grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is −

$grep pattern file(s)

The name **"grep"** comes from the ed (a Unix line editor) command **g/re/p**which means “globally search for a regular expression and print all lines containing it”.

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work −

$ls -l | grep "Aug"

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02

-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07

-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros

$

There are various options which you can use along with the **grep** command −

|  |  |
| --- | --- |
| **S.No.** | **Option & Description** |
| 1 | **-v**  Prints all lines that do not match pattern. |
| 2 | **-n**  Prints the matched line and its line number. |
| 3 | **-l**  Prints only the names of files with matching lines (letter "l") |
| 4 | **-c**  Prints only the count of matching lines. |
| 5 | **-i**  Matches either upper or lowercase. |

Let us now use a regular expression that tells grep to find lines with **"carol"**, followed by zero or other characters abbreviated in a regular expression as ".\*"), then followed by "Aug".−

Here, we are using the ***-i*** option to have case insensitive search −

$ls -l | grep -i "carol.\*aug"

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros

$

The sort Command

The **sort** command arranges lines of text alphabetically or numerically. The following example sorts the lines in the food file −

$sort food

Afghani Cuisine

Bangkok Wok

Big Apple Deli

Isle of Java

Mandalay

Sushi and Sashimi

Sweet Tooth

Tio Pepe's Peppers

$

The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting −

|  |  |
| --- | --- |
| **S.No.** | **Description** |
| 1 | **-n**  Sorts numerically (example: 10 will sort after 2), ignores blanks and tabs. |
| 2 | **-r**  Reverses the order of sort. |
| 3 | **-f**  Sorts upper and lowercase together. |
| 4 | **+x**  Ignores first **x** fields when sorting. |

More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by the order of size.

The following pipe consists of the commands **ls**, **grep**, and **sort** −

$ls -l | grep "Aug" | sort +4n

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros

-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro

-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02

$

This pipe sorts all files in your directory modified in August by the order of size, and prints them on the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

The pg and more Commands

A long output can normally be zipped by you on the screen, but if you run text through more or use the **pg** command as a filter; the display stops once the screen is full of text.

Let's assume that you have a long directory listing. To make it easier to read the sorted listing, pipe the output through **more** as follows −

$ls -l | grep "Aug" | sort +4n | more

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros

-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro

-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07

-rw-rw-r-- 1 john doc 14827 Aug 9 12:40 ch03

.

.

.

-rw-rw-rw- 1 john doc 16867 Aug 6 15:56 ch05

--More--(74%)

# Using Shell 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 will 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.

## Defining Variables

Variables are defined as follows −

variable\_name=variable\_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

VAR1="Zara Ali"

VAR2=100

## Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

#!/bin/sh

NAME="Zara Ali"

echo $NAME

The above script will produce the following value −

Zara Ali

## Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

#!/bin/sh

NAME="Zara Ali"

readonly NAME

NAME="Qadiri"

The above script will generate the following result −

/bin/sh: NAME: This variable is read only.

## Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

## 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 the command prompt.
* **Environment Variables** − An environment variable 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.

# Special Variables

For example, the **$** character represents the process ID number, or PID, of the current shell −

$echo $$

The above command writes the PID of the current shell −

29949

The following table shows a number of special variables that you can use in your shell scripts −

|  |  |
| --- | --- |
| **S.No.** | **Variable & Description** |
| 1 | **$0**  The filename of the current script. |
| 2 | **$n**  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 $2, and so on). |
| 3 | **$#**  The number of arguments supplied to a script. |
| 4 | **$\***  All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**  All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**  The exit status of the last command executed. |
| 7 | **$$**  The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!**  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 the 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

## Special Parameters $\* and $@

There are special parameters that allow accessing all the command-line arguments at once. **$\*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $\* or $@ special parameters −

#!/bin/sh

for TOKEN in $\*

do

echo $TOKEN

done

Here is a sample run for the above script −

$./test.sh Zara Ali 10 Years Old

Zara

Ali

10

Years

Old

**Note** − Here **do...done** is a kind of loop that will be covered in a subsequent tutorial.

## 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

$

# Using Shell Arrays

## 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"

Name= (‘’,’’,’’,’’)

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

# Shell Basic Operators

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators −

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

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

The following example shows how to add two numbers −

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

The above script will generate the following result −

Total value : 4

The following points need to be considered while adding −

* There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
* The complete expression should be enclosed between **‘ ‘**, called the backtick.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[ $a == $b ]**is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Checks if the value of two operands are equal or not; if yes, then the 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 the 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 the 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 the 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 the 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 the condition becomes true. | [ $a -le $b ] is true. |

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[ $a <= $b ]** is correct whereas, **[$a <= $b]** is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

|  |  |  |
| --- | --- | --- |
| **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 the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Show Examples

|  |  |  |
| --- | --- | --- |
| **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. |

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

|  |  |  |
| --- | --- | --- |
| **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 t |

## Arithmetic and Logical Operators

The following table lists out a few Arithmetic and Logical Operators −

|  |  |
| --- | --- |
| **S.No.** | **Operator & Description** |
| 1 | **( )**  Change precedence |
| 2 | **~**  1's complement |
| 3 | **!**  Logical negation |
| 4 | **\***  Multiply |
| 5 | **/**  Divide |
| 6 | **%**  Modulo |
| 7 | **+**  Add |
| 8 | **-**  Subtract |
| 9 | **<<**  Left shift |
| 10 | **>>**  Right shift |
| 11 | **==**  String comparison for equality |
| 12 | **!=**  String comparison for non equality |
| 13 | **=~**  Pattern matching |
| 14 | **&**  Bitwise "and" |
| 15 | **^**  Bitwise "exclusive or" |
| 16 | **|**  Bitwise "inclusive or" |
| 17 | **&&**  Logical "and" |
| 18 | **||**  Logical "or" |
| 19 | **++**  Increment |
| 20 | **--**  Decrement |
| 21 | **=**  Assignment |
| 22 | **\*=**  Multiply left side by right side and update left side |
| 23 | **/=**  Divide left side by right side and update left side |
| 24 | **+=**  Add left side to right side and update left side |
| 25 | **-=**  Subtract left side from right side and update left side |
| 26 | **^=**  "Exclusive or" left side to right side and update left side |
| 27 | **%=**  Divide left by right side and update left side with remainder |

## File Test Operators

The following operators test various properties associated with a Unix file.

|  |  |
| --- | --- |
| **S.No.** | **Operator & Description** |
| 1 | **-r file**  Checks if file is readable; if yes, then the condition becomes true. |
| 2 | **-w file**  Checks if file is writable; if yes, then the condition becomes true. |
| 3 | **-x file**  Checks if file is executable; if yes, then the condition becomes true. |
| 4 | **-f file**  Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. |
| 5 | **-z file**  Checks if file has size greater than 0; if yes, then the condition becomes true. |
| 6 | **-d file**  Checks if file is a directory; if yes, then the condition becomes true. |
| 7 | **-e file**  Checks if file exists; is true even if file is a directory but exists. |
| 8 | **-o file**  Checks if user owns the file; returns true if the user is the owner of the file. |

# Shell Decision Making

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

* The **if...else** statement
* The **case...esac** statement

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
* if...else...fi statement
* if...elif...else...fi statement

Most of the if statements check relations using relational operators discussed in the previous chapter.

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 like **C** or **C++** and **PERL**, etc.

# Shell Loop Types

In this chapter, we will discuss shell loops in Unix. A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

* The while loop
* The for loop
* The until loop
* The select loop

You will use different loops based on the situation. For example, the **while**loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the 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 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

* 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 if the required condition is not met. A loop that executes forever without terminating executes for 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

until [ $a -lt 10 ]

do

echo $a

a=expr $a + 1`

done

This loop continues forever because **a** is always **greater than** or **equal to 10**and it is never less than 10.

## 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 is 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 the exit from.

### Example

Here is a simple example which shows that loop terminates 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

Upon execution, you will receive the 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

Upon execution, you will receive the 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 the 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 the **continue** statement which returns from the continue statement and starts processing the 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

Upon execution, you will receive the 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

# Shell Substitution

## What is Substitution?

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

### Example

Here, the printing value of the variable is substituted by its value. Same time, **"\n"** is substituted by a new line −

#!/bin/sh

a=10

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

You will receive the following result. Here the **-e** option enables the interpretation of backslash escapes.

Value of a is 10

Following is the result without **-e** option −

Value of a is 10\n

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

|  |  |
| --- | --- |
| **S.No.** | **Escape & Description** |
| 1 | **\\**  backslash |
| 2 | **\a**  alert (BEL) |
| 3 | **\b**  backspace |
| 4 | **\c**  suppress trailing newline |
| 5 | **\f**  form feed |
| 6 | **\n**  new line |
| 7 | **\r**  carriage return |
| 8 | **\t**  horizontal tab |
| 9 | **\v**  vertical tab |

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

You can use the **-n** option to disable the insertion of a 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 the command substitution make sure that you use 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 demonstrates the 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"

Upon execution, you will receive the 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 −

|  |  |
| --- | --- |
| **S.No.** | **Form & Description** |
| 1 | **${var}**  Substitute the value of *var*. |
| 2 | **${var:-word}**  If *var* is null or unset, *word* is substituted for **var**. The value of *var*does not change. |
| 3 | **${var:=word}**  If *var* is null or unset, *var* is set to the value of **word**. |
| 4 | **${var:?message}**  If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly. |
| 5 | **${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}"

Upon execution, you will receive the 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

# Shell Functions

In this chapter, we will discuss in detail about the shell 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 tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**. This 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, followed by a list of commands enclosed within braces.

### Example

Following example shows the use of function −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World"

}

# Invoke your function

Hello

Upon execution, you will receive the following output −

$./test.sh

Hello World

## Pass Parameters to a Function

You can define a function that will accept parameters while calling the 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

Upon execution, you will receive the 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"

Upon execution, you will receive the 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 and also other functions. A function that calls itself is known as a ***recursive function***.

Following example demonstrates 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

Upon execution, you will receive the following result −

This is the first function speaking...

This is now the second function speaking...

## Function Call from Prompt

You can put definitions for commonly used functions inside your ***.profile***. These definitions will be available whenever you log in and you can use them at the 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 functions defined inside ***test.sh*** to be read 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, use the unset command with the **.f** option. This command is also used to remove the definition of a variable to the shell.

$unset .f function\_name