**Experiment 18**

**Shell: The Shell’s interpretive cycle, Shell variables, Environmental variables, Types of Shell, Changing one shell to other, Shell builtins. Basic shell commands: alias, awk, bg, chsh, echo, ssh, scp**.

Scripts are collections of commands, stored in a file. The shell can read this file and act on the commands as if they were typed at the keyboard. The shell also provides a variety of useful programming features to make scripts truly powerful.

**Basics of Shell Programming**

1. To get a Linux shell, you need to start a terminal.
2. To see what shell you have, run: echo $SHELL.
3. In Linux, the dollar sign ($) stands for a shell variable.
4. The ‘echo‘ command just returns whatever you type in.
5. The pipeline instruction (|) comes to rescue, when chaining several commands.
6. Linux commands have their own syntax, Linux won’t forgive you whatsoever is the mistakes. If you get a command wrong, you won’t flunk or damage anything, but it won’t work.
7. #!/bin/sh – It is called shebang. It is written at the top of a shell script and it passes the instruction to the program /bin/sh.

**About shell Script**

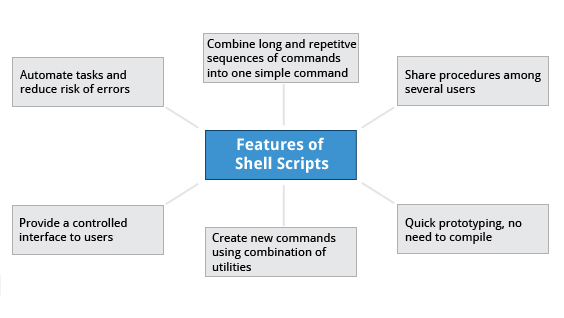
Shell script is just a simple text file with “.sh” extension, having executable permission.

Process of writing and executing a script

1. Open terminal.
2. Navigate to the place where you want to create script using ‘cd‘ command.
3. Cd (enter) [This will bring the prompt at Your home Directory].
4. touch hello.sh (Here we named the script as hello, remember the ‘.sh‘ extension is compulsory).
5. vi hello.sh (nano hello.sh) [You can use your favourite editor, to edit the script].
6. chmod 744 hello.sh (making the script executable).
7. sh hello.sh or ./hello.sh (running the script)

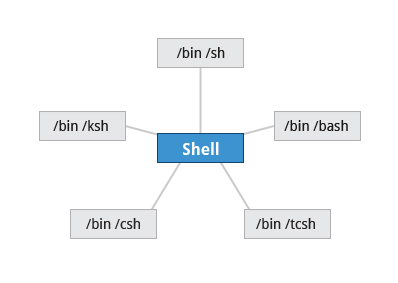
**Features and Capabilities**

A **shell** is a command line **interpreter** which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in. Shell scripting provides following features and capabilities.

**Command Shell Choices**

The command **interpreter** is tasked with executing statements that follow it in the script. Commonly used interpreters include: /usr/bin/perl , /bin/bash, /bin/csh , /usr/bin/python and /bin/sh.

Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command-line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

/bin/sh  
/bin/bash  
/bin/tcsh  
/bin/csh  
/bin/ksh

Most Linux users use the default **bash** shell, but those with long UNIX backgrounds with other shells may want to override the default.

**b) Syntax**

Scripts require you to follow standard language **syntax.** Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within **bash** scripts:

|  |  |
| --- | --- |
| **Character** | **Description** |
| # | Used to add a comment, **except** when used as \#, or as #! when starting a script |
| \ | Used at the end of a line to indicate continuation on to the next line |
| ; | Used to interpret what follows as a new command |
| $ | Indicates what follows is a variable |

Note that when # is inserted at the beginning of a line of commentary, the whole line is ignored.

# This line will not get executed.

**Writing your First Script**

**#!/bin/bash**

**# My first script**

**echo "Hello World!"**

**Writing your Second** **Script**

#! /bin/bash

echo "Hello $USER"

echo "Hey i am" $USER "and will be telling you about the current processes"

echo "Running processes List"

ps

**Taking user’s Input using Read command**

Up until now, we were using variables that has values assigned into the script itself. But what if we want to use different values for our variables every-time we are running our script. For ex, when we are doing arithemetic operation or when we have created a script to take backup of files & want to define which files needs to be backed up.

We can take user’s input as values for our variables by using read command.

Syntax for using read comamnd is

**read variable\_name**

Example

#!/bin/bash

# assigning user’s input as variable’s value

echo “ what is your name ?”

read name

echo “Welcome, $name”Script

Also, I would like to discuss couple of options that can be used with read command

* ***read –s variable\_name***will not show value entered by user on screen. Can be used for passing  sensitive information like a password.

S=silent

**Writing your Third Script**

Moving to, write our third and last script for this article. This script acts as an interactive script. Why don’t you, yourself execute this simple yet interactive script and tell us how you felt.

**#! /bin/bash**

**echo "Hey what's Your First Name?";**

**read a;**

**echo "welcome Mr./Mrs. $a, would you like to tell us, Your Last Name";**

**read b;**

**echo "Thanks Mr./Mrs. $a $b for telling us your name";**

**echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**

**echo "Mr./Mrs. $b, it's time to say you good bye"**

**Constructs using expr**

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

* Using the **expr** utility: **expr**is a standard but somewhat deprecated program. The syntax is as follows:

expr 8 + 8

echo $(expr 8 + 8)

* Using the $((...)) syntax: This is the built-in shell format. The syntax is as follows:  
  echo $((x+1))
* Using the built-in shell command **let**. The syntax is as follows:
* let x=( 1 + 2 ); echo $x

In modern shell scripts the use of **expr** is better replaced with var=$((...))

**What are Shell Variables?**

Variables store data in the form of characters and numbers. Similarly, Shell variables are used to store information and they can by the shell only. You can use variables as in any programming languages. There are no data types. A variable in bash can contain a number, a character, a string of characters.

For example, the following creates a shell variable and then prints it:

variable ="Hello"

echo $variable

**Variable Use**

**$# Stores the number of command-line arguments that were passed to the shell program.**

**$? Stores the exit value of the last command that was executed.**

**$0 Stores the first word of the entered command (the name of the shell program).**

**$\* Stores all the arguments that were entered on the command line ($1 $2 ...).**

**"$@" Stores all the arguments that were entered on the command line, individually quoted ("$1" "$2" ...).**

#!/bin/bash

greeting="Welcome"

user=$(whoami)

day=$(date +%A)

echo "$greeting back $user! Today is $day, which is the best day of the entire week!"

echo "Your Bash shell version is: $BASH\_VERSION. Enjoy!"

By now you should possess all required skills needed to create a new script, making it executable and running it on the command line. After running the above welcome.sh script, you will see an output similar to the one below:

|  |  |  |
| --- | --- | --- |
| -eq | Is Equal To | if [ $1 -eq 200 ] |
| -ne | Is Not Equal To | if [ $1 -ne 1 ] |
| -gt | Is Greater Than | if [ $1 -gt 15 ] |
| -ge | Is Greater Than Or Equal To | if [ $1 -ge 10 ] |
| -lt | Is Less Than | if [ $1 -lt 5 ] |
| -le | Is Less Than Or Equal To | if [ $1 -le 0 ] |
| == | Is Equal To | if (( $1 == $2 )) [Note: Used within double parentheses] |
| != | Is Not Equal To | if (( $1 != $2 )) |
| < | Is Less Than | if (( $1 < $2 )) |
| <= | Is Less Than Or Equal To | if (( $1 <= $2 )) |
| > | Is Greater Than | if (( $1 > $2 )) |
| >= | Is Greater Than Or Equal To | if (( $1 >= $2 )) |

**If statement**

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

**Syntax**

if [ expression ]

then

Statement(s) to be executed if expression is true

fi

The *Shell expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false* then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

**Example**

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

fi

if [ $a != $b ]

then

echo "a is not equal to b"

fi

**The above script will generate the following result −**

a is not equal to b

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

**Syntax**

if [ expression ]

then

Statement(s) to be executed if expression is true

else

Statement(s) to be executed if expression is not true

fi

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

**Example**

The above example can also be written using the *if...else* statement as follows −

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

else

echo "a is not equal to b"

fi

**Upon execution, you will receive the following result −**

a is not equal to b

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

**Syntax**

if [ expression 1 ]

then

Statement(s) to be executed if expression 1 is true

elif [ expression 2 ]

then

Statement(s) to be executed if expression 2 is true

elif [ expression 3 ]

then

Statement(s) to be executed if expression 3 is true

else

Statement(s) to be executed if no expression is true

fi

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

**Example**

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

**Upon execution, you will receive the following result −**

a is less than b

**While Loops**

**while [ <some test> ]**

**do  
<commands>  
done**

## **#!/bin/sh**

## **a=0**

## **while [ $a -lt 10 ]**

## **do**

## **echo $a**

## **a=`expr $a + 1`**

## **done**

## **Write script to print given numbers sum of all digit, For eg. If no is 123 it's sum of all digit will be 1+2+3 = 6**

echo "Enter a Number:"

read n

temp=$n

sd=0

sum=0

while [ $n -gt 0 ]

    do

        sd=$(( $n % 10 ))

        n=$(( $n / 10 ))

        sum=$(( $sum + $sd ))

    done

echo "Sum is $sum"

**Write script to print nos as 5,4,3,2,1 using while loop**

1. #!/bin/bash
2. *# Basic while loop*
3. counter=1
4. while [ $counter -le 10 ]
5. do
6. echo $counter
7. ((counter++))
8. done
9. echo All done

## **Nesting Loops**

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting **while** loop. The other loops can be nested based on the programming requirement in a similar way −

## **Nesting while Loops**

It is possible to use a while loop as part of the body of another while loop.

### Syntax

while command1 ; # this is loop1, the outer loop

do

Statement(s) to be executed if command1 is true

while command2 ; # this is loop2, the inner loop

do

Statement(s) to be executed if command2 is true

done

Statement(s) to be executed if command1 is true

done

### Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine −

#!/bin/sh

a=0

while [ "$a" -lt 10 ] # this is loop1

do

b="$a"

while [ "$b" -ge 0 ] # this is loop2

do

echo -n "$b "

b=`expr $b - 1`

done

echo

a=`expr $a + 1`

done

This will produce the following result. It is important to note how **echo -n** works here. Here **-n** option lets echo avoid printing a new line character.

0

1 0

2 1 0

3 2 1 0

4 3 2 1 0

5 4 3 2 1 0

6 5 4 3 2 1 0

7 6 5 4 3 2 1 0

8 7 6 5 4 3 2 1 0

9 8 7 6 5 4 3 2 1 0

**Until Loops**

The **until** loop is fairly similar to the while loop. The difference is that it will execute the commands within it until the test becomes true.

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

**until [ <some test> ]**

**do  
<commands>  
done**

1. *# Basic until loop*
2. #!/bin/bash
3. counter=1
4. until [ $counter -gt 10 ]
5. do
6. echo $counter
7. ((counter++))
8. done
9. echo All done

## **For loop syntax**

Numeric ranges for syntax is as follows:

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

## **Examples**

|  |
| --- |
| *#!/bin/bash*  **for** i **in** 1 2 3 4 5  **do**  **echo** "Welcome $i times"  **done** |

Sometimes you may need to set a step value (allowing one to count by two’s or to count backwards for instance). Latest bash version 3.0+ has inbuilt support for setting up ranges:

|  |
| --- |
| *#!/bin/bash*  **for** i **in** **{**1..5**}**  **do**  **echo** "Welcome $i times"  **done** |

Bash v4.0+ has inbuilt support for setting up a step value **using {START**..**END**..**INCREMENT}** syntax:

|  |
| --- |
| *#!/bin/bash*  **echo** "Bash version ${BASH\_VERSION}..."  **for** i **in** **{**0..10..2**}**  **do**  **echo** "Welcome $i times"  **done** |

## **Three-expression bash for loops syntax**

This type of for loop share a common heritage with the C programming language. It is characterized by a three-parameter loop control expression; consisting of an initializer (EXP1), a loop-test or condition (EXP2), and a counting expression (EXP3).

|  |
| --- |
| **for** **((** EXP1; EXP2; EXP3 **))**  **do**  command1  command2  command3  **done** |

A representative three-expression example in bash as follows:

|  |
| --- |
| *#!/bin/bash*  **for** **((** c=1; c**<**=5; **c++** **))**  **do**  **echo** "Welcome $c times"  **done** |

## How do I use for as infinite loops?

Infinite for loop can be created with empty expressions, such as:

|  |
| --- |
| *#!/bin/bash*  **for** **((** ; ; **))**  **do**  **echo** "infinite loops [ hit CTRL+C to stop]"  **done** |

## **Conditional exit with break**

You can do early exit with break statement inside the for loop. You can exit from within a FOR, WHILE or UNTIL loop using break. General break statement inside the for loop:

|  |
| --- |
| **for** I **in** 1 2 3 4 5  **do**  statements1 *#Executed for all values of ''I'', up to a disaster-condition if any.*  statements2  **if** **(**disaster-condition**)**  **then**  **break** *#Abandon the loop.*  **fi**  statements3 *#While good and, no disaster-condition.*  **Done** |

### Early continuation with continue statement

To resume the next iteration of the enclosing FOR, WHILE or UNTIL loop use continue statement.

|  |
| --- |
| **for** I **in** 1 2 3 4 5  **do**  statements1 *#Executed for all values of ''I'', up to a disaster-condition if any.*  statements2  **if** **(**condition**)**  **then**  **continue** *#Go to next iteration of I in the loop and skip statements3*  **fi**  statements3  **done** |

This script make backup of all file names specified on command line. If .bak file exists, it will skip the cp command.

*#!/bin/bash*

**for** **file** **in** **/**etc**/\***

**do**

**if** **[** "${file}" == "/etc/resolv.conf" **]**

**then**

countNameservers=$**(grep** -c nameserver **/**etc**/**resolv.conf**)**

**echo** "Total ${countNameservers} nameservers defined in ${file}"

**break**

**fi**

**done**

**Factorial of a number**

## Write a shell script to find the factorial of number using for loop.

#!/bin/bash

fact=1

#taking input from user

echo -e "enter a number"

read n

#if enter value less than 0

if [ $n -le 0 ] ; then

echo "invalid number"

exit

fi

**factorial logic**

if [ $n -gt 0 ] ; then

for((i=$n;i>=1;i--))

do

fact=`expr $fact \\* $i`

done

fi

echo "The factorial of $n is $fact"

**The case...esac Statement**

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which has been described in detail here −

* case...esac statement

The case...esac statement in the Unix shell is very similar to the switch...case statement we have in other programming languages

## **Syntax**

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

**case** word **in**

pattern1)

Statement(s) to be executed if pattern1 matches

;;

pattern2)

Statement(s) to be executed if pattern2 matches

;;

pattern3)

Statement(s) to be executed if pattern3 matches

;;

\*)

Default condition to be executed

;;

esac

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

## **Example**

#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in

"apple") echo "Apple pie is quite tasty."

;;

"banana") echo "I like banana nut bread."

;;

"kiwi") echo "New Zealand is famous for kiwi."

;;

esac

**Upon execution, you will receive the following result −**

New Zealand is famous for kiwi.

A good use for a case statement is the evaluation of command line arguments as follows −

#!/bin/sh

option="${1}"

case ${option} in

-f) FILE="${2}"

echo "File name is $FILE"

;;

-d) DIR="${2}"

echo "Dir name is $DIR"

;;

\*)

echo "`basename ${0}`:usage: [-f file] | [-d directory]"

exit 1 # Command to come out of the program with status 1

;;

esac

Here is a sample run of the above program −

**$./test.sh**

**test.sh: usage: [ -f filename ] | [ -d directory ]**

**$ ./test.sh -f index.htm**

$ vi test.sh

$ ./test.sh -f index.htm

File name is index.htm

$ ./test.sh -d unix

Dir name is unix

**Uses in Init Scripts**

If you have ever edited any of init scripts, you must be seen that most of init scripts uses case statements for running commands like start, stop etc.

case "$1" in

start)

start

;;

stop)

stop

;;

status)

status myservice

;;

restart)

stop

start

;;

\*)

echo "Use command: service myservice {start|stop|restart|status}"

exit 1

esac

**Command line arguments: Suppose you are in your home directory which has sub-** **directories dir-1, dir-2, dir-3, dir-4 and dir-5. Write a shell script which uses touch command in it to create files fil-i in dir-i for i taking values from 1 to 5 using command line arguments.**

**How to read command line arguments in a bash script**

To input arguments into a Bash script, like any normal command line program, there are special variables set aside for this.

The arguments are stored in variables with a number in the order of the argument starting at 1

 First Argument: $1

 Second Argument: $2

 Third Argument: $3

 Example

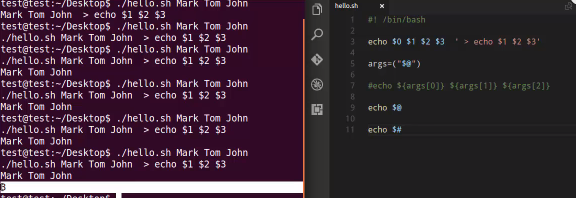
 command: ./script.bash alpha beta gamma

 Variables: $1=='alpha'; $2=='beta'; $3=='gamma'

The variable $0 is the script's name.

The total number of arguments is stored in $#.

The variables $@ and $\* return all the arguments.



**#!/bin/bash**

**cd /home/vivek**

**for i in 1 2 3**

**do**

**mkdir dir-$i**

**cd dir-$i**

**touch fil-$i**

**cd ..**

**done**

**#!/bin/bash**

**#echo $1 $2 $3 $4 $5**

**echo $#**

**echo $@**

**echo $\***

**Experiment**

**Write a Shell program to find the largest among three numbers.**

**PROGRAM**

echo “Enter the first number:” read a

echo “Enter the second number:” read b

echo “Enter the third number:” read c

if [ $a -gt $b -a $a -gt $c ] then

echo “$s is greater” elif [ $b -gt $c ]

then

echo “$b is greater”

else

echo “$c is greater”

fi

**Write a Shell program to find the largest among ‘n’ different numbers.**

**PROGRAM**

echo “Enter the number of elements:”

read n

l=0

for((i = 1 ; i <= n ; i++)) do

echo “Enter the number:” read no

if [ $no -gt $l ] then

l=$no

fi

done

echo “The largest numbers is : $l”

**String Comparison Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = or == | Is Equal To | if [ "$1" == "$2" ] |
| != | Is Not Equal To | if [ "$1" != "$2" ] |
| > | Is Greater Than (ASCII comparison) | if [ "$1" > "$2" ] |
| >= | Is Greater Than Or Equal To | if [ "$1" >= "$2" ] |
| < | Is Less Than | if [ "$1" < "$2" ] |
| <= | Is Less Than Or Equal To | if [ "$1" <= "$2" ] |
| -n | Is Not Null | if [ -n "$1" ] |
| -z | Is Null (Zero Length String) | if [ -z "$1"] |

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **=** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a = $b ] is not true. |
| **!=** | Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true. | [ $a != $b ] is true. |
| **-z** | Checks if the given string operand size is zero; if it is zero length, then it returns true. | [ -z $a ] is not true. |
| **-n** | Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true. | [ -n $a ] is not false. |
| **str** | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false. |

### String Operators | Shell Script

### There are many operators in Shell Script some of them are discussed based on string.

### Equal operator (=): This operator is used to check whether two strings are equal.

### Syntax:

|  |
| --- |
| Operands1 = Operand2Example:#!/bin/sh  str1="GeeksforGeeks";str2="geeks";if [ $str1 = $str2 ]then    echo "Both string are same";else    echo "Both string are not same";fi |
| Output:Both string are not same• Not Equal operator (!=): This operator is used when both operands are not equal.Syntax:Operands1 != Operands2Example:#!/bin/shstr1="GeeksforGeeks";str2="geeks";if [ $str1 != $str2 ]then    echo "Both string are not same";else    echo "Both string are same";fi |

### Output:

### Both string are not same

### • Less then (\<): It is a conditional operator and used to check operand1 is less then operand2.

### Syntax:

### Operand1 \< Operand2

### Example:

### #!/bin/sh

### str1="GeeksforGeeks";

### str2="Geeks";

### if [ $str1 \< $str2 ]

### then

### echo "$str1 is less then $str2";

### else

### echo "$str1 is not less then $str2";

### fi

### Output:

### GeeksforGeeks is not less then Geeks

### • Greater then (\>): This operator is used to check the operand1 is greater then operand2.

### Syntax:

### Operand1 \> Operand2

### Example:

### #!/bin/sh

### str1="GeeksforGeeks";

### str2="Geeks";

### if [ $str1 \> $str2 ]

### then

### echo "$str1 is greater then $str2";

### else

### echo "$str1 is less then $str2";

### fi

### Output:

### GeeksforGeeks is greater then Geeks

### • Check string length greater then 0: This operator is used to check the string is not empty.

### Syntax:

### [ -n Operand ]

### Example:

### #!/bin/sh

### str="GeeksforGeeks";

### if [ -n $str ]

### then

### echo "String is not empty";

### else

### echo "String is empty";

### fi

### Output:

### String is not empty

### • Check string length equal to 0: This operator is used to check the string is empty.

### Syntax:

### [ -z Operand ]

### Example:

### #!/bin/sh

### str="";

### if [ -z $str ]

### then

### echo "String is empty";

### else

### echo "String is not empty";

### fi

### Output:

### String is empty

## **File Test Operators**

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

[Show Examples](https://www.tutorialspoint.com/unix/unix-file-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true. | [ -k $file ] is false. |
| **-p file** | Checks if file is a named pipe; if yes, then the condition becomes true. | [ -p $file ] is false. |
| **-t file** | Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. | [ -t $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-s file** | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true. |

## **Example**

The following example uses all the **file test** operators −

Assume a variable file holds an existing file name **"/var/www/tutorialspoint/unix/test.sh"** the size of which is 100 bytes and has **read**, **write** and **execute** permission −

#!/bin/sh

file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]

then

echo "File has read access"

else

echo "File does not have read access"

fi

if [ -w $file ]

then

echo "File has write permission"

else

echo "File does not have write permission"

fi

if [ -x $file ]

then

echo "File has execute permission"

else

echo "File does not have execute permission"

fi

if [ -f $file ]

then

echo "File is an ordinary file"

else

echo "This is sepcial file"

fi

if [ -d $file ]

then

echo "File is a directory"

else

echo "This is not a directory"

fi

if [ -s $file ]

then

echo "File size is not zero"

else

echo "File size is zero"

fi

if [ -e $file ]

then

echo "File exists"

else

echo "File does not exist"

fi

The above script will produce the following result −

File does not have write permission

File does not have execute permission

This is sepcial file

This is not a directory

File size is not zero

File does not exist

The following points need to be considered while using file test operators −

* There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.

### Positional Parameter

A positional parameter is a variable within a shell program; its value is set from an argument specified on the command line that invokes the program. Positional parameters are numbered and are referred to with a preceding ``$'': **$1**, **$2**, **$3**, and so on.

A shell program may reference up to nine positional parameters. If a shell program is invoked with a command line that appears like this:

shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9

then positional parameter **$1** within the program is assigned the value **pp1**, positional parameter **$2** within the program is assigned the value **pp2**, and so on, at the time the shell program is invoked.

How Do I Access Command-Line Arguments

Create a simple shell script called cmdargs.sh:

*#!/bin/bash*

echo "The script name : $0"

echo "The value of the first argument to the script : $1"

echo "The value of the second argument to the script : $2"

echo "The value of the third argument to the script : $3"

echo "The number of arguments passed to the script : $#"

echo "The value of all command-line arguments (\$\* version) : $\*"

echo "The value of all command-line arguments (\$@ version) : $@"

Save and close the file. Rut it as follows:

chmod +x cmdargs.sh

./cmdargs.sh bmw ford toyota

Sample outputs:

The script name : ./cmdargs.sh

The value of the first argument to the script : bmw

The value of the second argument to the script : ford

The value of the third argument to the script : toyota

The number of arguments passed to the script : 3

The value of all command-line arguments ($\* version) : bmw ford toyota

The value of all command-line arguments ($@ version) : bmw ford toyota

The variable **$@** contains the value of all positional parameters, excluding $0.

The variable **$\*** is the same as **$@**, except when it is double-quoted. When enclosed in double quotes, **$\*** expands to **$1c$2c$3c**... where *c* is the first character of **$IFS**, bash's internal field separator variable. The IFS is used for word splitting, and its default value is "space, tab, or newline" — this is where bash sees the beginning and end of one word.

If the value of **$IFS** is "**\_**" (an underscore), **"$@"** expands to:

one two three four

Whereas **"$\*"** expands to:

one\_two\_three four

**Example: The Difference Between $@ and $\***

Create a shell script called pizza.sh:

*#!/bin/bash*

IFS=", "

echo "\* Displaying all pizza names using \$@"

echo "$@"

echo

echo "\* Displaying all pizza names using \$\*"

echo "$\*"

Save and close the file. Run it as follows:

chmod +x pizza.sh

./pizza.sh Margherita Tomato Panner Gourmet

**Sample outputs:**

**\* Displaying all pizza names using $@**

Margherita Tomato Panner Gourmet

**\*Displaying all pizza names using $\***

Margherita,Tomato,Panner,Gourmet

**Using if loop to Check Command-Line Arguments**

The following script captures the command-line arguments in special variables: $#, $1, $2, $3, and so on. The $# variable captures the number of command-line arguments. The following example use the command line arguments to do a comparison.

#!/bin/bash

if [ $1 -gt $2 ]

then

echo "num1 is larger"

else

echo "num2 is larger"

fi

You can run the above script as shown below.

$ ./a 23 45

num2 is larger

Often, you will want to check to see if you have arguments on which to act. There are a couple of ways to do this. First, you could simply check to see if $1 contains anything like so:

#!/bin/bash

**if** **[** "$1" != "" **];** **then**

**echo** "Positional parameter 1 contains something"

**else**

**echo** "Positional parameter 1 is empty"

**fi**

Second, the shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command ($0).

#!/bin/bash

**if** **[** $# -gt 0 **];** **then**

**echo** "Your **command** line contains $# arguments"

**else**

**echo** "Your **command** line contains no arguments"

**fi**

## **Exit Status**

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

**$./test.sh Zara Ali**

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

$echo $?

0

$

### Example

### let we make a script parameter.sh as below and run the script with four arguments page, linux, dot, com

### $ cat parameter.sh

### #!/bin/bash

### IFS=-

### echo $0

### echo $1

### echo $2

### echo $\*

### echo $#

### echo $@

### echo "$\*"

### echo "$@"

### echo $?

### unset IFS

### echo "$\*"

### After running the script we get the output as

### $ sh parameter.sh page linux dot com

### ./parameter.sh

### page

### linux

### page linux dot com

### 4

### page linux dot com

### page-linux-dot-com

### page linux dot com

### 0

### page linux dot com

### Array

Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

**How to Declare Array in Shell Scripting?**  
We can declare an array in a shell script in different ways.

1. Indirect Declaration

In Indirect declaration, We assigned a value in a particular index of Array Variable. No need to first declare.

ARRAYNAME[INDEXNR]=value

1. Explicit Declaration

In Explicit Declaration, First We declare array then assigned the values.

declare -a ARRAYNAME

1. Compound Assignment

In Compount Assignment, We declare array with a bunch of values. We can add other values later too.

ARRAYNAME=(value1 value2 .... valueN)

or  
[indexnumber=]string

ARRAYNAME=([1]=10 [2]=20 [3]=30)

## **Defining Array Values**

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

NAME01="Zara"

NAME02="Qadir"

NAME03="Mahnaz"

NAME04="Ayan"

NAME05="Daisy"

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

array\_name[index]=value

Here *array\_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands −

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

If you are using the **ksh** shell, here is the syntax of array initialization −

set -A array\_name value1 value2 ... valuen

If you are using the **bash** shell, here is the syntax of array initialization −

array\_name=(value1 ... valuen)

## **Accessing Array Values**

After you have set any array variable, you access it as follows −

${array\_name[index]}

Here *array\_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept −

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Index: ${NAME[0]}"

echo "Second Index: ${NAME[1]}"

The above example will generate the following result −

$./test.sh

First Index: Zara

Second Index: Qadir

You can access all the items in an array in one of the following ways −

${array\_name[\*]}

${array\_name[@]}

Here **array\_name** is the name of the array you are interested in. Following example will help you understand the concept −

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Method: ${NAME[\*]}"

echo "Second Method: ${NAME[@]}"

The above example will generate the following result −

$./test.sh

First Method: Zara Qadir Mahnaz Ayan Daisy

Second Method: Zara Qadir Mahnaz Ayan Daisy

**Various Operations on Arrays**

Many of the standard string operations work on arrays . Look at the following sample script which implements some operations on arrays (including string operations).

#!/bin/bash

array=( apple bat cat dog elephant frog )

#print first element

echo ${array[0]}

echo ${array:0}

#display all elements

echo ${array[@]}

echo ${array[@]:0}

#display all elements except first one

echo ${array[@]:1}

#display elements in a range

echo ${array[@]:1:4}

#length of first element

echo ${#array[0]}

echo ${#array}

#number of elements

echo ${#array[\*]}

echo ${#array[@]}

#replacing substring

echo ${array[@]//a/A}

exit 0

Following is the output produced on executing the above script.

apple

apple

apple bat cat dog elephant frog

apple bat cat dog elephant frog

bat cat dog elephant frog

bat cat dog elephant

5

5

6

6

Apple bAt cAt dog elephAnt frog

To traverse through the array elements we can also use for loop.

for i in “${array[@]}”

do

#access each element as $i. . .

done

### By Using while-loop

|  |
| --- |
| # !/bin/bash  # To declare static Array  arr=(1 12 31 4 5)  i=0    # Loop upto size of array  # starting from index, i=0  while [ $i -lt ${#arr[@]} ]  do      # To print index, ith      # element      echo ${arr[$i]}        # Increment the i = i + 1      i=`expr $i + 1`  done |

Output:

1

2

3

4

5

### Alias Command in linux

### What is a shell alias?

### A shell alias is a shortcut to reference a command. It can be used to avoid typing long commands or as a means to correct incorrect input. For common patterns it can reduce keystrokes and improve efficiency. A simple example is setting default options on commands to avoid having to type them each time a command is run.

### Why create a shell alias?

### For the following example suppose that a user prefers to confirm deleting a file before using the rm command. The rm command supports this with the -i option.

### rm -i file.txt

### remove file.txt? y

### To avoid forgetting to use the -i option each time an alias can be created so that each time rm is run it will use the -i option and prompt the user to confirm.

### How to set an shell alias

### Building on the previous example an alias can be directly set in the shell as follows.

### alias rm='rm -i'

### Now when the rm command is run it will use the alias and the -i option.

### rm file.txt

### remove file.txt?

### Note that setting an alias in this way only works for the life of a shell session. When the shell is closed the alias will be lost. To make an alias persist across shell sessions and reboots a configuration file for the shell should be used. For bash this is the .bashrc file. If you are using zsh it is the .zshrc file.

### The .bashrc file

### A .bashrc file can be used to set configuration for a shell. In this example a local user’s .bashrc file will be used. If you are running the zsh shell use a file called .zshrc.

### Open the ~/.bashrc file in your preferred text editor. If it does not exist create it.

### touch ~/.bashrc

### How to add a shell alias to .bashrc

### Within the .bashrc files aliases can now be added. The format is simple. First declare the command you wish to alias, then specify the command to run instead.

### alias rm='rm -i'

### For this example we replace rm with rm -i so that the user is prompted before deleting the file.

### Once the .bashrc file is saved the shell needs to be reloaded for the alias to take effect.

### source ~/.bashrc

### The alias should now be available and typing rm will be interpreted as rm -i.

### Some examples of aliases

### The following are some practical examples of using aliases.

### # ensure git commits are signed

### alias git commit='git commit -S'

### # shorthand for vim

### alias v="vim"

### # setting preferred options on ls

### alias ls='ls -lhF'

### # prompt user if overwriting during copy

### alias cp='cp -i'

### # prompt user when deleting a file

### alias rm='rm -i'

### # always print in human readable form

### alias df="df -h"

### Creating an Unalias :

### Removing an existing alias is known as unaliasing.

### Syntax:

### unalias [alias name]

### chsh command in Linux with examples

### chsh command in Linux is used to change the user’s login shell(currently login shell). Shell is an interactive user interface with an operating system and can be considered an outer layer of the operating system. The bash shell is one of the most widely used login shells in Linux. This command allows the user to change the shell from the current shell. It can also give warning if the shell is not present in the*/etc/shells* file. The superuser can change the login shell for the existing accounts.

### Syntax:

### chsh [OPTIONS] [LOGIN]

### Example 1: To show the list of all shells.

### You can use echo command along with ‘$SHELL’ to check the current shell

### Or

### $ chsh -l

### Example 2: To change the current loging shell

### $ echo $SHELL

### /bin/bash

### $ chsh –s /bin/dash

### $ echo $SHELL

### /bin/dash

### Or

### $ chsh

### Password: (provide the root password and write the shell name)

### /bin/dash (Shell name to which you want to switch)

### $ echo $SHELL

### /bin/dash

### [[[[

### Options:

### • -l: Used to specifies your login shell.

### • -u: Prints the list of shells.

### • -v: Shows information about version and exits.

### • -s: Used to set the shell as your login shell

### AWK command in Unix/Linux with examples

### Awk is a scripting language used for manipulating data and generating reports.The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

### Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

### Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan.

### WHAT CAN WE DO WITH AWK ?

### 1. AWK Operations:

### (a) Scans a file line by line

### (b) Splits each input line into fields

### (c) Compares input line/fields to pattern

### (d) Performs action(s) on matched lines

### 2. Useful For:

### (a) Transform data files

### (b) Produce formatted reports

### 3. Programming Constructs:

### (a) Format output lines

### (b) Arithmetic and string operations

### (c) Conditionals and loops

### Syntax:

### awk options 'selection \_criteria {action }' input-file > output-file

### Options:

### -f program-file : Reads the AWK program source from the file program-file, instead of from the

### first command line argument.

### -F fs : Use fs for the input field separator

### Sample Commands

### Example:

### Consider the following text file as the input file for all cases below.

### $cat > employee.txt

### ajay manager account 45000

### sunil clerk account 25000

### varun manager sales 50000

### amit manager account 47000

### tarun peon sales 15000

### deepak clerk sales 23000

### sunil peon sales 13000

### satvik director purchase 80000

### 1. Default behavior of Awk : By default Awk prints every line of data from the specified file.

### $ awk '{print}' employee.txt

### Output

### ajay manager account 45000

### sunil clerk account 25000

### varun manager sales 50000

### amit manager account 47000

### tarun peon sales 15000

### deepak clerk sales 23000

### sunil peon sales 13000

### satvik director purchase 80000

### In the above example, no pattern is given. So the actions are applicable to all the lines. Action print without any argument prints the whole line by default, so it prints all the lines of the file without failure.

### 2. Print the lines which matches with the given pattern.

### $ awk '/manager/ {print}' employee.txt

### Output:

### ajay manager account 45000

### varun manager sales 50000

### amit manager account 47000

### In the above example, the awk command prints all the line which matches with the ‘manager’.

### 3. Spliting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $0 represents the whole line.

### $ awk '{print $1,$4}' employee.txt

### Output:

### ajay 45000

### sunil 25000

### varun 50000

### amit 47000

### tarun 15000

### deepak 23000

### sunil 13000

### satvik 80000

### In the above example, $1 and $4 represents Name and Salary fields respectively.

### Built In Variables In Awk

### Awk’s built-in variables include the field variables—$1, $2, $3, and so on ($0 is the entire line) — that break a line of text into individual words or pieces called fields.

### NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

### NF: NF command keeps a count of the number of fields within the current input record.

### FS: FS command contains the field separator character which is used to divide fields on the input line. The default is “white space”, meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

### RS: RS command stores the current record separator character. Since, by default, an input line is the input record, the default record separator character is a newline.

### OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

### ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

### Examples:

### Use of NR built-in variables (Display Line Number)

### $ awk '{print NR,$0}' employee.txt

### Output:

### 1 ajay manager account 45000

### 2 sunil clerk account 25000

### 3 varun manager sales 50000

### 4 amit manager account 47000

### 5 tarun peon sales 15000

### 6 deepak clerk sales 23000

### 7 sunil peon sales 13000

### 8 satvik director purchase 80000

### In the above example, the awk command with NR prints all the lines along with the line number.

### Use of NF built-in variables (Display Last Field)

### $ awk '{print $1,$NF}' employee.txt

### Output:

### ajay 45000

### sunil 25000

### varun 50000

### amit 47000

### tarun 15000

### deepak 23000

### sunil 13000

### satvik 80000

### In the above example $1 represents Name and $NF represents Salary. We can get the Salary using $NF , where $NF represents last field.

### Another use of NR built-in variables (Display Line From 3 to 6)

### $ awk 'NR==3, NR==6 {print NR,$0}' employee.txt

### Output:

### 3 varun manager sales 50000

### 4 amit manager account 47000

### 5 tarun peon sales 15000

### 6 deepak clerk sales 23000

### More Examples

### For the given text file:

### $cat > geeksforgeeks.txt

### A B C

### Tarun A12 1

### Man B6 2

### Praveen M42 3

### 1) To print the first item along with the row number(NR) separated with ” – “ from each line in geeksforgeeks.txt:

### $ awk '{print NR "- " $1 }' geeksforgeeks.txt

### 1 - Tarun

### 2 – Manav

### 3 - Praveen

### 2) To return the second row/item from geeksforgeeks.txt:

### $ awk '{print $2}' geeksforgeeks.txt

### A12

### B6

### M42

### 3) To print any non empty line if present

### $ awk 'NF > 0' geeksforgeeks.txt

### 0

### 4) To count the lines in a file:

### $ awk 'END { print NR }' geeksforgeeks.txt

### 3

### Environment Varriables

### Introduction

When interacting with your server through a shell session, there are many pieces of information that your shell compiles to determine its behavior and access to resources. Some of these settings are contained within configuration settings and others are determined by user input.

One way that the shell keeps track of all of these settings and details is through an area it maintains called the **environment**. The environment is an area that the shell builds every time that it starts a session that contains variables that define system properties.

In this guide, we will discuss how to interact with the environment and read or set environmental and shell variables interactively and through configuration files. We will be using an Ubuntu 12.04 VPS as an example, but these details should be relevant on any Linux system.

## **How the Environment and Environmental Variables Work**

Every time a shell session spawns, a process takes place to gather and compile information that should be available to the shell process and its child processes. It obtains the data for these settings from a variety of different files and settings on the system.

Basically the environment provides a medium through which the shell process can get or set settings and, in turn, pass these on to its child processes.

The environment is implemented as strings that represent key-value pairs. If multiple values are passed, they are typically separated by colon (:) characters. Each pair will generally look something like this:

KEY=value1:value2:...

If the value contains significant white-space, quotations are used:

KEY="value with spaces"

The keys in these scenarios are variables. They can be one of two types, environmental variables or shell variables.

**Environmental variables** are variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell.

**Shell variables** are variables that are contained exclusively within the shell in which they were set or defined. They are often used to keep track of ephemeral data, like the current working directory.

By convention, these types of variables are usually defined using all capital letters. This helps users distinguish environmental variables within other contexts.

## **Printing Shell and Environmental Variables**

Each shell session keeps track of its own shell and environmental variables. We can access these in a few different ways.

We can see a list of all of our environmental variables by using the env or printenv commands. In their default state, they should function exactly the same:

printenv

SHELL=/bin/bash

TERM=xterm

USER=demouser

LS\_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:su=37;41:sg=30;43:ca:...

MAIL=/var/mail/demouser

PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

PWD=/home/demouser

LANG=en\_US.UTF-8

SHLVL=1

HOME=/home/demouser

LOGNAME=demouser

LESSOPEN=| /usr/bin/lesspipe %s

LESSCLOSE=/usr/bin/lesspipe %s %s

\_=/usr/bin/printenv

This is fairly typical of the output of both printenv and env. The difference between the two commands is only apparent in their more specific functionality. For instance, with printenv, you can requests the values of individual variables:

printenv SHELL

/bin/bash

On the other hand, env let’s you modify the environment that programs run in by passing a set of variable definitions into a command like this:

env VAR1="blahblah" command\_to\_run command\_options

Since, as we learned above, child processes typically inherit the environmental variables of the parent process, this gives you the opportunity to override values or add additional variables for the child.

As you can see from the output of our printenv command, there are quite a few environmental variables set up through our system files and processes without our input.

These show the environmental variables, but how do we see shell variables?

The set command can be used for this. If we type set without any additional parameters, we will get a list of all shell variables, environmental variables, local variables, and shell functions:

set

### If we want to completely unset a variable, either shell or environmental, we can do so with the unset command:

### unset TEST\_VAR

### We can verify that it is no longer set:

### echo $TEST\_VAR

### Nothing is returned because the variable has been unset.

### The Difference between Login, Non-Login, Interactive, and Non-Interactive Shell Sessions

### The bash shell reads different configuration files depending on how the session is started.

### One distinction between different sessions is whether the shell is being spawned as a “login” or “non-login” session.

### A login shell is a shell session that begins by authenticating the user. If you are signing into a terminal session or through SSH and authenticate, your shell session will be set as a “login” shell.

### If you start a new shell session from within your authenticated session, like we did by calling the bash command from the terminal, a non-login shell session is started. You were were not asked for your authentication details when you started your child shell.

### Another distinction that can be made is whether a shell session is interactive, or non-interactive.

### An interactive shell session is a shell session that is attached to a terminal. A non-interactive shell session is one is not attached to a terminal session.

### So each shell session is classified as either login or non-login and interactive or non-interactive.

### A normal session that begins with SSH is usually an interactive login shell. A script run from the command line is usually run in a non-interactive, non-login shell. A terminal session can be any combination of these two properties.

### Whether a shell session is classified as a login or non-login shell has implications on which files are read to initialize the shell session.

### A session started as a login session will read configuration details from the /etc/profile file first. It will then look for the first login shell configuration file in the user’s home directory to get user-specific configuration details.

### It reads the first file that it can find out of ~/.bash\_profile, ~/.bash\_login, and ~/.profile and does not read any further files.

### In contrast, a session defined as a non-login shell will read /etc/bash.bashrc and then the user-specific ~/.bashrc file to build its environment.

### Non-interactive shells read the environmental variable called BASH\_ENV and read the file specified to define the new environment.

### Implementing Environmental Variables

### As you can see, there are a variety of different files that we would usually need to look at for placing our settings.

### This provides a lot of flexibility that can help in specific situations where we want certain settings in a login shell, and other settings in a non-login shell. However, most of the time we will want the same settings in both situations.

### Fortunately, most Linux distributions configure the login configuration files to source the non-login configuration files. This means that you can define environmental variables that you want in both inside the non-login configuration files. They will then be read in both scenarios.

### We will usually be setting user-specific environmental variables, and we usually will want our settings to be available in both login and non-login shells. This means that the place to define these variables is in the ~/.bashrc file.

### Open this file now:

### nano ~/.bashrc

### This will most likely contain quite a bit of data already. Most of the definitions here are for setting bash options, which are unrelated to environmental variables. You can set environmental variables just like you would from the command line:

### export VARNAME=value

### Any new environmental variables can be added anywhere in the ~/.bashrc file, as long as they aren’t placed in the middle of another command or for loop. We can then save and close the file. The next time you start a shell session, your environmental variable declaration will be read and passed on to the shell environment. You can force your current session to read the file now by typing:

### source ~/.bashrc

### If you need to set system-wide variables, you may want to think about adding them to /etc/profile, /etc/bash.bashrc, or /etc/environment.

Let's study some common environment variables -

|  |  |
| --- | --- |
| **Variable** | **Description** |
| **PATH** | This variable contains a colon (:)-separated list of directories in which your system looks for executable files. [Linux - Environment Variables](https://www.guru99.com/images/echoPath.png) When you enter a command on terminal, the shell looks for the command in different directories mentioned in the $PATH variable. If the command is found, it executes. Otherwise, it returns with an error 'command not found'. |
| **USER** | The username |
| **HOME** | Default path to the user's home directory |
| **EDITOR** | Path to the program which edits the content of files |
| **UID** | User's unique ID |
| **TERM** | Default terminal emulator |
| **SHELL** | Shell being used by the user |

### Setting Shell and Environmental Variables

### To better understand the difference between shell and environmental variables, and to introduce the syntax for setting these variables, we will do a small demonstration.

### Creating Shell Variables

### We will begin by defining a shell variable within our current session. This is easy to accomplish; we only need to specify a name and a value. We’ll adhere to the convention of keeping all caps for the variable name, and set it to a simple string.

### TEST\_VAR='Hello World!'

### Here, we’ve used quotations since the value of our variable contains a space. Furthermore, we’ve used single quotes because the exclamation point is a special character in the bash shell that normally expands to the bash history if it is not escaped or put into single quotes.

### We now have a shell variable. This variable is available in our current session, but will not be passed down to child processes.

### We can see this by grepping for our new variable within the set output:

### set | grep TEST\_VAR

### TEST\_VAR='Hello World!'

### We can verify that this is not an environmental variable by trying the same thing with printenv:

### printenv | grep TEST\_VAR

### No output should be returned.

### Let’s take this as an opportunity to demonstrate a way of accessing the value of any shell or environmental variable.

### echo $TEST\_VAR

### Hello World!

### If we want to completely unset a variable, either shell or environmental, we can do so with the unset command:

### unset TEST\_VAR

### We can verify that it is no longer set:

### echo $TEST\_VAR

### Nothing is returned because the variable has been unset.

### ssh command in linux

### SSH (SSH client) is a program for remotely accessing a machine, it enables a user to execute commands on a remote host. It is one of the most recommended method for logging in to a remote host, since it is designed to provide secure encrypted communications between two untrusted hosts over an insecure network.

### SSH uses both a system-wide as well as a user-specific (custom) configuration file. In this tutorial, we will explain how to create a custom ssh configuration file and use certain options to connect to remote hosts.

### SSH Client Config Files

### Below are the locations of the ssh client configuration files:

### /etc/ssh/ssh\_config – this is the default, system-wide configuration file. It contains settings that apply to all users of ssh client machine.

### ~/.ssh/config or $HOME/.ssh/config – is the user-specific/custom configuration file. It has configurations that apply to a specific user. It therefore overrides default settings in the system-wide config file. This is the file we will create and use.

### How To Create User Specific SSH Configuration File

### This file is usually not created by default, so you need to create it with the read/write permissions for only the user.

### $ touch ~/.ssh/config

### $ chmod 0700 ~/.ssh/config

### The above file contains sections defined by hosts specifications, and a section is only applied to hosts that match one of the patterns set in the specification.

### The conventional format of ~/.ssh/config is as follows, and all empty lines as well as lines starting with ‘#’ are considered as comments:

### Host host1

### ssh\_option1=value1

### ssh\_option2=value1 value2

### ssh\_option3=value1

### Host host2

### ssh\_option1=value1

### ssh\_option2=value1 value2

### Host \*

### ssh\_option1=value1

### ssh\_option2=value1 value2

### From the format above:

### Host host1 – is a header definition for host1, this is where a host specification starts and it ends with the next header definition, Host host2 making a section.

### host1, host2 are simply host aliases to use on the command line, they are not the actual hostnames of the remote hosts.

### The configuration options such as ssh\_option1=value1, ssh\_option2=value1 value2 apply to a matched host and should be indented for well organized formatting.

### For an option such as ssh\_option2=value1 value2, the value value1 is considered first, then value2.

### The header definition Host \* (where \* is a pattern – wildcard that matches zero or more characters) will match zero or more hosts.

### Still considering the format above, this is how ssh reads the config file. If you execute a ssh command to remotely access host1 like so:

### $ ssh host1

### The above ssh command will does the following things:

### match the host alias host1 in the config file and applies the options set under the definition header, Host host1.

### then moves to the next host section, Host host2 and finds that the name provided on the command line doesn’t match, so no options are used from here.

### It proceeds to the last section, Host \*, which matches all hosts. Here, it applies all the options in this section to the host connection. But it can not override any values of options that where already used in the previous section(s).

### The same applies to host2.

## **How to Access a Remote Server**

To connect to a remote machine, you need its IP address or name. Load the terminal or any SSH client and type ssh followed by the IP address:

ssh 192.168.56.101

or name:

ssh test.server.com

### Specify a Username for SSH connection

SSH uses the current user when accessing a remote server. To specify a user for an SSH connection, run the command in this format:

ssh username@hostname\_or\_ip

For instance:

ssh testuser@10.0.0.55

## Use a Different Port Number for SSH Connection

By default, the SSH server listens for a connection on port 22. If the port setting in the SSH config file has been changed, you’ll need to specify the port. Otherwise, you will get this error:

To connect to a remote host with a custom SSH port number, use the **-p**flag. For example:

ssh test.server.com -p 3322

### Generate SSH Keys Using SSH Keygen

### To improve the security of SSH connections, generate a key pair with the keygen utility. The pair consists of a public and private key. The public key can be shared, while the private key needs to stay secure.

### SSH key pairs are used to authenticate clients to servers automatically. When you create an SSH key pair, there is no longer a need to enter a password to access a server.

### On the host machine’s terminal, use this command to create a key pair:

### ssh-keygen -t rsa

### To use default settings, hit Enter on the prompts for file location and passphrase.

### Copy Public SSH Key

### To use the key pair for SSH authentication, you’ll need to copy the public key to a server. The key is the file *id\_rsa.pub* previously created with SSH keygen utility.

### To copy your key to a server, run this command from the client:

### ssh-copy-id *hostname\_or\_IP*

### You can also specify a username if you don’t want to use the current user.

### Enter the password to authenticate when asked. After this, you will no longer need to use the password to connect to the same server.

### SCP command in linux

### scp (secure copy) command in Linux system is used to copy file(s) between servers in a secure way. The SCP command or secure copy allows secure transferring of files in between the local host and the remote host or between two remote hosts. It uses the same authentication and security as it is used in the Secure Shell (SSH) protocol. SCP is known for its simplicity, security and pre-installed availability.

### Copy a File Remotely over SSH with SCP

### You can securely copy files over the SSH protocol using the SCP tool. The basic syntax is:

### scp *fileName* *user@remotehost:/home/username/destination*

### For example, to copy a file *sample3*to your Desktop on a remote server with a username *test*, type in:

### scp sample3 test@10.0.10.5:/home/test/Desktop

### The output shows a summary of the operation.

### Make sure to use the uppercase -P flag if you need to specify the port.

### Restart SSH service

### When you make changes to the SSH configuration, you’ll need to restart the service.

### Depending on the Linux distro, run one of the following commands on the machine where you modified the settings:

### sudo ssh service restart

### or:

### sudo sshd service restart

### Finally, enter the password to complete the process. As a result, the next SSH session will use the new settings.

### Array Initialization and Usage

With newer versions of bash, it supports one-dimensional arrays. An array can be explicitly declared by the declare shell-builtin.

declare -a var

But it is not necessary to declare array variables as above. We can insert individual elements to array directly as follows.

var[XX]=<value>

Another convenient way of initializing an entire array is by using the pair of parenthesis as shown below.

var=( element1 element2 element3 . . . elementN )

There is yet another way of assigning values to arrays. This way of initialization is a sub-category of the previously explained method.

array=( [XX]=<value> [XX]=<value> . . . )

We can also read/assign values to array during the execution time using the read shell-builtin.

read -a array

Now upon executing the above statement inside a script, it waits for some input. We need to provide the array elements separated by space (and not carriage return). After entering the values press enter to terminate.

To traverse through the array elements we can also use for loop

## EXPERIMENT 30

## Write shell script to show various system configuration like

## 1) Currently logged user and his logname 2) Your current shell 3) Your home directory 4) Your operating system type 5) Your current path setting

#!/bin/bash

echo -e "User name: $USER

echo -e "shell name: $bash

echo -e "Current Shell: $SHELL"

echo -e "Home Directory: $HOME"

echo -e "Your O/s Type: $OSTYPE"

echo -e "PATH: $PATH"

echo -e "Current directory: $pwd`"