of *

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

a=20
b=5

echo $(expr $a + $b )
echo $(expr $a - $b )
echo $(expr $a \* $b )
echo $(expr $a / $b )
echo $(expr $a % $b )

root@vm:/home/lucky# ./hello.sh
25
15
100
4
0
```

```
bc(1)                                                   General Comm

NAME
       bc - An arbitrary precision calculator language

SYNTAX
       bc [ -hlwsqv ] [long-options] [  file ... ]

DESCRIPTION
       bc is a language that supports arbitrary precision numbers
       larities in the syntax to the C programming language.  A s
       requested,  the  math  library  is  defined  before proces
       listed on the command line in the order listed.  After all
       All  code  is  executed  as  it is read.  (If a file conta
       standard input.)

       This version of bc contains several extensions beyond trad
       mand  line  options can cause these extensions to print a
       accepted by this processor.  Extensions will be identified

   OPTIONS
       -h, --help
              Print the usage and exit.

       -i, --interactive
              Force interactive mode.
```

```
man bc
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo "20.8+4"


root@vm:/home/lucky# ./hello.sh
20.8+4
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo "20.9+45" | bc


root@vm:/home/lucky# ./hello.sh
65.9
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo "20.9+4" | bc
echo "20.9-4" | bc
echo "20.9*4" | bc
echo "20.9/4" | bc
echo "20.9%4" | bc

root@vm:/home/lucky# ./hello.sh
24.9
16.9
83.6
5
.9
```

For division, it is not getting proper output, in terms of decimal points

Scale is used to fix 2 points after decimal

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo "20.9+4" | bc
echo "20.9-4" | bc
echo "20.9*4" | bc
echo "scale=2;20.9/4" | bc
echo "20.9%4" | bc

root@vm:/home/lucky# vi hello.sh
root@vm:/home/lucky# ./helo.sh
bash: ./helo.sh: No such file or directory
root@vm:/home/lucky# ./hello.sh
24.9
16.9
83.6
5.22
.9
```

-l is used to call the math libraries like power,sqrt

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

n1=20.9
n2=4


echo "$n1+$n2" | bc
echo "$n1-$n2" | bc
echo "20.9*4" | bc
echo "scale=2;20.9/4" | bc
echo "20.9%4" | bc

num=6

echo "sqrt($num)" | bc -l

root@vm:/home/lucky# ./hello.sh
24.9
16.9
83.6
5.22
.9
2.44948974278317809819
```

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

n1=20.9
n2=4


echo "$n1+$n2" | bc
echo "$n1-$n2" | bc
echo "20.9*4" | bc
echo "scale=2;20.9/4" | bc
echo "20.9%4" | bc

num=6

echo "sqrt($num)" | bc -l
echo "3^3" | bc -l

root@vm:/home/lucky# ./hello.sh
24.9
16.9
83.6
5.22
.9
2.44948974278317809819
27
```

Case

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

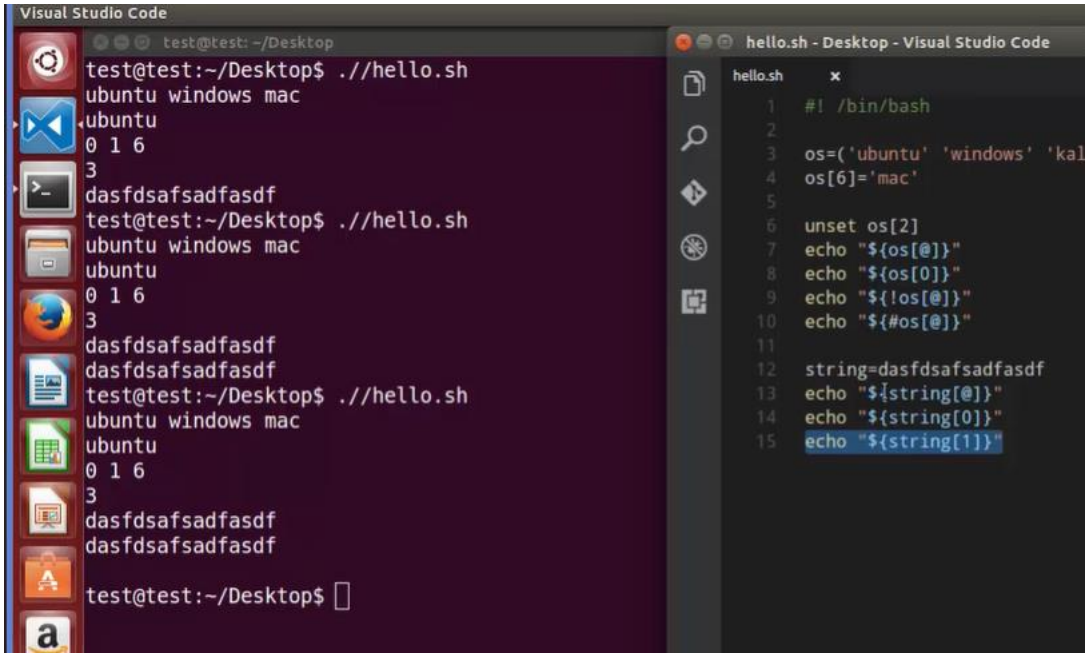
```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

vehicle=$1

case $vehicle in
        "car" )
                echo "Rent of $vehicle is 100 dollar " ;;
        "van" )
                echo "Rent of $vehicle is 80 dollar " ;;
        "bus" )
                echo "Rent of $vehicle is 100 dollar " ;;
        "bicycle" )
                echo "Rent of $vehicle is 5 dollar " ;;
        * )
                echo "vehicle is unknown" ;;
esac

root@vm:/home/lucky# ./hello.sh
vehicle is unknown
root@vm:/home/lucky# ./hello.sh car
Rent of car is 100 dollar
root@vm:/home/lucky# ./hello.sh van
Rent of van is 80 dollar
root@vm:/home/lucky# ./hello.sh bus
Rent of bus is 100 dollar
root@vm:/home/lucky# ./hello.sh bicycle
Rent of bicycle is 5 dollar
root@vm:/home/lucky#
```

The 'LANG' environment variable indicates the language/locale and encoding, where "C" is the language setting

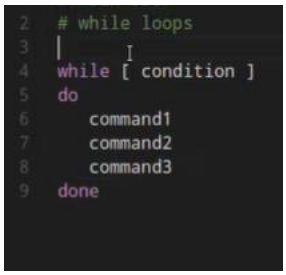
```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

echo -e "enter some character: \c"
read value

case $value in
        [0-9] )
                echo "User entered $value 0 to 9 " ;;
        [a-z] )
                echo "Rent of $value a to z " ;;
        [A-Z] )
                echo "Rent of $value A to Z " ;;
        ? )
                echo "Rent of $value single special character " ;;
        * )
                echo "unknown input" ;;
esac

root@vm:/home/lucky# ./hello.sh
enter some character: g
Rent of g a to z
```

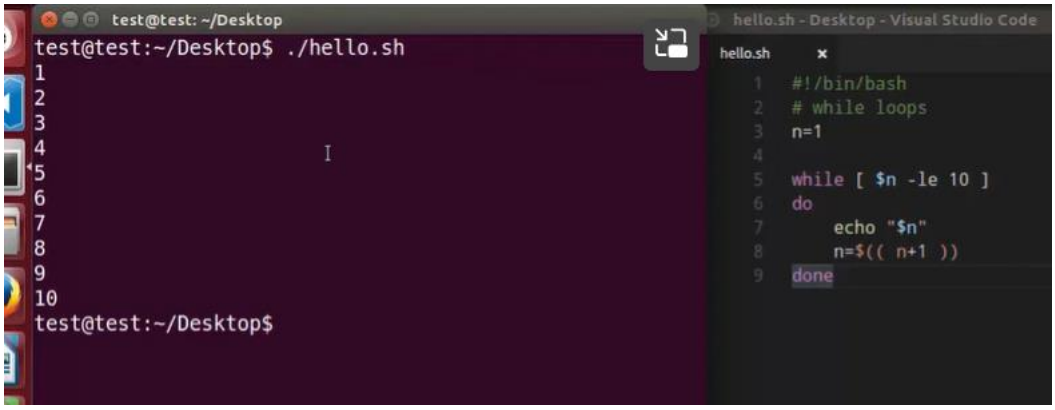
Array

```
root@vm:/home/lucky# cat hello.sh
#! /usr/bin/bash

os=('ubuntu'  'windows'   'kali')

os[3]='mac'
os[0]='linux'
unset os[1]
echo "${os[@]}"
echo "${!os[@]}"
echo "${#os[@]}"
root@vm:/home/lucky# ./hello.sh
linux kali mac
0 2 3
3
```



Length of the string is only one, if we assign string to array





test@test: ~/Desktop

```
5
6
7
8
9
10
test@test:~/Desktop$ ./hello.sh
1
2
3
4
5
6
7
8
9
10
test@test:~/Desktop$ ./hello.sh
1
2
3
4
5
6
```

hello.sh - Desktop - Visual Studio Code

hello.sh

```bash
#!/bin/bash
# while loops
n=1

while (( $n <= 10 ))
do
    echo "$n"
    (( ++n ))
done
```

minal

test@test: ~/Desktop

```
test@test:~/Desktop$ ./hello.sh
1
2
3
4
5
6
7
8
9
10
test@test:~/Desktop$
```

hello.sh - Desktop - Visual Studio Code

hello.sh

```bash
#!/bin/bash
# while loops
n=1

while [ $n -le 10 ]
do
    echo "$n"
    (( n++ ))
    sleep 1
done
```

```
test@test:~/Desktop$ ./hello.sh
1
2
3
test@test:~/Desktop$
```

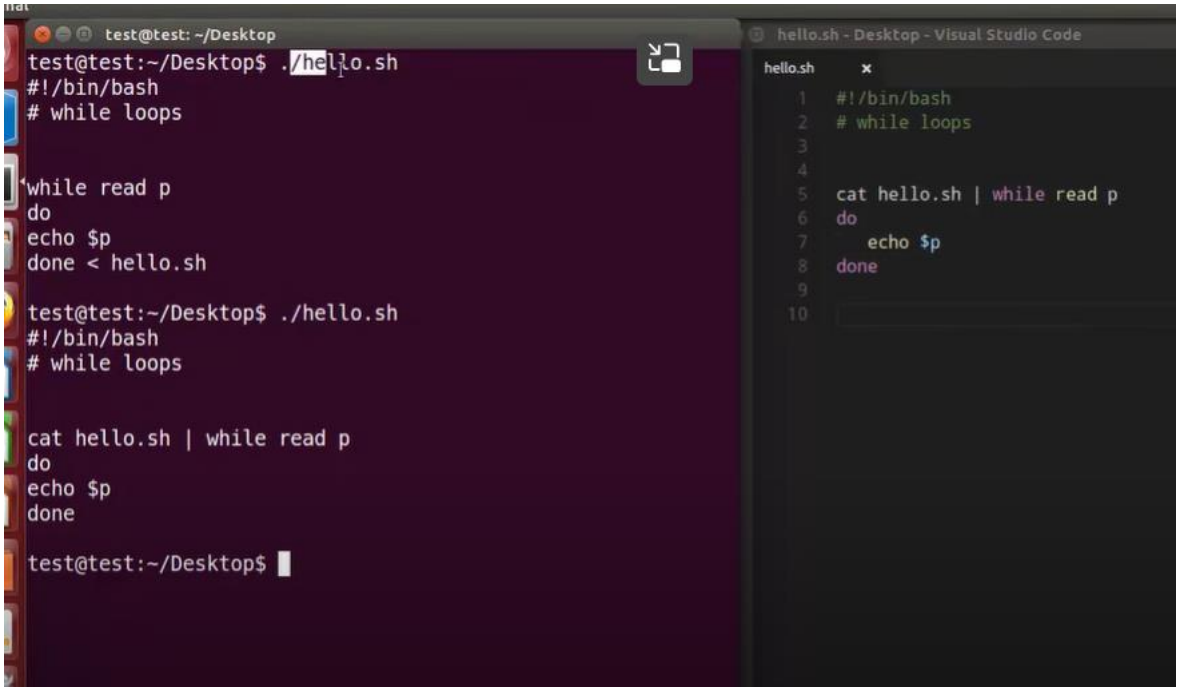
hello.sh

```bash
#!/bin/bash
# while loops
n=1

while [ $n -le 3 ]
do
    echo "$n"
    (( n++ ))
    gnome-terminal &
done
```

```
test@test:~/Desktop$ ./hello.sh
#!/bin/bash
# while loops


while read p
do
echo $p
done < hello.sh

test@test:~/Desktop$
```

hello.sh

```bash
#!/bin/bash
# while loops


while read p
do
    echo $p
done < hello.sh
```



IFS stands for internal feel separator

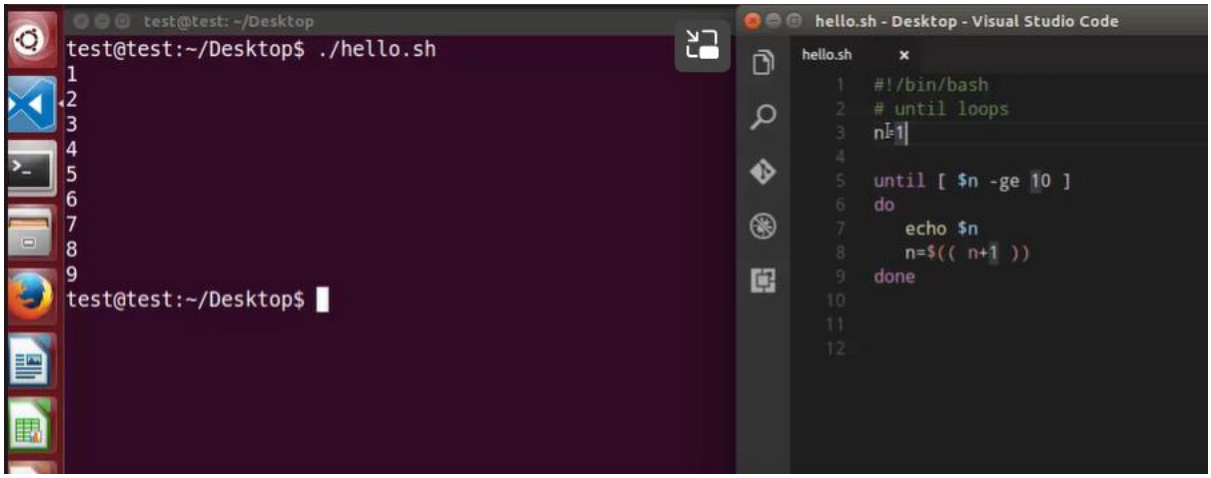
```
test@test: ~/Desktop
# while loops


while read p
do
echo $p
done < hello.sh

test@test:~/Desktop$ ./hello.sh
#!/bin/bash
# while loops


cat hello.sh | while read p
do
echo $p
done

test@test:~/Desktop$ ./hello.sh
#!/bin/bash
# while loops


while IFS= read -r line
do
echo $line
done < hello.sh
```

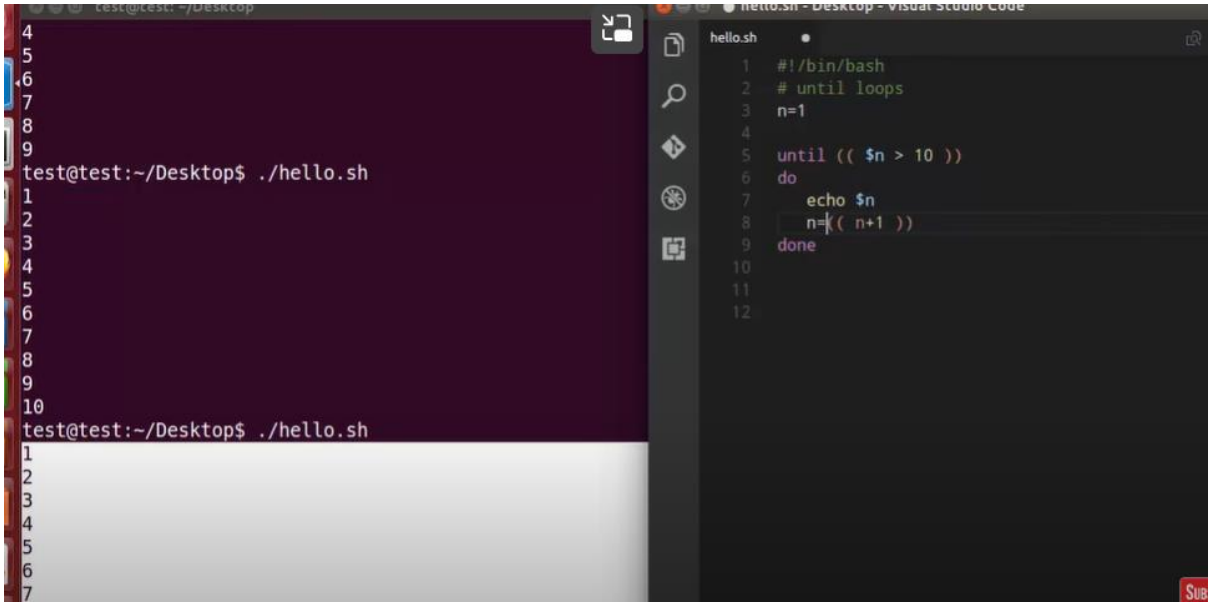
```
hello.sh - Desktop - Visual Studio Code
hello.sh
#!/bin/bash
# while loops


while IFS= read -r line
do
   echo $line
done < hello.sh
```

```
test@test: ~/Desktop
test@test:~/Desktop$ cat /etc/host.conf
# The "order" line is only used by old versions of the C li
brary.
order hosts,bind
multi on
test@test:~/Desktop$ ./hello.sh
# The "order" line is only used by old versions of the C li
brary.
order hosts,bind
multi on
test@test:~/Desktop$
```

```
hello.sh - Desktop - Visual Studio Code
hello.sh
#!/bin/bash
# while loops


while IFS= read -r line
do
   echo $line
done < /etc/host.conf
```

```
until [condition ]
do
   command1
   command2
   ...
   ....
   commandN
done
```





Loops: Sometimes we want to run a command (or group of commands) over and over. This is called iteration, repetition, or looping. The most commonly used shell repetition structure is the for loop, which has the general form:

for variable in list

do

command(s)

done

```
for VARIABLE in 1 2 3 4 5 .. N
do
    command1
    command2
    commandN
done
#OR----------------------------------------

for VARIABLE in file1 file2 file3
do
    command1 on $VARIABLE
    command2
    commandN
done
#OR----------------------------------------

for OUTPUT in $(Linux-Or-Unix-Command-H
do
    command1 on $OUTPUT
    command2 on $OUTPUT
    commandN
done
#OR----------------------------------------
for (( EXP1; EXP2; EXP3 ))
```

```
for VARIABLE in 1 2 3 4 5 .. N
do
    command1
    command2
    commandN
done
#OR----------------------------------------

for VARIABLE in file1 file2 file3
do
    command1 on $VARIABLE
    command2
    commandN
done
#OR----------------------------------------

for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
    command1 on $OUTPUT
    command2 on $OUTPUT
    commandN
done
#OR----------------------------------------
for (( EXP1; EXP2; EXP3 ))
do
```

```
test@test:~/Desktop$ ./hello.sh
1
2
3
4
5
test@test:~/Desktop$
```

```
#!/bin/bash
# for loops

for i in 1 2 3 4 5
do
    echo $i
done
```

```
3
4
5
test@test:~/Desktop$ ./hello.sh
1
2
3
4
5
6
7
8
9
10
test@test:~/Desktop$ ./hello.sh
1
3
5
7
9
test@test:~/Desktop$ ./hello.sh
0
2
```

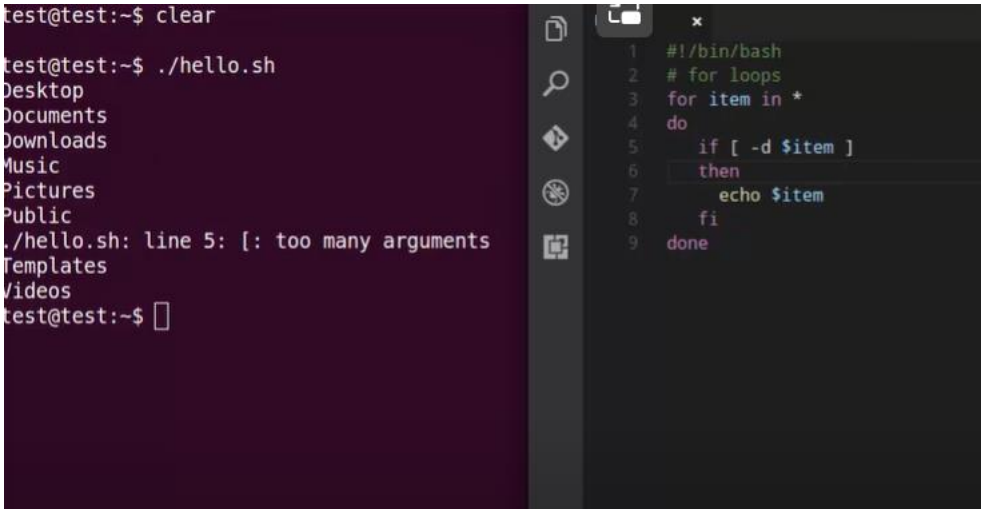
```bash
#!/bin/bash
# for loops

for i in {0..10..2}
do
    echo $i
done
```

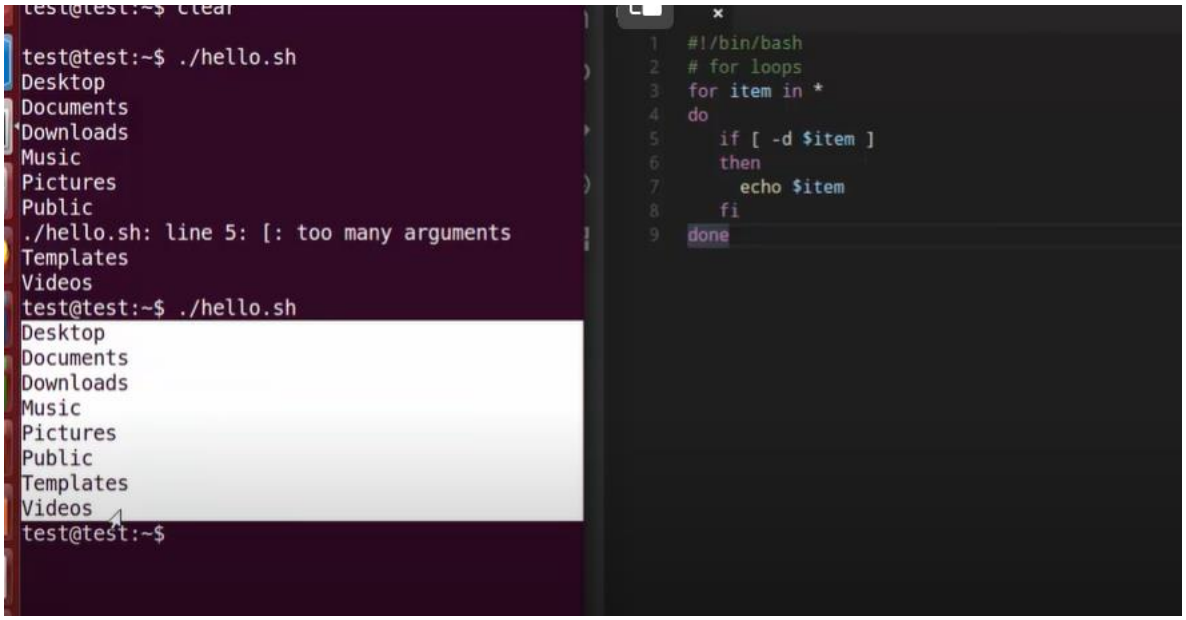
```bash
echo ${BASH_VERSION}
```

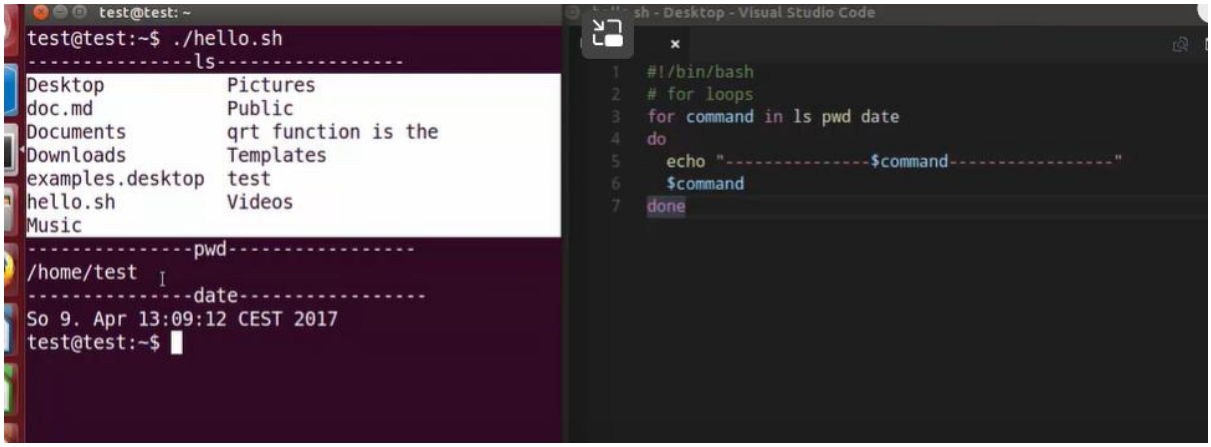
```
4
5
6
7
8
9
10
test@test:~/Desktop$ ./hello.sh
1
3
5
7
9
test@test:~/Desktop$ ./hello.sh
0
2
4
6
8
10
test@test:~/Desktop$ ./hello.sh
4.3.11(1)-release
0
2
4
6
8
10
```

hello.sh

```bash
#!/bin/bash
# for loops
echo ${BASH_VERSION}
for (( i=0; i<5; i++ ))
do
    echo $i
done
```







SELECT COMMAND Constructs simple menu from word list. It Allows user to enter a number instead of a word. So User enters sequence number corresponding to the word. ---------- Syntax:

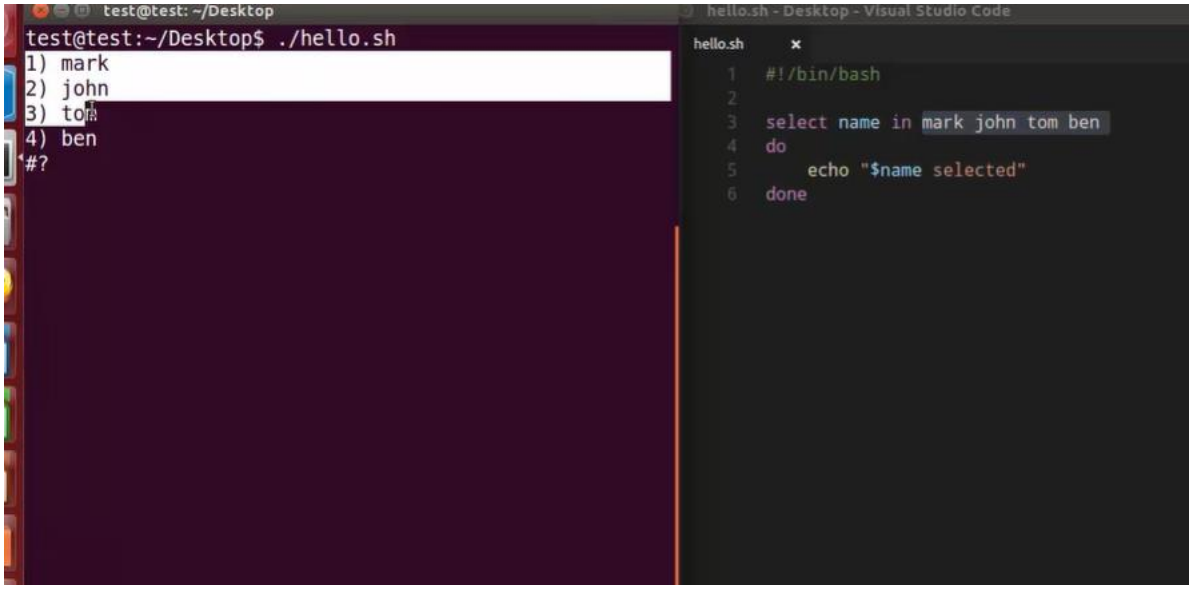
select WORD in LIST

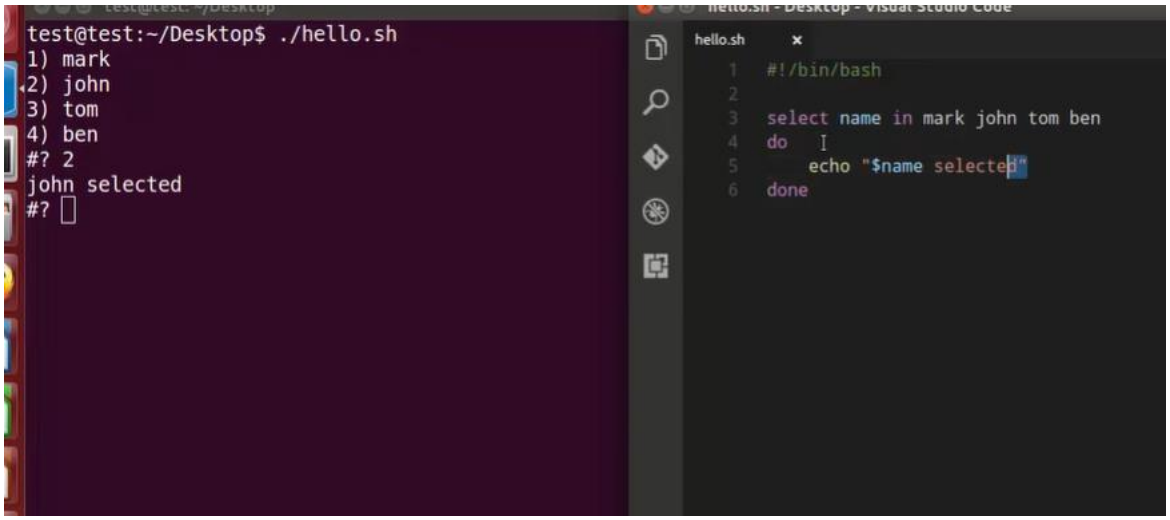
do

RESPECTIVE-COMMANDS

done

Loops until end of input, i.e. ^d (or ^c)





```
test@test:~/Desktop$ ./hello.sh
1) mark
2) john
3) tom
4) ben
#? 2
john selected
#? 3
tom selected
#? 1
mark selected
#? ^C
test@test:~/Desktop$ ./hello.sh
```

```bash
#!/bin/bash

select name in mark john tom ben
do
    case $name in
    mark)
      echo mark selected
      ;;
    john)
      echo john selected
      ;;
    tom)
      echo tom selected
      ;;
    ben)
      echo ben selected
      ;;
    *)
      echo "Error please provide the no. between 1..4"
    esac
done
```

The break statement is used to exit the current loop before its normal ending. The continue statement resumes iteration of an enclosing for, while, until or select loop.

```
7
8
9
10
test@test:~/Desktop$ ./hello.sh
./hello.sh: line 5: [: missing `]'
1
./hello.sh: line 5: [: missing `]'
2
./hello.sh: line 5: [: missing `]'
3
./hello.sh: line 5: [: missing `]'
4
./hello.sh: line 5: [: missing `]'
5
./hello.sh: line 5: [: missing `]'
6
./hello.sh: line 5: [: missing `]'
7
./hello.sh: line 5: [: missing `]'
8
./hello.sh: line 5: [: missing `]'
9
./hello.sh: line 5: [: missing `]'
10
test@test:~/Desktop$ ./hello.sh
```

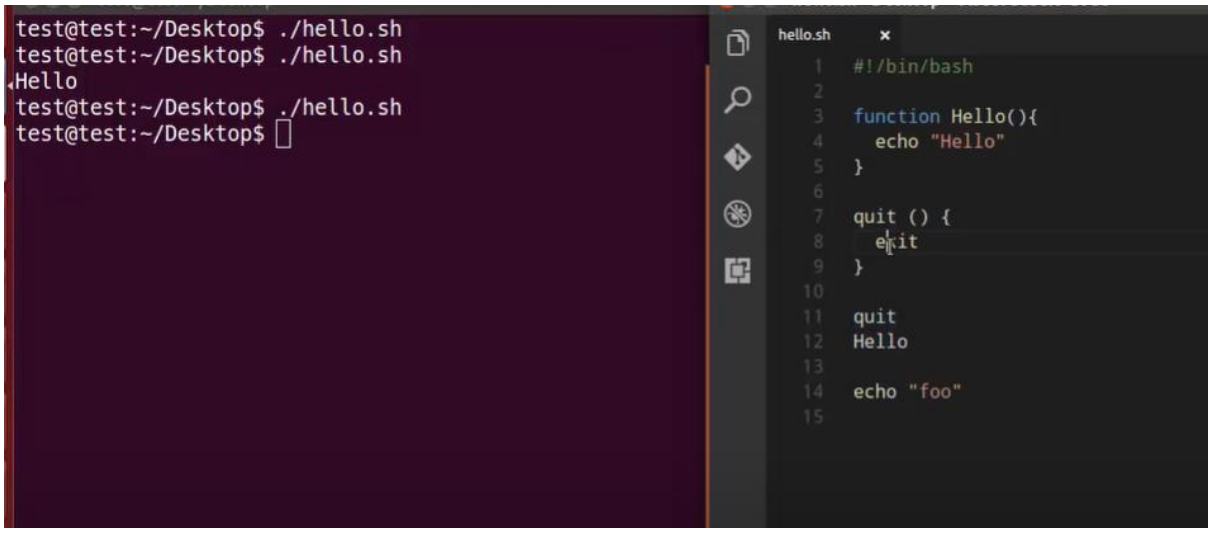
```bash
#!/bin/bash

for (( i=1 ; i<=10 ; i++ ))
do
    if [ $i -gt 5 ]
    then
      break
    fi
    echo "$i"
done
```

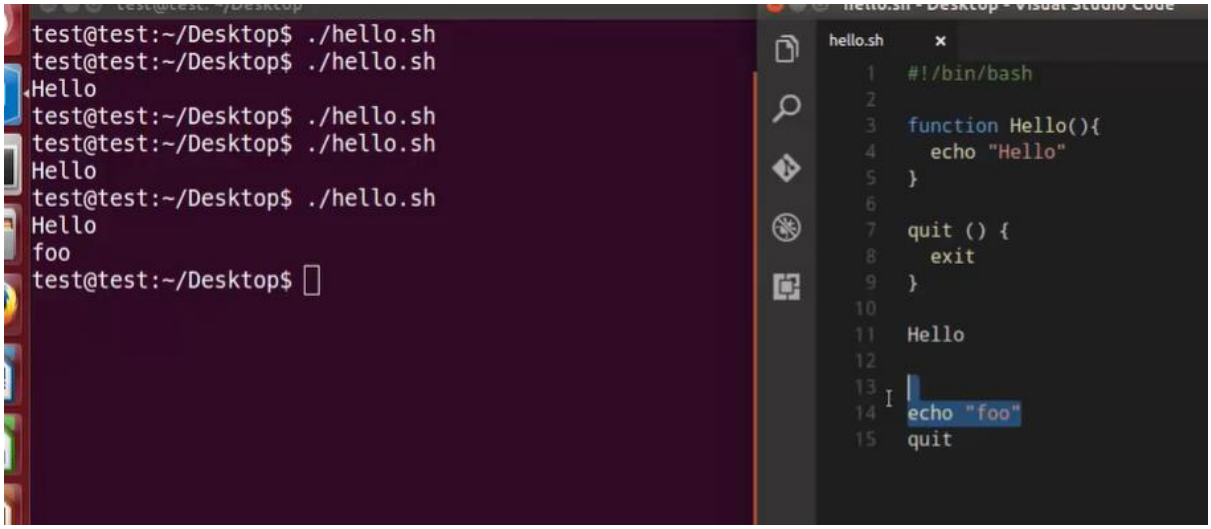
```
test@test:~/Desktop$ ./hello.sh
1
2
4
5
7
8
9
10
test@test:~/Desktop$
```

```bash
#!/bin/bash

for (( i=1 ; i<=10 ; i++ ))
do
    if [ $i -eq 3 -o $i -eq 6 ]
    then
      continue
    fi
    echo "$i"
done
```

Functions : Functions make scripts easier to maintain. Basically it breaks up the program into smaller pieces. A function performs an action defined by you, and it can return a value if you wish.

```
function name(){
  Commands
}

name () {
  Commands
}
```





```
test@test:~/Desktop$ ./hello.sh
test@test:~/Desktop$ ./hello.sh
Hello
test@test:~/Desktop$ ./hello.sh
test@test:~/Desktop$ ./hello.sh
Hello
test@test:~/Desktop$ ./hello.sh
Hello
foo
test@test:~/Desktop$ ./hello.sh
Hello
foo
test@test:~/Desktop$ ./hello.sh
Hello
World
foo
test@test:~/Desktop$ ./hello.sh
Hello
World
Again
foo
test@test:~/Desktop$
```

```bash
#!/bin/bash

function print(){
  echo $1 $2 $3
}

quit () {
  exit
}

print Hello
print World
print Again

echo "foo"
quit
```

```
test@test:~/Desktop$ ./hello.sh
the name is Max
foo
test@test:~/Desktop$
```

```bash
#!/bin/bash

function print(){
  name=$1
  echo "the name is $name"
}

print Max

echo "foo"
```

All variables are global in shell script

```
test@test:~/Desktop$ ./hello.sh
the name is Max
foo
test@test:~/Desktop$ ./hello.sh
The name is Tom
the name is Max
foo
test@test:~/Desktop$
```

```bash
#!/bin/bash

function print(){
  name=$1
  echo "the name is $name"
}

name="Tom"

echo "The name is $name"

print Max

echo "foo"
```

```
test@test:~/Desktop$ ./hello.sh
the name is Max
foo
test@test:~/Desktop$ ./hello.sh
The name is Tom
the name is Max
foo
test@test:~/Desktop$ ./hello.sh
The name is Tom : Before
the name is Max
The name is Max : After
test@test:~/Desktop$ ./hello.sh
The name is Tom : Before
the name is Max
The name is Tom : After
test@test:~/Desktop$
```

hello.sh

```bash
#!/bin/bash

function print(){
  local name=$1
  echo "the name is $name"
}

name="Tom"

echo "The name is $name : Before"

print Max

echo "The name is $name : After"
```

```
test@test:~/Desktop$ ./hello.sh
./hello.sh: line 7: var: readonly variable
test@test:~/Desktop$ ./hello.sh
./hello.sh: line 7: var: readonly variable
var => 31
test@test:~/Desktop$ ./hello.sh
./hello.sh: line 7: var: readonly variable
var => 31
Hello World
test@test:~/Desktop$ ./hello.sh
./hello.sh: line 7: var: readonly variable
var => 31
./hello.sh: line 19: hello: readonly function
test@test:~/Desktop$
```

hello.sh

```bash
#!/bin/bash

var=31

readonly var

var=50

echo "var => $var"

hello() {
  echo "Hello World"
}

readonly -f hello

hello() {
  echo "Hello World Again"
}
```