

# **SHELL SCRIPTING**

A large, stylized version of the 'URX' logo in a light gray color. The letters are bold and blocky, with a slight 3D effect. The 'U' has a small, dark gray, multi-lined detail at its top left corner.

## Shell Scripting

### What is shell?

A unix shell program interprets the user command which are either entered by the user or which can be read from a file called shell script or shell programs.

Shell scripts are interpreted and not compiled. So, u can simply write shell script and run the script u don't need any compiler to compile it.

Different types of shell:

Type “cat /etc./shells” this command will give you the different types of shell supported by your Linux.

sh: bourne shell is the oldest shell to exist. It is used in many older linux systems.

Bash: bourne again shell was reinvented and is improved version of sh. This is Standard GNU shell which is intuitive and flexible.

### Writing first shell script:

**#!/bin/bash:**

**#!** Is shebang and **/bin/bash** is the location of shell. It tells shell interpreter that which shell is being use.

**echo “hello world”.**

To echo hello world.

### Variables and Comments:

Use “#” to write comment.

**Variables:** variables are containers which stores some data inside them.

In shell there are two types of variables, first one is **system variables** which are predefined.

And second one is **user variables** which are declared by users.

U can declare variable in small and upper cases both according to u.

```
#!/bin/bash

echo "Hello World"
echo this is without inverted commas

#SYSTEM VARIABLES
echo $BASH
echo $PWD
echo $BASH_VERSION
echo $HOME

#USER VARIABLES

name=vipul
age=34
gender=male

echo The name is $name.
echo The age is $age.
echo The gender is $gender.

~
~
~
~
~
~
~
~
-- REPLACE --
```

Output:

```
vipul@VipsyVrx: ~
vipul@VipsyVrx:~$ ./first.sh
Hello World
this is without inverted commas
/bin/bash
/home/vipul
5.1.16(1)-release
/home/vipul
The name is vipul.
The age is 34.
The gender is male.
vipul@VipsyVrx:~$
```

## Read User Input:

To take input from user. Use “read”

```
#!/bin/bash

echo "Enter your name:"
read name #here name is the variable in which user given value will be stored.
echo " The name is $name "

echo "Enter the cities: "
read city1 city2 city3 # To enter multiple inputs.
echo "Cities: $city1, $city2, $city3 "

read -p "USERNAME: " username #To enter input on same line
read -sp "PASSWORD: " password #-s flag is used to hide the input to achieve confidentiality.

echo USERNAME: $username
echo PASSWORD: $password

echo "Enter Subjects: "
read -a subject #to store values inside a array.
echo "Subject : ${subject[0]}, ${subject[1]}, ${subject[2]}, ${subject[3]} "

echo "Enter Age "
read #if you dont give any input variable then shell stores the values in REPLY variable.
echo "age = $REPLY "
```

## Output:

```
vipul@VipsyVrx:~$ ./input.sh
Enter your name:
Vipul
The name is Vipul
Enter the cities:
thane virar juhu
Cities: thane, virar, juhu
USERNAME: Vipul
PASSWORD: USERNAME: Vipul
PASSWORD: Jiraya
Enter Subjects:
hist maths geo sci
Subject : hist, maths, geo, sci
Enter Age
34
age = 34
vipul@VipsyVrx:~$
```

**-p flag:** to take user input on same line

**-a flag:** to store values in array.

**-sp flag:** to achieve confidentiality.

## Pass arguments to a Bash-Script:

In the script, these arguments can be accessed using special positional parameters:

**\$0** - The name of the script.

**\$1 to \$9** - The first to ninth arguments.

**\${10}, \${11}, etc.** - The tenth and subsequent arguments (use curly braces to differentiate).

You can use the special parameter **\$#** to get the number of arguments passed

### Using "\$@" and "\$\*"

**"\$@"** - Expands to all positional parameters as separate words.

**"\$\*"** - Expands to all positional parameters as a single word.

```
#!/bin/bash

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

#to pass arguments into an array.
args=("$@")

echo ${args[0]} ${args[1]} ${args[2]}
echo "$@"
echo "$#"

```

Output:

```
vipul@VipsyVrx:~$ ./passarg.sh vipul neel arnav
./passarg.sh vipul neel arnav > echo $0 $1 $2 $3
{vipul neel arnav}
vipul neel arnav
3
vipul@VipsyVrx:~$

```

## 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
```

## Numeric Comparisons

- `eq`: Equal to
- `ne`: Not equal to
- `lt`: Less than
- `le`: Less than or equal to
- `gt`: Greater than
- `ge`: Greater than or equal to

## String Comparisons

- `=`: Equal to
- `!=`: Not equal to
- `<`: Less than (in ASCII alphabetical order)
- `>`: Greater than (in ASCII alphabetical order)
- `z`: String is null (zero length)
- `n`: String is not null (non-zero length)

Conditional statements in Bash scripting allow you to execute code blocks based on certain conditions. These conditions can involve comparisons of numbers, strings, file attributes, and more. Below are the main types of conditional statements used in Bash scripting

### **IF statements:**

The `if` statement is the most basic form of conditional statement. It allows you to execute a block of code if a specified condition is true.

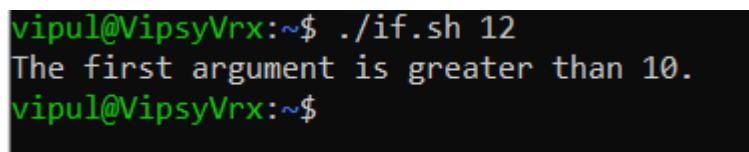
#### **Syntax:**

```
if [ condition];  
then  
    # code to execute if condition is true  
fi
```

#### **Example:**

```
#!/bin/bash  
  
if [ "$1" -gt 10 ];  
then  
    echo "The first argument is greater than 10."  
fi
```

#### **Output:**



```
vipul@VipsyVrx:~$ ./if.sh 12  
The first argument is greater than 10.  
vipul@VipsyVrx:~$
```



## IF ELSE Statements:

The `if-else` statement allows you to execute one block of code if a condition is true and another block if it is false.

### Syntax:

```
if [ condition ]; then
    # code to execute if condition is true
else
    # code to execute if condition is false
fi
```

### Example:

```
#!/bin/bash
if [ "$1" -gt 10 ]; then
    echo "The first argument is greater than 10."
else
    echo "The first argument is not greater than 10."
fi
```

### Output:

```
vipul@VipsyVrx:~$ ./ifelse.sh 5
The first argument is not greater than 10.
vipul@VipsyVrx:~$ ./ifelse.sh 19
The first argument is greater than 10.
vipul@VipsyVrx:~$
```

## If-Elif-Else Statements:

The `if-elif-else` statement allows you to test multiple conditions in sequence.

Syntax:

```
if [ condition1 ]; then
    # code to execute if condition1 is true
elif [ condition2 ]; then
    # code to execute if condition2 is true
else
    # code to execute if all conditions are false
Fi
```

### Example:

```
#!/bin/bash

if [ "$1" -gt 10 ]; then
    echo "The first argument is greater than 10."
elif [ "$1" -eq 10 ]; then
    echo "The first argument is equal to 10."
else
    echo "The first argument is less than 10."
Fi
```

### Output:

```
vipul@VipsyVrx:~$ ./ieelif.sh 16
The first argument is greater than 10.
vipul@VipsyVrx:~$ ./ieelif.sh 10
The first argument is equal to 10.
vipul@VipsyVrx:~$ ./ieelif.sh 5
The first argument is less than 10.
vipul@VipsyVrx:~$
```

## File Test Operator:

- `-e`: File exists
- `-f`: File is a regular file
- `-d`: Directory exists
- `-r`: File is readable
- `-w`: File is writable
- `-x`: File is executable
- `-s`: File has a non-zero size

## Example:

```
#!/bin/bash

echo -e "Enter the name of file: \c "
read filename

if [ -e "$filename" ];
then
    echo "$filename exists"
else
    echo "$filename doesnt exists"
fi
~
```

## Output:

```
vipul@VipsyVrx:~$ ./exist.sh
Enter the name of file: input.sh
input.sh exists
vipul@VipsyVrx:~$ ./exist.sh
Enter the name of file: itachi
itachi doesnt exists
vipul@VipsyVrx:~$
```

## Logical AND (&&):

The Logical AND "&&" is a Boolean operator that executes following commands based on the outcome of previously executed commands. Logical AND (&&) if `[[expr1 && expr2]]` Returns `expr1` if it can be converted to false; otherwise, returns `expr2`.

Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.

-a flag stands for &operator:

If `["$age" -eq 18 -a "$age" -lt 30]`

```
GNU nano 6.2 or.sh
#!/bin/bash

echo -e "Enter the age of person: \c"
read age

if [ "$age" -eq 18 ] && [ "$age" -lt 30 ];
then
    echo "Your age is valid "
else
    echo "your age is not valid"
fi
```

Output:

```
vipul@VipsyVrx:~$ ./or.sh
Enter the age of person: 32
your age is not valid
vipul@VipsyVrx:~$ ./or.sh
Enter the age of person: 23
your age is not valid
vipul@VipsyVrx:~$ ./or.sh
Enter the age of person: 18
Your age is valid
vipul@VipsyVrx:~$
```

## Logical OR:

The Logical OR "||" is a Boolean operator that executes following commands based on the outcome of previously executed commands.

Logical OR (||) if [[expr1 ||expr2]] Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if one operands are true; otherwise, returns false.

**for using OR operator use ||**

```
if [ "$age" -gt 18 ] || [ "$age" -lt 30 ]
```

**The -o option provide an alternative compound condition test.**

```
if [ "$age" -gt 18 -o "$age" -lt 30 ]
```

**if [[ \$condition1 || \$condition2 ]] # Also works.**

```
if [[ "$age" -gt 18 || "$age" -lt 30 ]]
```

```
GNU nano 6.2
#!/bin/bash

echo -e "Enter the age of person: \c"
read age

if [ "$age" -eq 18 ] || [ "$age" -lt 30 ];
then
    echo "Your age is valid "
else
    echo "your age is not valid"
fi
```

## Output:

```
vipul@VipsyVrx:~$ ./orrr.sh
Enter the age of person: 18
Your age is valid
vipul@VipsyVrx:~$ ./orrr.sh
Enter the age of person: 23
Your age is valid
vipul@VipsyVrx:~$ ./orrr.sh
Enter the age of person: 43
your age is not valid
vipul@VipsyVrx:~$
```

## Arithmetic Operation:

The Arithmetic expression is very important feature for performing number arithmetic operations in scripts.

By default variables are treated as strings in bash scripts.

But parsing a string to numbers is very easy using double parentheses and external command such as `expr`.

### Arithmetic operations in Bash Shell

```
GNU nano 6.2 arith.sh
#!/bin/bash
number_1=10
number_2=5
echo "Addition = $(( number_1 + number_2 ))"
echo "Subtraction = $(( number_1 - number_2 ))"
echo "Multiplication = $(( number_1 * number_2 ))"
echo "Division = $(( number_1 / number_2 ))"
echo "pre-increment = $(( ++number_1 ))"
echo "pre-decrement = $(( --number_2 ))"
```

### Output:

```
vipul@VipsyVrx:~$ ./arith.sh
Addition = 15
Subtraction = 5
Multiplication = 50
Division = 2
pre-increment = 11
pre-decrement = 4
vipul@VipsyVrx:~$
```

### `expr` — evaluate expression

we can also use `expr` command to perform Arithmetic operations on numbers.

```
GNU nano 6.2 expr.sh
#!/bin/bash
number_1=10
number_2=5
echo "Addition = $(expr $number_1 + $number_2 )"
echo "Subtraction = $(expr $number_1 - $number_2 )"
echo "Multiplication = $(expr $number_1 \* $number_2 )"
echo "Division = $(expr $number_1 / $number_2 )"
```

**Output:**

```
vipul@VipsyVrx:~$ ./expr.sh
Addition = 15
Subtraction = 5
Multiplication = 50
Division = 2
vipul@VipsyVrx:~$
```

**Floating point math operation in bash:**

bc, for basic calculator (often referred to as bench calculator), is "an arbitrary-precision calculator language" with syntax similar to the C programming language. bc is typically used as either a mathematical scripting language or as an interactive mathematical shell.

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

bc is a language that supports arbitrary precision numbers with interactive execution of statements. There are some similarities in the syntax to the C programming language. A standard math library is available by command line option. If requested, the math library is defined before processing any files. bc starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, bc reads from the standard input. All code is executed as it is read. (If a file contains a command to halt the processor, bc will never read from the standard input.)

**Example:**

```
vipul@VipsyVrx:~/ShellScripting$ cat bc.sh
#!/bin/bash

num1=12
num2=4

#bc: bash calculator

echo "25.5 + 5" | bc

#scale : to print the decimal points

echo "scale=2;25.5/5" | bc

echo "$num1 + $num2" | bc
echo "$num2 - $num2" | bc
echo "$num2 * $num2" | bc
echo "scale=4;$num2 / $num2" | bc
```

## Output:

```
vipul@VipsyVrx:~/ShellScripting$ ./bc.sh
30.5
5.10
16
0
16
1.0000
```

## The Case Statement:

The case statement is used to execute statements based on specific values. Often used in place of an if statement, if there are a large number of conditions.

## Syntax:

```
case $var in
    val1 )
        statements;
    val2 )
        statements;;
    *)
        statements;;
esac
```

In the above syntax:

- Value used can be an expression
- each set of statements must be ended by a pair of semicolons;
- a \*) is used to accept any value not matched with list of values

```
vipul@VipsyVrx:~/ShellScripting$ cat case.sh
#!/bin/bash

vehicle=$1

case $vehicle in
    "car")
        echo "User entered Car";;
    "Bike")
        echo "User Entered Bike";;
    "Truck")
        echo "User Entered Truck";;
    *)
        echo "Invalid Vehicle";;
esac

vipul@VipsyVrx:~/ShellScripting$ ./case.sh car
User entered Car
vipul@VipsyVrx:~/ShellScripting$ ./case.sh xyz
Invalid Vehicle
vipul@VipsyVrx:~/ShellScripting$
```



```

vipul@VipsyVrx:~/ShellScripting$ ./case.sh car
User entered Car
vipul@VipsyVrx:~/ShellScripting$ ./case.sh cd
Invalid Vehicle
vipul@VipsyVrx:~/ShellScripting$ █

```

## Case2.sh

```

vipul@VipsyVrx:~/ShellScripting$ ./case2.sh
Enter some character : 3
User entered 3 0 to 9
vipul@VipsyVrx:~/ShellScripting$ ./case2.sh
Enter some character : a
User entered a a to z
vipul@VipsyVrx:~/ShellScripting$ ./case2.sh
Enter some character : G
User entered G A to Z
vipul@VipsyVrx:~/ShellScripting$ ./case2.sh
Enter some character : *
User entered * Special character
vipul@VipsyVrx:~/ShellScripting$ ./case2.sh
Enter some character : abdddddad
Unknown input
vipul@VipsyVrx:~/ShellScripting$

```

## Array variables:

An array is a variable containing multiple values. Any variable may be used as an array. There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously. Arrays are zero-based: the first element is indexed with the number 0. Indirect declaration is done using the following syntax to declare a variable: `ARRAY[INDEXNR]=value`

```

vipul@VipsyVrx:~/ShellScripting$ cat array.sh
#!/bin/bash

gun=('Operator' 'Vandal' 'Phantom' )

echo "${gun[@]}"
echo "${gun[0]}"
echo "${!gun[@]}"
echo "${#gun[@]}"

gun[5]='Odin' #adding in array at index 5

echo "${gun[@]}"
echo "${!gun[@]}"

unset gun[1] #Unset will unset the array index and make it null. Shifting doesnt happens 1 will be null now

echo "${gun[@]}"
echo "${!gun[@]}"

String=HereComestheParty

echo "${String[@]}"
echo "${String[0]}"
echo "${String[1]}"
vipul@VipsyVrx:~/ShellScripting$

```

**Output:**

```
vipul@VipsyVrx:~/ShellScripting$ ./array.sh
Operator Vandal Phantom
Operator
0 1 2
3
Operator Vandal Phantom Odin
0 1 2 5
Operator Phantom Odin
0 2 5
HereComestheParty
HereComestheParty
```

**While Loop:**

Loop is a block of code that is repeated a number of times. The repeating is performed either a pre-determined number of times until a particular condition is satisfied ( while and until loops) To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

**While Loop: SYNTAX**

While [ conditions]

do

    Command1

    Command2

    Command3

    Commandn

done

**Code and Output:**

```
#!/bin/bash

#(( )) is used for operations [ ] is used for eq/le operation
n=1

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

vipul@VipsyVrx:~/ShellScripting$ ./while.sh
1
2
3
4
5
6
7
8
9
10
vipul@VipsyVrx:~/ShellScripting$
```

## using sleep and open terminal with WHILE Loops:

In bash scripting, the sleep command is used to pause the execution of a script for a specified amount of time. This can be useful in various scenarios, such as when you want to wait for a certain process to complete, introduce a delay between commands, or create a simple timer.

### sleep NUMBER[SUFFIX]

NUMBER: The amount of time to sleep.

SUFFIX: The unit of time. The default unit is seconds. You can also use:

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

Examples:

Sleep 2m

Sleep 3s

## Read a file content in Bash:

How do I read a text file line by line under a Linux or UNIX-like system using BASH shell

? You can use while..do..done bash loop to read file line by line on a Linux, OSX, or Unix-like system.

There are basically three ways to read file content using bash.

1a: While loop: Single line at a time: Input redirection.

1b: While loop: Single line at a time:

Open the file, read from a file descriptor (in this case file descriptor #4).

2: While loop: Read file into single variable and parse.

```
vipul@VipsyVrx:~/ShellScripting$ cat while1a.sh
#!/bin/bash

while read p
do
    echo $p
done < while1a.sh

vipul@VipsyVrx:~/ShellScripting$ ./while1a.sh
#!/bin/bash

while read p
do
echo $p
done < while1a.sh
```

```
vipul@VipsyVrx:~/ShellScripting$ cat while1b.sh
#!/bin/bash

cat while1b.sh | while read p
do
    echo "$p"
done

vipul@VipsyVrx:~/ShellScripting$ ./while1b.sh
#!/bin/bash

cat while1b.sh | while read p
do
echo "$p"
done
```

## Until Loop:

The until structure is very similar to the while structure. The until structure loops until the condition is true. So basically, it is “until this condition is true, do this”.

```
SYNTAX:

until [ conditions ]
do
    Command1
    Command2
    Command3
done

vipul@VipsyVrx:~/ShellScripting$
```

## Output:

```
vipul@VipsyVrx:~/ShellScripting$ cat until.sh
#!/bin/bash

n=1

until [ $n -ge 10 ]
do
    echo $n
    n=$(( n+1 ))
done

vipul@VipsyVrx:~/ShellScripting$ ./until.sh
1
2
3
4
5
6
7
8
9
```

## FOR Loop:

for Loops: Sometimes we want to run a command (or group of commands) over and over. This is called iteration, repetition, or looping.

```
SYNTAX For FOR LOOPSYNTAX For FOR LOOP

for VARIABLE in 1 2 3 4 5 .. N
do
    command1
    command2
    commandN
done

-----

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

-----

for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
    command1 on $OUTPUT
    command2 on $OUTPUT
    commandN
done

-----

for (( EXP1; EXP2; EXP3 ))
do
    command1
    command2
    command3
done
```

```
vipul@VipsyVrx:~/ShellScripting$ cat for.sh
#!/bin/bash
```

```
for i in {1..10}
do
    echo $i
done
echo ""
```

```
echo "Another usage"
echo ""
for w in {1..10..2}
do
    echo $w
done
echo ""
```

```
echo "Another usage"
echo ""
for (( x=0; x<10; x++ ))
do
    echo $x
done
```

```
vipul@VipsyVrx:~/ShellScripting$ ./for.sh
```

```
1
2
3
4
5
6
7
8
9
10
```

```
Another usage
```

```
1
3
5
7
9
```

```
Another usage
```

```
0
1
2
3
4
5
6
7
8
9
```

## FOR loop to execute commands:

Using ‘\*’ with for loop means to iterate through every file and directory

```
vipul@VipsyVrx:~/ShellScripting$ cat forco.sh
#!/bin/bash

for item in *
do
    if [ -d item ]
    then
        echo $item
    else
        echo "No Directory"
    fi
done

vipul@VipsyVrx:~/ShellScripting$ ./forco.sh
No Directory
No Directory
No Directory
```

## Select Loop:

**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.

```
vipul@VipsyVrx:~/ShellScripting$ cat select.sh
#!/bin/bash

select name in ram raj rohan
do
    echo $name
done

vipul@VipsyVrx:~/ShellScripting$ ./select.sh
1) ram
2) raj
3) rohan
#? 1
ram
#? 2
raj
#? 3
rohan
#? 4

#? ^Z
[2]+  Stopped                  ./select.sh
vipul@VipsyVrx:~/ShellScripting$
```



## Break:

```
vipul@VipsyVrx:~/ShellScripting$ cat break.sh
#!/bin/bash

for (( i=0; i<10; i++ ))
do
    if [ $i -gt 5 ]
    then
        break
    fi
    echo $i
done

vipul@VipsyVrx:~/ShellScripting$ ./break.sh
0
1
2
3
4
5
```

## Continue:

```
vipul@VipsyVrx:~/ShellScripting$ cat continue.sh
#!/bin/bash

for (( i=0; i<10; i++ ))
do
    if [ $i -eq 3 -o $i -eq 6 ]
    then
        continue
    fi
    echo $i
done

vipul@VipsyVrx:~/ShellScripting$ ./continue.sh
0
1
2
4
5
7
8
9
```

## Function:

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.

```
vipul@VipsyVrx:~/ShellScripting$ cat function.sh
#!/bin/bash

function add(){
    echo addition;
}
add

vipul@VipsyVrx:~/ShellScripting$ ./function.sh
addition
```

## LOCAL VARIABLES IN FUNCTIONS:

Variables defined within functions are global, i.e. their values are known throughout the entire shell program keyword “local” inside a function definition makes referenced variables “local” to that function. Local variables can be created by using the keyword local

```
vipul@VipsyVrx:~/ShellScripting$ cat localvari.sh
#!/bin/bash

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

name=Tom

echo "The Name is $name: Before"

print Max

echo "The Name is $name: after"

vipul@VipsyVrx:~/ShellScripting$ ./localvari.sh
The Name is Tom: Before
the name is Max
The Name is Max: after
```

Here name variable value gets updated from Tom to Max after function gets executed.

```

vipul@VipsyVrx:~/ShellScripting$ cat localvari.sh
#!/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"

vipul@VipsyVrx:~/ShellScripting$ ./localvari.sh
The Name is Tom: Before
the name is Max
The Name is Tom: after

```

This is after setting function variable to local.

## READONLY:

readonly command can be used to make you variables and functions read only that means you cannot change the value of variables or you cannot overwrite a function.

## Signals and traps:

\$\$ is used to print pid of script itself.

Signals are notifications sent to a process to indicate that some event has occurred. For example:

SIGINT: Interrupt signal (CTRL + C).

SIGTERM: Termination signal.

SIGHUP: Hangup signal, usually when a terminal closes.

SIGKILL: Kill signal (cannot be caught or ignored).

## Traps:

A trap is a mechanism that allows you to capture a signal and execute a command or script in response. The trap command in Bash is used to specify these actions.

```
vipul@VipsyVrx:~/ShellScripting$ cat trap.sh

#!/bin/bash

# Define a trap to execute when the script receives a SIGINT (Ctrl+C)
trap 'echo "Caught SIGINT signal, exiting..."; exit' SIGINT

# Define a trap to execute on script exit
trap 'echo "Script exited!"' EXIT

echo "This script will run indefinitely until interrupted."
while true; do
    sleep 1
done

vipul@VipsyVrx:~/ShellScripting$ ./trap.sh
This script will run indefinitely until interrupted.
^CCaught SIGINT signal, exiting...
Script exited!
```

## How to debug a bash script:

Use **bash -x scriptname**.

To get more detail view of script.

## **Purpose of shell scripting in DevOps:**

To reduce manual effort, time and complexity.

**Writing a shell script to check node health of vm whenever someone comes and ask us to check it.**

```
#!/bin/bash
```

```
#Author: Vipul
```

```
#Date: 01/08/2024
```

```
#This Script outputs the node health
```

```
#Version: v1
```

```
Set -x # debug mode
```

```
Df -h
```

```
Free -g
```

```
Nproc
```

```
Ps -ef | grep amazon | awk “ “ ‘{print $2}’
```