**Unix shell script**

What is shell?

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Computer understand the language of 0's and 1's called binary language.

In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is pass to kernel.

Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

|  |  |  |  |
| --- | --- | --- | --- |
| Shell Name | Developed by | Where | Remark |
| BASH ( Bourne-Again SHell ) | Brian Fox and Chet Ramey | Free Software Foundation | Most common shell in Linux. It's Freeware shell. |
| CSH (C SHell) | Bill Joy | University of California (For BSD) | The C shell's syntax and usage are very similar to the C programming language. |
| KSH (Korn SHell) | David Korn | AT & T Bell Labs | -- |
| TCSH | See the man page. Type $ man tcsh | -- | TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH). |

Tip: To find all available shells in your system type following command:  
$ **cat /etc/shells**

Note that each shell does the same job, but each understand a different command syntax and provides different built-in functions.

In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!

Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux Os what users want. If we are giving commands from keyboard it is called command line interface ( Usually in-front of $ prompt, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt ).

Tip: To find your current shell type following command  
**$ echo $SHELL**

**Shell Types**

In UNIX there are two major types of shells:

* The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
* The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows −

* Bourne shell ( sh)
* Korn shell ( ksh)
* Bourne Again shell ( bash)
* POSIX shell ( sh)

The different C-type shells follow −

* C shell ( csh)
* TENEX/TOPS C shell ( tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".

The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.

**Unix Architecture**

****

The main concept that unites all versions of UNIX is the following four basics:

** Kernel:** The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, tash scheduling and file management.

** Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the Unix variants.

** Commands and Utilities:** There are various command and utilities which you would use in your day to day activities. cp, mv, cat and grep etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.

** Files and Directories:** All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

**Variables**

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

**Variable Names**

The name of a variable can contain only letters ( a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

By convention, Unix Shell variables would have their names in UPPERCASE.

The following examples are valid variable names −

\_ALI

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invalid variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as !,\*, or - is that these characters have a special meaning for the shell.

Variables are referenced like this: $name, here is an example:

|  |
| --- |
| #!/bin/sh  msg1=Hello  msg2=There!  echo $msg1 $msg2 |

## Variable Types

When a shell is running, three main types of variables are present −

* **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.
* **Environment Variables** − An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
* **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

|  |  |
| --- | --- |
| **Variables** | **Description** |
| **$0** | The filename of the current script. |
| **$n** | The filename of the current script. These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| **$#** | The number of arguments supplied to a script. |
| **$\*** | All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| **$@** | All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| **$?** | The exit status of the last command executed. |
| **$$** | The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| **$!** | The process number of the last background command. |

Command-Line Arguments

The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to command line −

#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1"

echo "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

Here is a sample run for the above script −

$./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

Example :

suresh@ubuntu:~/Unix/Shell$ more specialvaribles.sh

#!/bin/bash

echo " check here special varibles"

echo "\$0" `basename $0`

echo "\$1" $1

echo "\$2" $2

echo "\$@" $@

echo "\$#" $#

echo "\$$" $$

lssl

echo "\$?" $?

sh /home/suresh/Unix/Demo.sh &

echo "\$!" $!

echo "\$\*" $\*

**Output :**

suresh@ubuntu:~/Unix/Shell$ ./specialvaribles.sh raja 10

check here special varibles

$0 specialvaribles.sh

$1 raja

$2 10

$@ raja 10

$# 2

$$ 3945

./specialvaribles.sh: line 9: lssl: command not found

$? 127

$! 3948

$\* raja 10

**Example**

#!/bin/sh

if [ "$1" = "1" ]

then

echo "The first choice is nice"

elif [ "$1" = "2" ]

then

echo "The second choice is just as nice"

elif [ "$1" = "3" ]

then

echo "The third choice is excellent"

else

echo "I see you were wise enough not to choose"

echo "You win"

fi

**Defining Array Values**

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

Say that 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 is to assign a value to one of its indices. This is expressed as follows −

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 **ksh** shell the here is the syntax of array initialization −

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

If you are using **bash** shell the 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 the simplest example −

#!/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]}"

This would produce 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 is the simplest example −

#!/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[@]}"

This would produce following result −

$./test.sh

First Method: Zara Qadir Mahnaz Ayan Daisy

Second Method: Zara Qadir Mahnaz Ayan Daisy

**Man Page Sections:**

Man pages are generally divided into sections, which generally vary by the man page author's preference. Here are some of the more common sections −

|  |  |
| --- | --- |
| **Section** | **Description** |
| NAME | Name of the command |
| SYNOPSIS | General usage parameters of the command. |
| DESCRIPTION | Generally describes of the command and what it does |
| OPTIONS | Describes all the arguments or options to the command |
| SEE ALSO | Lists other commands that are directly related to the command in the man page or closely resembling its functionality. |
| BUGS | Explains any known issues or bugs that exist with the command or its output |
| EXAMPLES | Common usage examples that give the reader an idea of how the command can be used. |
| AUTHORS | The author of the man page/command. |

**Directory Structure:**

Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

A UNIX file system is a collection of files and directories that has the following properties −

* It has a root directory (/) that contains other files and directories.
* Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an inode.
* By convention, the root directory has an inode number of 2 and the lost+found directory has an inode number of 3. Inode numbers 0 and 1 are not used. File inode numbers can be seen by specifying the -i option to ls command.
* It is self contained. There are no dependencies between one filesystem and any other.

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix −

|  |  |
| --- | --- |
| **Directory** | **Description** |
| / | This is the root directory which should contain only the directories needed at the top level of the file structure. |
| /bin | This is where the executable files are located. They are available to all user. |
| /dev | These are device drivers. |
| /etc | Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages. |
| /lib | Contains shared library files and sometimes other kernel-related files. |
| /boot | Contains files for booting the system. |
| /home | Contains the home directory for users and other accounts. |
| /mnt | Used to mount other temporary file systems, such as cdrom and floppy for the CD-ROM drive and floppy diskette drive, respectively |
| /proc | Contains all processes marked as a file by process number or other information that is dynamic to the system. |
| /tmp | Holds temporary files used between system boots |
| /usr | Used for miscellaneous purposes, or can be used by many users. Includes administrative commands, shared files, library files, and others |
| /var | Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data |
| /sbin | Contains binary (executable) files, usually for system administration. For example *fdisk* and *ifconfig* utlities. |
| /kernel | Contains kernel files |

**Operators:**

There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.

There are following operators which we are going to discuss −

* Arithmetic Operators.
* Relational Operators.
* Boolean Operators.
* String Operators.
* File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

Here is simple example to add two numbers −

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

This would produce following result −

Total value : 4

## Arithmetic Operators:

There are following arithmetic operators supported by Bourne Shell.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Addition - Adds values on either side of the operator |
| - | Subtraction - Subtracts right hand operand from left hand operand |
| \* | Multiplication - Multiplies values on either side of the operator |
| / | Division - Divides left hand operand by right hand operand |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder |
| = | Assignment - Assign right operand in left operand |
| == | Equality - Compares two numbers, if both are same then returns true. |
| != | Not Equality - Compares two numbers, if both are different then returns true. |

# Unix - Shell Arithmetic Operators Example

#!/bin/bash

a=10

b=20

val=`expr $a + $b`

echo "a + b : $val"

val=`expr $a - $b`

echo "a - b : $val"

val=`expr $a \\* $b`

echo "a \* b : $val"

val=`expr $b / $a`

echo "b / a : $val"

val=`expr $b % $a`

echo "b % a : $val"

if [ $a == $b ]

then

echo "a is equal to b"

fi

if [ $a != $b ]

then

echo "a is not equal to b"

fi

This would produce following result −

a + b : 30

a - b : -10

a \* b : 200

b / a : 2

b % a : 0

a is not equal to b

**Example:**

echo "Addition Program"

echo "enetr a values : "

read a

echo "enetr b values : "

read b

echo "enetr c values : "

read c

echo "enetr d values : "

read d

echo "enetr e values : "

read e

echo "enetr f values : "

read f

sub=`expr $a - $b`

add=$(expr "$c" + "$d")

div=$((e / f))

echo "Gave input a-b values : " $sub;

echo "Gave input c+d values : " $add;

echo "Gave input e/f values : " $div;

## Relational Operators:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **Eq** | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a -eq $b ] is not true. |
| **Ne** | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a -ne $b ] is true. |
| **Gt** | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | [ $a -gt $b ] is not true. |
| **Lt** | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | [ $a -lt $b ] is true. |
| **Ge** | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | [ $a -ge $b ] is not true. |
| **Le** | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | [ $a -le $b ] is true. |

# Unix - Shell Relational Operators Example

#!/bin/sh

a=10

b=20

if [ $a -eq $b ]

then

echo "$a -eq $b : a is equal to b"

else

echo "$a -eq $b: a is not equal to b"

fi

if [ $a -ne $b ]

then

echo "$a -ne $b: a is not equal to b"

else

echo "$a -ne $b : a is equal to b"

fi

if [ $a -gt $b ]

then

echo "$a -gt $b: a is greater than b"

else

echo "$a -gt $b: a is not greater than b"

fi

if [ $a -lt $b ]

then

echo "$a -lt $b: a is less than b"

else

echo "$a -lt $b: a is not less than b"

fi

if [ $a -ge $b ]

then

echo "$a -ge $b: a is greater or equal to b"

else

echo "$a -ge $b: a is not greater or equal to b"

fi

if [ $a -le $b ]

then

echo "$a -le $b: a is less or equal to b"

else

echo "$a -le $b: a is not less or equal to b"

fi

This would produce following result −

10 -eq 20: a is not equal to b

10 -ne 20: a is not equal to b

10 -gt 20: a is not greater than b

10 -lt 20: a is less than b

10 -ge 20: a is not greater or equal to b

10 -le 20: a is less or equal to b

## Boolean Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ! | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| -o | This is logical OR. If one of the operands is true then condition would be true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| -a | This is logical AND. If both the operands are true then condition would be true otherwise it would be false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

# Unix - Shell Boolean Operators Example

#!/bin/sh

a=10

b=20

if [ $a != $b ]

then

echo "$a != $b : a is not equal to b"

else

echo "$a != $b: a is equal to b"

fi

if [ $a -lt 100 -a $b -gt 15 ]

then

echo "$a -lt 100 -a $b -gt 15 : returns true"

else

echo "$a -lt 100 -a $b -gt 15 : returns false"

fi

if [ $a -lt 100 -o $b -gt 100 ]

then

echo "$a -lt 100 -o $b -gt 100 : returns true"

else

echo "$a -lt 100 -o $b -gt 100 : returns false"

fi

if [ $a -lt 5 -o $b -gt 100 ]

then

echo "$a -lt 5 -o $b -gt 100 : returns true"

else

echo "$a -lt 5 -o $b -gt 100 : returns false"

fi

This would produce following result −

10 != 20 : a is not equal to b

10 -lt 100 -a 20 -gt 15 : returns true

10 -lt 100 -o 20 -gt 100 : returns true

10 -lt 5 -o 20 -gt 100 : returns false

## String Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the value of two operands are equal or not, if yes then 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 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 non-zero length then it returns true. | [ -n $a ] is not false. |
| Str | Check if str is not the empty string. If it is empty then it returns false. | [ $a ] is not false. |

# Shell String Operators Example

#bin/bash

i="y"

a=0

t=0

while [ $i = "y" ]

do

echo "1.Compare 2 strings :"

echo "2.Concatanet string"

echo "3.Find length of string"

echo "4.Occurance of word"

echo "5.Reverse of string"

echo "6.Exit"

echo "Enter your choice"

read ch

case $ch in

1)echo "Enter first String"

read s1

echo "Enter second string "

read s2

if [ $s1 = $s2 ]

then

echo "Two strings are equal "

else

echo "Two strings are not equal"

fi;;

2)echo "Enter one string "

read s1

echo "Enter second string "

read s2

echo $s1 $s2;;

3)echo "Enter any String"

read s1

t=`echo $s1|wc -c`

t=`expr $t - 1`

echo "Length of "$s1" is "$t;;

4)echo "Enter any String "

read s1

echo "Enter word u want to find occurance of:"

read c1

t=`echo $s1|wc -c`

t=`expr $t - 1`

echo "length "$t

while [ $t -ne 0 ]

do

temp=`echo $s1|cut -c $t`

temp2=`echo $temp2 $temp`

#echo $temp2

if [ $temp2 = $c1 ]

then

a=`expr $a + 1`

t=`expr $t - 1`

else

t=`expr $t - 1`

fi

done

echo "Occurance of "$c1" is "$a;;

5)echo "Enter any string :"

read s1

t=`echo $s1|wc -c`

t=`expr $t - 1`

echo "length "$t

while [ $t -ne 0 ]

do

temp=`echo $s1|cut -c $t`

echo $temp

temp2=`echo $temp2 $temp`

t=`expr $t - 1`

done

echo $temp2;;

6)exit;;

\*)echo "Invalid choice";;

esac

echo "Do u want to continue ?"

read i

if [ $i != "y" ]

then

exit

fi

done

Ex 2 :

#!/bin/bash

a="abc"

b="efg"

if [ $a = $b ]

then

echo "$a = $b : a is equal to b"

else

echo "$a = $b: a is not equal to b"

fi

if [ $a != $b ]

then

echo "$a != $b : a is not equal to b"

else

echo "$a != $b: a is equal to b"

fi

if [ -z $a ]

then

echo "-z $a : string length is zero"

else

echo "-z $a : string length is not zero"

fi

if [ -n $a ]

then

echo "-n $a : string length is not zero"

else

echo "-n $a : string length is zero"

fi

if [ $a ]

then

echo "$a : string is not empty"

else

echo "$a : string is empty"

fi

This would produce following result −

abc = efg: a is not equal to b

abc != efg : a is not equal to b

-z abc : string length is not zero

-n abc : string length is not zero

abc : string is not empty

### Identify String Length inside Bash Shell Script

${#string}

The above format is used to get the length of the given bash variable.

$ cat len.sh

#! /bin/bash

var="Welcome to the nextgen4it"

echo ${#var}

$ ./len.sh

26

### Extract a Substring from a Variable inside Bash Shell Script

Bash provides a way to extract a substring from a string. The following example expains how to parse n characters starting from a particular position.

${string:position}

Extract substring from $string at $position

${string:position:length}

Extract $length of characters substring from $string starting from $position. In the below example, first echo statement returns the substring starting from 15th position. Second echo statement returns the 4 characters starting from 15th position. Length must be the number greater than or equal to zero.

$ cat substr.sh

#! /bin/bash

var="Welcome to the nextgen4it"

echo ${var:15}

echo ${var:15:4}

$ ./substr.sh

nextgen4it

n4it.

### Shortest Substring Match

Following syntax deletes the shortest match of $substring from front of $string

${string#substring}

Following syntax deletes the shortest match of $substring from back of $string

${string%substring}

Following sample shell script explains the above two shortest substring match concepts.

$ cat shortest.sh

#! /bin/bash

filename="bash.string.txt"

echo ${filename#\*.}

echo ${filename%.\*}

$ ./shortest.sh

After deletion of shortest match from front: string.txt

After deletion of shortest match from back: bash.string

In the first echo statement substring ‘\*.’ matches the characters and a dot, and # strips from the front of the string, so it strips the substring “bash.” from the variable called filename. In second echo statement substring ‘.\*’ matches the substring starts with dot, and % strips from back of the string, so it deletes the substring ‘.txt’

### Longest Substring Match

Following syntax deletes the longest match of $substring from front of $string

${string##substring}

Following syntax deletes the longest match of $substring from back of $string

${string%%substring}

Following sample shell script explains the above two longest substring match concepts.

$ cat longest.sh

#! /bin/bash

filename="bash.string.txt"

echo "After deletion of longest match from front:" ${filename##\*.}

echo "After deletion of longest match from back:" ${filename%%.\*}

$ ./longest.sh

After deletion of longest match from front: txt

After deletion of longest match from back: bash

In the above example, ##\*. strips longest match for ‘\*.’ which matches “bash.string.” so after striping this, it prints the remaining txt. And %%.\* strips the longest match for .\* from back which matches “.string.txt”, after striping  it returns “bash”.

### Find and Replace String Values inside Bash Shell Script

#### Replace only first match

${string/pattern/replacement}

It matches the pattern in the variable $string, and replace only the first match of the pattern with the replacement.

$ cat firstmatch.sh

#! /bin/bash

filename="bash.string.txt"

echo "After Replacement:" ${filename/str\*./operations.}

$ ./firstmatch.sh

After Replacement: bash.operations.txt

#### Replace all the matches

${string//pattern/replacement}

It replaces all the matches of pattern with replacement.

$ cat allmatch.sh

#! /bin/bash

filename="Path of the bash is /bin/bash"

echo "After Replacement:" ${filename//bash/sh}

$ ./allmatch.sh

After Replacement: Path of the sh is /bin/sh

.

#### Replace beginning and end

${string/#pattern/replacement}

Following syntax replaces with the replacement string, only when the pattern matches beginning of the $string.

${string/%pattern/replacement}

Following syntax replaces with the replacement string, only when the pattern matches at the end of the given $string.

$ cat posmatch.sh

#! /bin/bash

filename="/root/admin/monitoring/process.sh"

echo "Replaced at the beginning:" ${filename/#\/root/\/tmp}

echo "Replaced at the end": ${filename/%.\*/.ksh}

$ ./posmatch.sh

Replaced at the beginning: /tmp/admin/monitoring/process.sh

Replaced at the end: /root/admin/monitoring/process.ksh

## File Test Operators

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

# Shell File Test Operators Example

#!/bin/bash

file="/home/suresh/Unix/Shell/array.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

This would produce following result −

File has read access

File has write permission

File has execute permission

File is an ordinary file

This is not a directory

File size is zero

File exists

**Conditions :**

## The if...else statements:

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if..else statement −

[**if...fi statement**](http://www.tutorialspoint.com/unix/if-fi-statement.htm)

**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

This will produce following result −

a is not equal to b

[**if...else...fi statement**](http://www.tutorialspoint.com/unix/if-else-statement.htm)

**Example**

If we take above example then it can be written in better way using *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

This will produce following result −

a is not equal to b

[**if...elif...else...fi statement**](http://www.tutorialspoint.com/unix/if-elif-statement.htm)

**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

This will produce following result −

a is less than b

## 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 is detailed here −

* [case...esac statement](http://www.tutorialspoint.com/unix/case-esac-statement.htm)

Unix Shell's case...esac is very similar to switch...case statement we have in other programming languages like C or C++ and PERL etc.

**Example**

#!/bin/sh

FRUIT=$1

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."

;;

"\*") echo "this is expecting parameter please pass apple or banana or kiwi"

;;

esac

This will produce 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}"

if [ -e $FILE ]

then

echo "File name is $FILE"

else

echo "File is $FILE not found"

fi

;;

-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 this 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

$

**EX:**

#!/bin/sh

echo "Enter a number between 1 and 10. "

read NUM

case $NUM in

1) echo "one" ;;

2) echo "two" ;;

3) echo "three" ;;

4) echo "four" ;;

5) echo "five" ;;

6) echo "six" ;;

7) echo "seven" ;;

8) echo "eight" ;;

9) echo "nine" ;;

10) echo "ten" ;;

\*) echo "INVALID NUMBER!" ;;

esac

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers −

[**The while loop**](http://www.tutorialspoint.com/unix/while-loop.htm)

Example

Here is a simple example that uses the while loop to display the numbers zero to nine −

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

[**The for loop**](http://www.tutorialspoint.com/unix/for-loop.htm)

**Example**

Here is a simple example that uses for loop to span through the given list of numbers −

#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9

do

echo $var

done

*#!/bin/ksh*

**for** i **in** **{**1..100**}**

**do**

*# your-unix-command-here*

**echo** $i

**done**

*#!/bin/bash*

**for** **((**i=1;i**<**=100;i++**))**;

**do**

*# your-unix-command-here*

**echo** $i

**done**

[**The until loop**](http://www.tutorialspoint.com/unix/until-loop.htm)

**example**

Here is a simple example that uses the until loop to display the numbers zero to nine −

#!/bin/sh

a=0

until [ ! $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

[**The select loop**](http://www.tutorialspoint.com/unix/select-loop.htm)

**Example**

Here is a simple example to let the user select a drink of choice −

#!/bin/ksh

select DRINK in tea cofee water juice appe all none

do

case $DRINK in

tea|cofee|water|all)

echo "Go to canteen"

;;

juice|appe)

echo "Available at home"

;;

none)

break

;;

\*) echo "ERROR: Invalid selection"

;;

esac

done

The menu presented by the select loop looks like the following −

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

#? juice

Available at home

#? none

$

You can change the prompt displayed by the select loop by altering the variable PS3 as follows −

$PS3="Please make a selection => " ; export PS3

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

Please make a selection => juice

Available at home

Please make a selection => none

$

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

Once you have good programming practice you would start using appropriate loop based on situation. Here while and for loops are available in most of the other programming languages like C, C++ and PERL etc.

## Nesting Loops

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

Here is an example of nesting **while** loop and similar way other loops can be nested based on programming requirement −

## 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 following result. It is important to note how **echo -n** works here. Here **-n** option let echo to avoid printing a new line character.

## Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows −

$ wc -l users

2 users

$

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file *users* −

$ wc -l < users

2

$

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

## Here Document

A *here document* is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is −

command << delimiter

document

delimiter

* The **break** statement
* The **continue** statement

## The infinite Loop

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

## Example

Here is a simple example that uses the while loop to display the numbers zero to nine −

#!/bin/sh

a=10

while [ $a -ge 10 ]

do

echo $a

a=`expr $a + 1`

done

This loop would continue forever because a is alway greater than or equal to 10 and it would never become less than 10. So this true example of infinite loop.

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

until command

do

Statement(s) to be executed until command is true

done

Here Shell *command* is evaluated. If the resulting value is *false*, given*statement(s)* are executed. If *command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

**Example**

Here is a simple example that uses the until loop to display the numbers zero to nine −

#!/bin/sh

a=0

until [ ! $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This will produce following result −

0

1

2

3

4

5

6

7

8

9

The *select* loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

Syntax

select var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

For every selection a set of commands would be executed with-in the loop. This loop was introduced in ksh and has been adapted into bash. It is not available in sh.

**Example**

Here is a simple example to let the user select a drink of choice −

#!/bin/ksh

select DRINK in tea cofee water juice appe all none

do

case $DRINK in

tea|cofee|water|all)

echo "Go to canteen"

;;

juice|appe)

echo "Available at home"

;;

none)

break

;;

\*) echo "ERROR: Invalid selection"

;;

esac

done

The menu presented by the select loop looks like the following −

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

#? juice

Available at home

#? none

$

## The break statement

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

## Syntax

The following **break** statement would be used to come out of a loop −

break

The break command can also be used to exit from a nested loop using this format −

break n

Here **n** specifies the nth enclosing loop to exit from.

## Example

Here is a simple example which shows that loop would terminate as soon as a becomes 5:

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

if [ $a -eq 5 ]

then

break

fi

a=`expr $a + 1`

done

This will produce following result −

0

1

2

3

4

5

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0 −

#!/bin/sh

for var1 in 1 2 3

do

for var2 in 0 5

do

if [ $var1 -eq 2 -a $var2 -eq 0 ]

then

break 2

else

echo "$var1 $var2"

fi

done

done

This will produce following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from inner loop as well.

1 0

1 5

## The continue statement

The **continue** statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

## Syntax

continue

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

continue n

Here n specifies the nth enclosing loop to continue from.

## Example

The following loop makes use of continue statement which returns from the continue statement and start processing next statement −

#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS

do

Q=`expr $NUM % 2`

if [ $Q -eq 0 ]

then

echo "Number is an even number!!"

continue

fi

echo "Found odd number"

done

This will produce following result −

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

**What is Substitution?**

The shell performs substitution when it encounters an expression that contains one or more special characters.

**Example**

Following is the example, while printing value of the variable its substitued by its value. Same time "\n" is substituted by a new line −

#!/bin/sh

a=10

echo -e "Value of a is $a \n"

This would produce following result. Here **-e** option enables interpretation of backslash escapes.

Value of a is 10

Here is the result without -e option:

Value of a is 10\n

Here are following escape sequences which can be used in echo command −

|  |  |
| --- | --- |
| **Escape** | **Description** |
| **\\** | Backslash |
| **\a** | alert (BEL) |
| **\b** | Backspace |
| **\c** | suppress trailing newline |
| **\f** | form feed |
| **\n** | new line |
| **\r** | carriage return |
| **\t** | horizontal tab |
| **\v** | vertical tab |

You can use **-E** option to disable interpretation of backslash escapes (default).

You can use **-n** option to disable insertion of new line.

**Command Substitution**

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

**Syntax**

The command substitution is performed when a command is given as:

`command`

When performing command substitution make sure that you are using the backquote, not the single quote character.

**Example**

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrate command substitution −

#!/bin/sh

DATE=`date`

echo "Date is $DATE"

USERS=`who | wc -l`

echo "Logged in user are $USERS"

UP=`date ; uptime`

echo "Uptime is $UP"

This will produce following result −

Date is Thu Jul 2 03:59:57 MST 2009

Logged in user are 1

Uptime is Thu Jul 2 03:59:57 MST 2009

03:59:57 up 20 days, 14:03, 1 user, load avg: 0.13, 0.07, 0.15

**Variable Substitution**

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions −

|  |  |
| --- | --- |
| **Form** | **Description** |
| **${var}** | Substitue the value of *var*. |
| **${var:-word}** | If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change. |
| **${var:=word}** | If *var* is null or unset, *var* is set to the value of **word**. |
| **${var:?message}** | If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly. |
| **${var:+word}** | If *var* is set, *word* is substituted for var. The value of *var*does not change. |

Example

Following is the example to show various states of the above substitution −

#!/bin/sh

echo ${var:-"Variable is not set"}

echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}

echo "2 - Value of var is ${var}"

unset var

echo ${var:+"This is default value"}

echo "3 - Value of var is $var"

var="Prefix"

echo ${var:+"This is default value"}

echo "4 - Value of var is $var"

echo ${var:?"Print this message"}

echo "5 - Value of var is ${var}"

This would produce following result −

Variable is not set

1 - Value of var is

Variable is not set

2 - Value of var is Variable is not set

3 - Value of var is

This is default value

4 - Value of var is Prefix

Prefix

5 - Value of var is Prefix

**The Metacharacters**

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example **?** matches with a single charater while listing files in a directory and an **\*** would match more than one characters. Here is a list of most of the shell special characters (also called metacharacters) −

\* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

**Example**

Following is the example which show how to print a **\*** or a **?** −

#!/bin/sh

echo Hello; Word

This would produce following result −

Hello

./test.sh: line 2: Word: command not found

shell returned 127

Now let us try using a quoted character −

#!/bin/sh

echo Hello\; Word

This would produce following result −

Hello; Word

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell −

#!/bin/sh

echo "I have \$1200"

This would produce following result −

I have $1200

There are following four forms of quotings −

|  |  |
| --- | --- |
| **Quoting** | **Description** |
| **Single quote** | All special characters between these quotes lose their special meaning. |
| **Double quote** | Most special characters between these quotes lose their special meaning with these exceptions:   * $ * ` * \$ * \' * \" * \\ |
| **Backslash** | Any character immediately following the backslash loses its special meaning. |
| **Back Quote** | Anything in between back quotes would be treated as a command and would be executed. |

**The Single Quotes**

Consider an echo command that contains many special shell characters −

echo <-$1500.\*\*>; (update?) [y|n]

Putting a backslash in front of each special character is tedious and makes the line difficult to read −

echo \<-\$1500.\\*\\*\>\; \(update\?\) \[y\|n\]

There is an easy way to quote a large group of characters. Put a single quote ( ') at the beginning and at the end of the string −

echo '<-$1500.\*\*>; (update?) [y|n]'

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

echo 'It\'s Shell Programming'

**The Double Quotes**

Try to execute the following shell script. This shell script makes use of single quote −

VAR=ZARA

echo '$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]'

This would produce following result −

$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows −

VAR=ZARA

echo "$VAR owes <-\$1500.\*\*>; [ as of (`date +%m/%d`) ]"

Now this would produce following result −

ZARA owes <-$1500.\*\*>; [ as of (07/02) ]

Double quotes take away the special meaning of all characters except the following −

* $ for parameter substitution.
* Backquotes for command substitution.
* \$ to enable literal dollar signs.
* \` to enable literal backquotes.
* \" to enable embedded double quotes.
* \\ to enable embedded backslashes.
* All other \ characters are literal (not special).

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

echo 'It\'s Shell Programming'

**The Back Quotes**

Putting any Shell command in between back quotes would execute the command

Syntax:

Here is the simple syntax to put any Shell **command** in between back quotes −

Example

var=`command`

Example

Following would execute **date** command and produced result would be stored in DATA variable.

DATE=`date`

echo "Current Date: $DATE"

This would produce following result −

Current Date: Thu Jul 2 05:28:45 MST 2009

**Redirection Commands**

Following is the complete list of commands which you can use for redirection −

|  |  |
| --- | --- |
| **Command** | **Description** |
| pgm > file | Output of pgm is redirected to file |
| pgm < file | Program pgm reads its input from file. |
| pgm >> file | Output of pgm is appended to file. |
| n > file | Output from stream with descriptor n redirected to file. |
| n >> file | Output from stream with descriptor n appended to file. |
| n >& m | Merge output from stream n with stream m. |
| n <& m | Merge input from stream n with stream m. |
| << tag | Standard input comes from here through next tag at start of line. |
| | | Takes output from one program, or process, and sends it to another. |

**Functions**

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

## Creating Functions

To declare a function, simply use the following syntax −

function\_name () {

list of commands

}

The name of your function is function\_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

## Example

Following is the simple example of using function −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World"

}

# Invoke your function

Hello

When you would execute above script it would produce following result −

$./test.sh

Hello World

$

## Pass Parameters to a Function

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

}

# Invoke your function

Hello Zara Ali

This would produce following result −

$./test.sh

Hello World Zara Ali

$

## Returning Values from Functions

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the**return** command whose syntax is as follows −

return code

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

## Example

Following function returns a value 1 −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

return 10

}

# Invoke your function

Hello Zara Ali

# Capture value returnd by last command

ret=$?

echo "Return value is $ret"

This would produce following result −

$./test.sh

Hello World Zara Ali

Return value is 10

$

## Nested Functions

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a *recursive function*.

Following simple example demonstrates a nesting of two functions −

#!/bin/sh

# Calling one function from another

number\_one () {

echo "This is the first function speaking..."

number\_two

}

number\_two () {

echo "This is now the second function speaking..."

}

# Calling function one.

number\_one

This would produce following result −

This is the first function speaking...

This is now the second function speaking...

## Function Call from Promp

You can put definitions for commonly used functions inside your *.profile* so that they'll be available whenever you log in and you can use them at command prompt.

Alternatively, you can group the definitions in a file, say *test.sh*, and then execute the file in the current shell by typing −

$. test.sh

This has the effect of causing any functions defined inside test.sh to be read in and defined to the current shell as follows −

$ number\_one

This is the first function speaking...

This is now the second function speaking...

$

To remove the definition of a function from the shell, you use the unset command with the .f option. This is the same command you use to remove the definition of a variable to the shell.

$unset .f function\_name

**Find biggest number**

#!/bin/sh

#

# Written Suresh.

#

#Script to find out bigest number

#

# Algo:

# 1) START: Take three nos as n1,n2,n3.

# 2) Is n1 is greater than n2 and n3, if yes

# print n1 is bigest no goto step 5, otherwise goto next step

# 3) Is n2 is greater than n1 and n3, if yes

# print n2 is bigest no goto step 5, otherwise goto next step

# 4) Is n3 is greater than n1 and n2, if yes

# print n3 is bigest no goto step 5, otherwise goto next step

# 5) END

#

#

if [ $# -ne 3 ]

then

echo "$0: number1 number2 number3 are not given" >&2

exit 1

fi

n1=$1

n2=$2

n3=$3

if [ $n1 -gt $n2 ] && [ $n1 -gt $n3 ]

then

echo "$n1 is Bigest number"

elif [ $n2 -gt $n1 ] && [ $n2 -gt $n3 ]

then

echo "$n2 is Bigest number"

elif [ $n3 -gt $n1 ] && [ $n3 -gt $n2 ]

then

echo "$n3 is Bigest number"

elif [ $1 -eq $2 ] && [ $1 -eq $3 ] && [ $2 -eq $3 ]

then

echo "All the three numbers are equal"

else

echo "I can not figure out which number is biger"

fi

#!/bin/sh

#

# Written by Suresh

#

# Algo:

# 1) START: set value of i to 5 (since we want to start from 5, if you

# want to start from other value put that value)

# 2) Start While Loop

# 3) Chechk, Is value of i is zero, If yes goto step 5 else

# continue with next step

# 4) print i, decement i by 1 (i.e. i=i-1 to goto zero) and

# goto step 3

# 5) END

#

i=5

while test $i != 0

do

echo "$i

"

i=`expr $i - 1`

done

#!/bin/sh

#

# Script to reverse given no

#

# Algo:

# 1) Input number n

# 2) Set rev=0, sd=0

# 3) Find single digit in sd as n % 10 it will give (left most digit)

# 4) Construct revrse no as rev \* 10 + sd

# 5) Decrment n by 1

# 6) Is n is greater than zero, if yes goto step 3, otherwise next step

# 7) Print rev

#

if [ $# -ne 1 ]

then

echo "Usage: $0 number"

echo " I will find reverse of given number"

echo " For eg. $0 123, I will print 321"

exit 1

fi

n=$1

rev=0

sd=0

while [ $n -gt 0 ]

do

sd=`expr $n % 10`

rev=`expr $rev \\* 10 + $sd`

n=`expr $n / 10`

done

echo "Reverse number is $rev"

#!/bin/sh

#

#

# Algo:

# 1) Input number n

# 2) Set sum=0, sd=0

# 3) Find single digit in sd as n % 10 it will give (left most digit)

# 4) Construct sum no as sum=sum+sd

# 5) Decrment n by 1

# 6) Is n is greater than zero, if yes goto step 3, otherwise next step

# 7) Print sum

#

if [ $# -ne 1 ]

then

echo "Usage: $0 number"

echo " I will find sum of all digit for given number"

echo " For eg. $0 123, I will print 6 as sum of all digit (1+2+3)"

exit 1

fi

n=$1

sum=0

sd=0

while [ $n -gt 0 ]

do

sd=`expr $n % 10`

sum=`expr $sum + $sd`

n=`expr $n / 10`

done

echo "Sum of digit for numner is $sum"

**Script to check whether "/\*" is included, in $1 or not**

#!/bin/sh

#

# Script to check whether "/\*" is included, in $1 or not

#

cat "$1" > /tmp/file.$$ 2>/tmp/file0.$$

grep "\*" /tmp/file.$$ >/tmp/file0.$$

if [ $? -eq 1 ]

then

echo "Required i.e. $1/\*"

else

echo "Symbol is Not required"

fi

rm -f /tmp/file.$$

rm -f /tmp/file0.$$

**Shell script to print contains of file from given line no to next**

#!/bin/sh

#

#

# Shell script to print contains of file from given line no to next

# given numberlines

#

#

# Print error / diagnostic for user if no arg's given

#

if [ $# -eq 0 ]

then

echo "$0:Error command arguments missing!"

echo "Usage: $0 start\_line uptoline filename"

echo "Where start\_line is line number from which you would like to print file"

echo "uptoline is line number upto which would like to print"

echo "For eg. $0 5 5 myfile"

echo "Here from myfile total 5 lines printed starting from line no. 5 to"

echo "line no 10."

exit 1

fi

#

# Look for sufficent arg's

#

if [ $# -eq 3 ]; then

if [ -e $3 ]; then

tail +$1 $3 | head -n$2

else

echo "$0: Error opening file $3"

exit 2

fi

else

echo "Missing arguments!"

fi

**echo command with escape sequance to give differnt effects**

#!/bin/bash

#

# echo command with escape sequance to give differnt effects

#

# Syntax: echo -e "escape-code your message, var1, var2 etc"

# For eg. echo -e "\033[1m Hello World"

# | |

# | |

# Escape code Message

#

clear

echo -e "\033[1m Hello World"

# bold effect

echo -e "\033[5m Blink"

# blink effect

echo -e "\033[0m Hello World"

# back to noraml

echo -e "\033[31m Hello World"

# Red color

echo -e "\033[32m Hello World"

# Green color

echo -e "\033[33m Hello World"

# See remaing on screen

echo -e "\033[34m Hello World"

echo -e "\033[35m Hello World"

echo -e "\033[36m Hello World"

echo -e -n "\033[0m "

# back to noraml

echo -e "\033[41m Hello World"

echo -e "\033[42m Hello World"

echo -e "\033[43m Hello World"

echo -e "\033[44m Hello World"

echo -e "\033[45m Hello World"

echo -e "\033[46m Hello World"

echo -e "\033[0m Hello World"

# back to normal

**Declare the array of 5 subscripts to hold 5 numbers**

#!/bin/sh

#

#

# Declare the array of 5 subscripts to hold 5 numbers

#

declare nos[5]=(4 -1 2 66 10)

#

# Prints the number befor sorting

#

echo "Original Numbers in array:"

for (( i = 0; i <= 4; i++ ))

do

echo ${nos[$i]}

done

#

# Now do the Sorting of numbers

#

for (( i = 0; i <= 4 ; i++ ))

do

for (( j = $i; j <= 4; j++ ))

do

if [ ${nos[$i]} -gt ${nos[$j]} ]; then

t=${nos[$i]}

nos[$i]=${nos[$j]}

nos[$j]=$t

fi

done

done

#

# Print the sorted number

#

echo -e "\nSorted Numbers in Ascending Order:"

for (( i=0; i <= 4; i++ ))

do

echo ${nos[$i]}

done