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 −

\_AL

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="Vimlesh"

The above example defines the variable NAME and assigns the value "Vimlesh" 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="Vimlesh"

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="Vimlesh"

echo $NAME

The above script will produce the following value −

Vimlesh

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="Vimlesh"

readonly NAME

NAME="aa"

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="Vilesh"

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

|  |  |
| --- | --- |
| **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 Vimlesh

File Name : ./test.sh

First Parameter : aa

Second Parameter : vvvv

Quoted Values: Vimlesh

Quoted Values: Vimlesh

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 aaa vvvv 10 Years Old

aaa

vvvv

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 Vimlesh

File Name : ./test.sh

First Parameter : aaa

Second Parameter : vvvv

Quoted Values: Vimlesh

Quoted Values: Vimlesh

Total Number of Parameters : 2

$echo $?

0

$

About expr

**expr** evaluates arguments as an expression.

expr syntax

expr *EXPRESSION*

expr *OPTION*

Options

|  |  |
| --- | --- |
| **--help** | Display a help message and exit. |
| **--version** | Display version information and exit. |

Expressions

**expr** prints the value of *EXPRESSION* to standard output. A blank line below separates increasing precedence groups.

*EXPRESSION* may be:

|  |  |
| --- | --- |
| *ARG1* **|** *ARG2* | *ARG1* if it is neither null nor 0, otherwise *ARG2*. |
| *ARG1* **&** *ARG2* | *ARG1* if neither argument is null or 0, otherwise 0. |
| *ARG1* **<** *ARG2* | *ARG1* is less than *ARG2*. |
| *ARG1* **<=** *ARG2* | *ARG1* is less than or equal to *ARG2*. |
| *ARG1* **=** *ARG2* | *ARG1* is equal to *ARG2*. |
| *ARG1* **!=** *ARG2* | *ARG1* is unequal to *ARG2*. |
| *ARG1* **>=** *ARG2* | *ARG1* is greater than or equal to *ARG2*. |
| *ARG1* **>** *ARG2* | *ARG1* is greater than *ARG2*. |
| *ARG1* **+** *ARG2* | arithmetic sum of *ARG1* and *ARG2*. |
| *ARG1* **-** *ARG2* | arithmetic difference of *ARG1* and *ARG2*. |
| *ARG1* **\*** *ARG2* | arithmetic product of *ARG1* and *ARG2*. |
| *ARG1* **/** *ARG2* | arithmetic quotient of *ARG1* divided by *ARG2*. |
| *ARG1* **%** *ARG2* | arithmetic remainder of *ARG1* divided by *ARG2*. |
| *STRING* **:** *REGEXP* | anchored pattern match of regular expression *REGEXP* in *STRING*. |
| **match** *STRING* *REGEXP* | same as *STRING* **:** *REGEXP*. |
| **substr** *STRING* *POSLENGTH* | substring of *STRING*, *POS* counted from 1. |
| **index** *STRING* *CHARS* | index in *STRING* where any *CHARS* is found, or **0**. |
| **length** *STRING* | length of *STRING*. |
| **+** *TOKEN* | interpret *TOKEN* as a string, even if it is a keyword like '**match**' or an operator like '**/**'. |
| **(** *EXPRESSION* **)** | value of *EXPRESSION*. |

## EXAMPLES

**Example-1:**

To perform addition of two numbers:

$ expr 3 + 5

output:

8

**Example-2:**

To  perform substraction of two numbers:

$ expr 5 - 3

output:

2

**Example-3:**

To perform multiplication of two numbers ( note: The multiplication operator (\*) must be escaped when used in an arithmetic expression with **expr** )

$ expr 5 \\* 3

output:

15

**Example-4:**

To perform division operation:

$ expr 10 / 2

output:

5

**Example-5:**

To increament variable :

$ y=10

$ y=`expr $y + 1`

$ echo $y

output:  
11

**Example-6:**

To find length of string

a=hello

b=`expr length $a`

echo $b

output:

5

**Example-7:**

To find the index/position of character in a string

a=hello

b=`expr index $a l`

echo $b

output:

3 (  as letter l is at position 3.)

**Example-8:**

To find substring of string:

a=hello

b=`expr substr $a 2 3` ( where 2 is position and 3 is length, command is to get substring from position 2 of length 3 characters)

echo $b

output:

ell

**Example-9:**

The following is an example involving boolean expressions ( |-  or operator ):

$ expr length "abcdef" "<" 5 "|" 15 - 4 ">" 8  
  
output:  
1

**Example-10:**

The following is an example involving boolean expressions ( & - and operator ):

$ expr length "abcdef" "<" 5 "&" 15 - 4 ">" 8

# Test Conditions Within a Bash Script

### A Basic Example

test 1 -eq 2 && echo "yes" || echo "no"

The above command can be broken down as follows:

* test - this means you are about to perform a comparison
* 1 - the first element you are going to compare
* -eq (how are you comparing, in this case, you are testing whether one number equals another)
* 2 - the element you are comparing the first element again
* && - run the following statement if the result is true
* echo "yes" - the command to run if the comparison returns true
* || - run the following statement if the result is false
* echo "no" - the command to run if the comparison returns false

In essence, the command is comparing 1 to 2 and they match the echo "yes" statement is executed which displays "yes" and if they do not match the echo "no" statement is executed which displays "no".

### Comparing Numbers

If you are comparing elements that parse as numbers you can use the following comparison operators:

* -eq - does value 1 equal value 2
* -ge - is value 1 greater or equal to value 2
* -gt - is value 1 greater than value 2
* -le - is value 1 less than or equal to value 2
* -lt - is value 1 less than value 2
* -ne - does value 1 not equal value 2

**Examples:**

test 1 -eq 2 && echo "yes" || echo "no"

(displays "no" to the screen because 1 does not equal 2)

test 1 -ge 2 && echo "yes" || echo "no"

(displays "no" to the screen because 1 is not greater or equal to 2)

test 1 -gt 2 && echo "yes" || echo "no"

(displays "no" to the screen because 1 is not greater than 2)

test 1 -le 2 && echo "yes" || echo "no"

(displays "yes" to the screen because 1 is less than or equal to 2)

test 1 -lt 2 && echo "yes" || echo "no"

(displays "yes" to the screen because 1 is less than or equal to 2)

test 1 -ne 2 && echo "yes" || echo "no"

(displays "yes" to the screen because 1 does not equal 2)

### Comparing Text

If you are comparing elements that parse as strings you can use the following comparison operators:

* = - does string 1 match string 2
* != - is string 1 different to string 2
* -n - is the string length greater than 0
* -z - is the string length 0

**Examples:**

test "string1" = "string2" && echo "yes" || echo "no"

(displays "no" to the screen because "string1" does not equal "string2")

test "string1" != "string2" && echo "yes" || echo "no"

(displays "yes" to the screen because "string1" does not equal "string2")

test -n "string1" && echo "yes" || echo "no"

(displays "yes" to the screen because "string1" has a string length greater than zero)

test -z "string1" && echo "yes" || echo "no"

(displays "no" to the screen because "string1" has a string length greater than zero)

### Comparing Files

If you are comparing files you can use the following comparison operators:

* -ef - Do the files have the same device and inode numbers (are they the same file)
* -nt - Is the first file newer than the second file
* -ot - Is the first file older than the second file
* -b - The file exists and is [block special](https://en.wikipedia.org/wiki/Device_file)
* -c - The file exists and is character special
* -d - The file exists and is a directory
* -e - The file exists
* -f - The file exists and is a regular file
* -g - The file exists and has the specified group number
* -G - The file exists and owner by the user's group
* -h - The file exists and is [a symbolic link](https://www.lifewire.com/create-symbolic-links-ln-command-4059723)
* -k - The file exists and has its [sticky bit](https://en.wikipedia.org/wiki/Sticky_bit) set
* -L - The same as -h
* -O - The file exists you are the owner
* -p - The file exists and is a named pipe
* -r - The file exists and is readable
* -s - The file exists and has a size greater than zero
* -S - The file exists and is a [socket](https://en.wikipedia.org/wiki/Unix_domain_socket)
* -t - The file descriptor is opened on a terminal
* -u - The file exists and the set-user-id bit is set
* -w - The file exists and is writable
* -x - The file exists and is executable

**Examples:**

test /path/to/file1 -n /path/to/file2 && echo "yes"

(If file1 is newer than file2 then the word "yes" will be displayed)

test  -e /path/to/file1  && echo "yes"

(if file1 exists the word "yes" will be displayed)

test  -O /path/to/file1  && echo "yes"

(if you own file1 then the word "yes" is displayed")

**Terminology**

* Block special - The file is a block device which means that data is read in blocks of bytes. These are generally device files such as hard drives.
* Character special - The file is acted upon immediately when you write to it and is commonly a device such as a serial port

### Comparing Multiple Conditions

Thus far everything has been comparing one thing against another but what if you want to compare two conditions.

For example, if an animal has 4 legs and goes "moo" it is probably a cow. Simply checking for 4 legs doesn't guarantee that you have a cow but checking the sound it makes surely does.​

To test both conditions at once use the following statement:

test 4 -eq 4 -a "moo" = "moo" && echo "it is a cow" || echo "it is not a cow"

The key part here is the -a which stands for and.

There is a better and more commonly used way of performing the same test and that is as follows:

test 4 -eq 4 && test "moo" = "moo" && echo "it is a cow" || echo "it is not a cow"

Another test you might want to make is comparing two statements and if either is true output a string. For example, if you want to check that a file named "file1.txt" exists or a file called "file1.doc" exists you can use the following command

test -e file1.txt -o -e file1.doc && echo "file1 exists" || echo "file1 does not exist"

The key part here is the -o which stands for or.

There is a better and more commonly used way of performing the same test and that is as follows:

test -e file1.txt || test -e file1.doc && echo "file1 exists" || echo "file1 does not exist"

Eliminating The Test Keyword

You don't actually need to use the word test to perform the comparison. All you have to do is enclose the statement in square brackets as follows:

[ -e file1.txt ] && echo "file1 exists" || echo "file1 does not exist"

The [ and ] basically means the same as test.

Now you know this you can improve on comparing multiple conditions as follows:

[ 4 -eq 4 ] && [ "moo" = "moo" ] && echo "it is a cow" || echo "it is not a cow"

[ -e file1.txt ] || [ -e file1.doc ] && echo "file1 exists" || echo "file1 does not exist"