<https://www.javatpoint.com/bash-if-statement>

<https://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>

**Comments:**

* Using # - single line
* We can write multi-line comments in bash scripting by enclosing the comments between <<COMMENT and COMMENT.

**Variables:**

1. System variables- they are like environment variables like HOME, PWD etc.- To know the list of these variables in your system, type the commands **set, env**, and **printenv** on the command line terminal
2. User defined variables-
   1. Declaration:

Var\_Name = value --- there should not be any space before and after = sign, else error as unknown command

* 1. Accessing:

$Var\_name

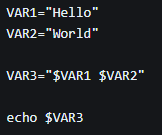
**Quotes:**

* While working with simple texts and string, there will be no differences either we use a single quote or double quote.
* shell variable expansion will only work with double-quotes. If you define any variable in single quotes, then it will not be considered as a variable.
* Summary- for variable we use double quotes when calling it
* There is no need of using any quotes, either single or double, to define a variable with single character value such as **var1=variable**.

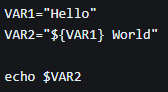
Bash variables are **untyped**, which means just type the variable name by assigning its value, and it will automatically consider that data type.

**Concatenate:**

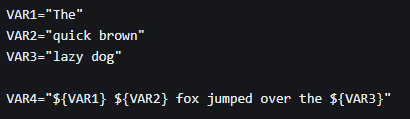
* way to do this in bash is by writing the variables one after the other



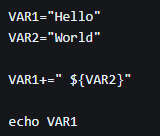
* join two strings where one is a literal and another is a variable, you can do so by following a similar pattern



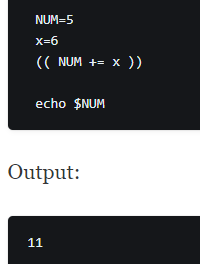
This method also works for joining multiple variables, This method is also useful if you want to join a string variable with a number.



* Using += operator-  The operator is generally used to append numbers and strings, and other variables to an existing variable



If instead, you wanted to treat this as a numeric expression, you would have to wrap it in $(( )) arithmetic expansion.



**Arguments:**

following default-parameters or the special variables.

* **$0** specifies the name of the script to be invoked.
* **$1-$9** stores the names of the first 9 arguments or can be used as the arguments' positions.
* **$#** specifies the total number (count) of arguments passed to the script.
* **$\*** stores all the command line arguments by joining them together.
* **$@** stores the list of arguments as an array.
* **$?** specifies the process ID of the current script.
* **$$** specifies the exit status of the last command or the most recent execution process.
* **$!** shows ID of the last background job.

1. Passing argument value Using position number:

It is the first way of accessing the arguments by using the default parameters ($1...$9).

#! /bin/bash

Echo “you entered $1 and $2”

$ ./script.sh 10 20

1. Passing argument value as array:

variable\_name=("$@")

Where **$@** is the default argument which is used to store the arguments (we pass) as an array.

To access:

${variable\_name[i]}

#! /bin/bash

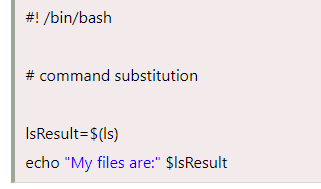
Var=(“$@”)

Echo “you entered ${var[0]} and ${var[1]}

$ ./script.sh 10 200

1. Command Substitution

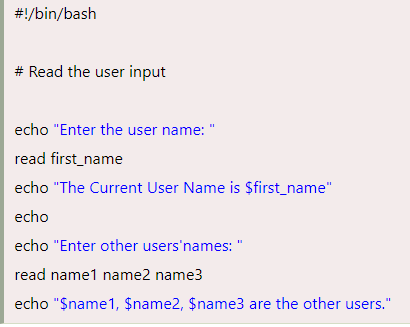
 It takes the output of the Bash command, stores in a variable (generally), and display back with echo.



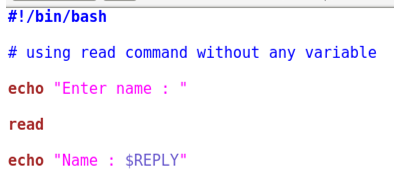
**User Input:**

To read the Bash user input, we use the built-in Bash command called **read**. It takes input from the user and assigns it to the variable. It reads only a single line from the Bash shell

read <variable\_name>



If we don't pass any variable with the read command, then we can pass a built-in variable called **REPLY** (should be prefixed with the $ sign) while displaying the input.

****

we enter the input on the same PROMPT by using the **-p command line option**

read -p PROMPT <variable\_name>

read -p "username:" user\_var

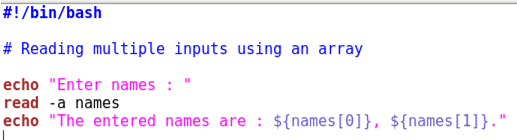
keep the input on silent mode, such that whatever be a user input on the command line will be hidden to others. So, we pass a username and hide the password (silent mode) by using the command line options (-s, -p)

read -sp "password : " pass\_var

Enter multiple inputs using an array.

read -a <variable\_name>

Where **-a** helps script to read an array, and variable\_name refers to an array

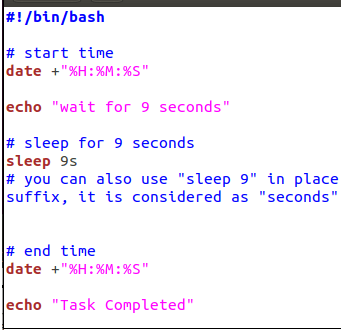


**Sleep**

Sleep is a command-line utility which allows us to suspend the calling process for a specified time. In other words, Bash sleep command is used to insert a delay or pause the execution for a specified period of time.

sleep number[suffix]

* s - seconds
* m - minutes
* h - hours
* d - days
* When there is no suffix, then the number is considered to be in seconds (by default).



**Arithmetic Operators:**

|  |  |
| --- | --- |
| + , - , \*, |  |
| / | Returns quotient |
| \*\* | Exponential |
| % | Returns remainder |
| += , -= , \*=, /=, %= |  |

Performing arithmetic operations:

1. Using (( expression ))

Double parentheses is the easiest mechanism to perform basic arithmetic operations in the Bash shell. We can use this method by using double brackets with or without a leading $.

((expression))

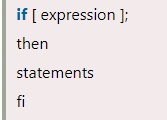
2.Using Let build in command

let <arithmetic expression>

let "z = $(( x + y ))"

**Conditional Statements**

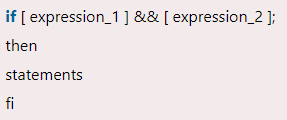
1. If statements:



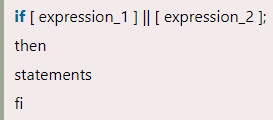
The statement between **then** and **fi** (If backwards) will be executed only if the expression (between the square brackets) is true.

#### Note: Observe the spaces used in the first line and a semicolon at the end of the first line; both are mandatory to use. If conditional statement ends with fi.

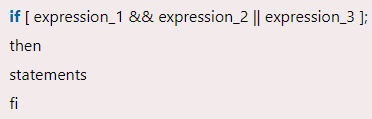
For using multiple conditions with AND operator:



For using multiple conditions with OR operator:



For compound expressions with AND & OR operators,



#! /bin/bash

Read -p “enter a number: ” <**space**> variable

If <**space**> [<**space**>condition<**space**>];

Then

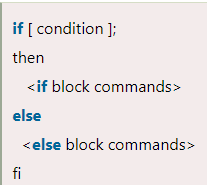
Echo “….”

Fi

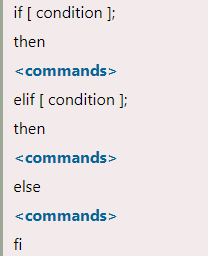
If options:

|  |  |
| --- | --- |
| ! EXPRESSION | To check if EXPRESSION is false. |
| -n STRING | To check if the length of STRING is greater than zero. |
| -z STRING | To check if the length of STRING is zero (i.e., it is empty) |
| STRING1 == STRING2 | To check if STRING1 is equal to STRING2. |
| STRING1 != STRING2 | To check if STRING1 is not equal to STRING2. |
| INTEGER1 -eq INTEGER2 | To check if INTEGER1 is numerically equal to INTEGER2. |
| INTEGER1 -gt INTEGER2 | To check if INTEGER1 is numerically greater than INTEGER2. |
| INTEGER1 -lt INTEGER2 | To check if INTEGER1 is numerically less than INTEGER2. |
| -d FILE | To check if FILE exists and it is a directory. |
| -e FILE | To check if FILE exists. |
| -r FILE | To check if FILE exists and the read permission is granted. |
| -s FILE | To check if FILE exists and its size is greater than zero (which means that it is not empty). |
| -w FILE | To check if FILE exists and the write permission is granted. |
| x FILE | To check if FILE exists and the execute permission is granted. |

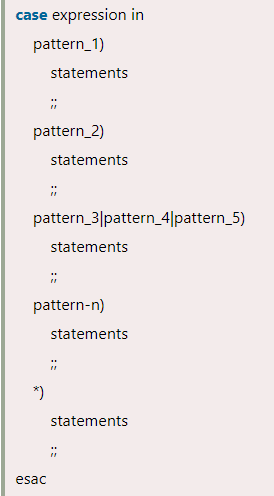
2.If – Else statement



3.Else-if statement

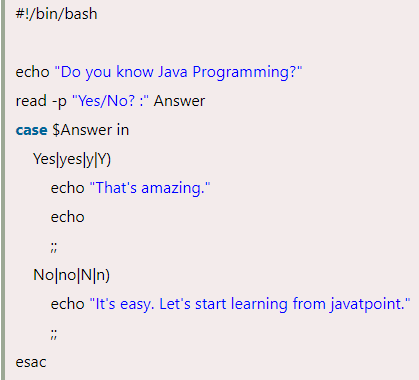


1. Case statement



There are some key points of bash case statements:

* Each case statement in bash starts with the 'case' keyword, followed by the case expression and 'in' keyword. The case statement is closed by 'esac' keyword.
* We can apply multiple patterns separated by | operator. The ) operator indicates the termination of a pattern list.
* A pattern containing the statements is referred to as a clause, and it must be terminated by double semicolon (;;).
* An asterisk symbol (\*) is used as a final pattern to define the default case. It is used as a default case when used as the last case.



One dimensional array:

Var=(…..,…,…)

${var[@]} === will print the array

${var[index]} == will print specific index element

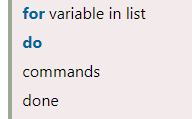
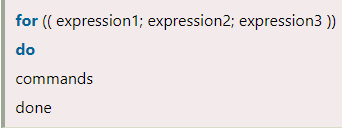
${!var[@]} == prints indexes of array

${#var[@]} == prints length of array

**Loops:**

1.For Loop:

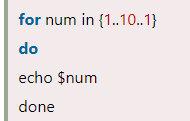
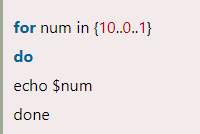
We can apply 'for loop' on bash script in two ways. One way is 'for-in' and another way is the c-style syntax.

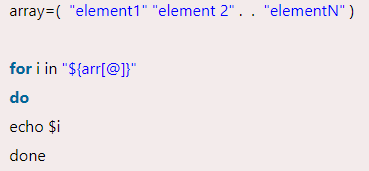
There are some key points of 'for loop' statement:

* Each block of 'for loop' in bash starts with 'do' keyword followed by the commands inside the block. The 'for loop' statement is closed by 'done' keyword.
* The number of time for which a 'for loop' will iterate depends on the declared list variables.
* The loop will select one item from the list and assign the value on a variable which will be used within the loop.
* After the execution of commands between 'do' and 'done', the loop goes back to the top and select the next item from the list and repeat the whole process.
* The list can contain numbers or string etc. separated by spaces.

We can increase or decrease a specified value by adding two another dots (..) and the value to step by, e.g., {START..END..INCREMENT}.

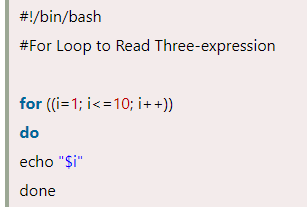
 

We can use 'for loop' to iterate the values of an array.

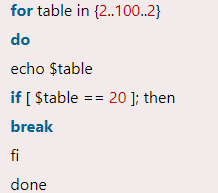


### For Loop to Read Three-expression:-

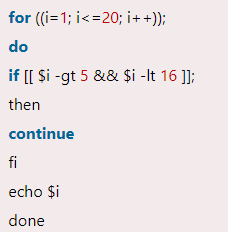
Three expression syntax is the most common syntax of 'for loop'. The first expression refers to the process of initialization, the second expression refers to the termination, and the third expression refers to the increment or decrement.

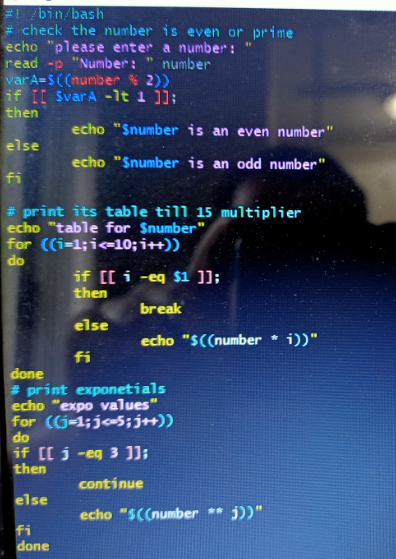


A 'break' statement can be used inside 'for' loop to terminate from the loop.



We can use the 'continue' statement inside the 'for' loop to skip any specific statement on a particular condition. It tells Bash to stop executing that particular iteration of the loop and process the next iteration.



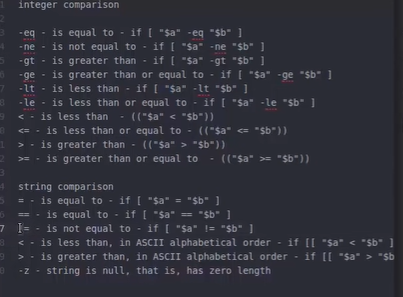


If (($var >= 10 ))

Then

//

Fi

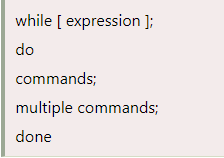


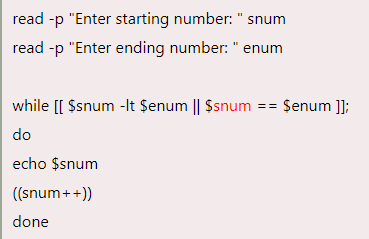
With integers, use (( ))

With strings, use [[ ]]

* Comparison operators with integers, use -ew / -gt / -lt within [ ]
* Integers With angle brackets like < , > , <= … use (( ))
* For string, use [[ ]]

2.While loop:

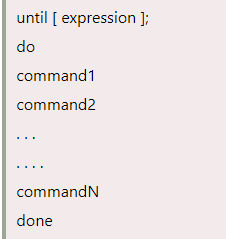




A break statement can be used to stop the loop as per the applied condition

A continue statement can be used to skip the iteration for a specific condition inside the while loop

3.Untill loop



*File Test Operation:*

-e is the flag to check file existence

If [ -e $filename ]

-f is the flag to check if it’s a regular file or not

-d is the flag to check for directory

-s is the flag to check file is not empty

-r is the flag to check file has read permission

-w is the flag to check file has write permission

-x is the flag to check the file has execute permission

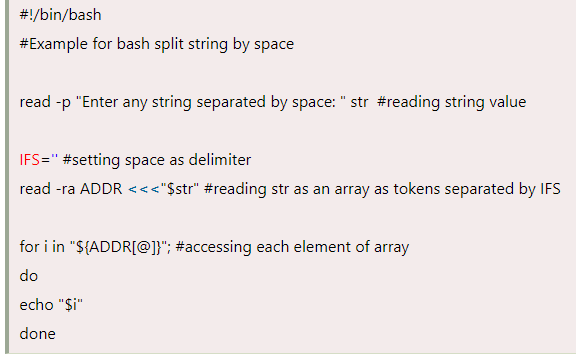
**String:**

* An equal operator (=) is used to check whether two strings are equal.
* Not equal operator (!=) is used to define that strings are not equal.
* The 'less than operator (\<)' is a conditional operator which is used to check if string1 is less than string2.
* The 'greater than operator (\>)` is used to check if string1 is greater than string2.
* This operator is used to check if the string is zero or greater than zero. [ -n Operand ]
* This operator is used to check if the string is empty or equal to zero. [ -z Operand ]

Split:

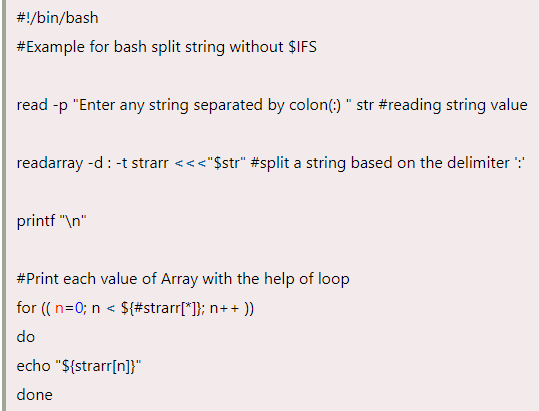
*Using $IFS:*

* $IFS is a special internal variable which is used to split a string into words. $IFS variable is called '**Internal Field Separator**' which determines how Bash recognizes boundaries. $IFS is used to assign the specific delimiter [**IFS=''**] for dividing the string. The white space is a default value of $IFS. However, we can also use values such as '\t', '\n', '-' etc. as the delimiter.
* After assigning the delimiter, a string can be read by two options: '-r' and '-a'. i.e., **read -ra ARR <<< "$str"**.  
  Here, the option '-r' is used to define that backslash (\) is a character rather than escape character. The '-a' option is used to define that the words (separated by $IFS) are assigned to the sequential index of array beginning at zero.
* Then we apply bash 'for' loop to access the tokens which are split into an array.



*Normal way:*

he 'readarray' command with -d option is used to split the string data. The -d option is applied to define the separator character in the command like $IFS. Moreover, the bash loop is used to print the string in split form.



**Substring:**

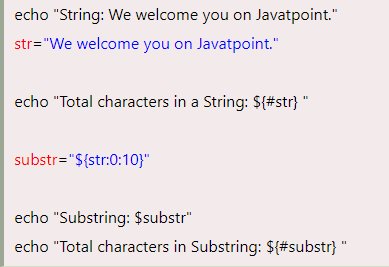
The syntax of the substring extraction can be defined as:

${variable:offset:length}

Variable: var name holding the string

Offset: position to start

-ve value for back direction



**Functions:**

Following are some key points about bash functions:

* A function has to be declared in the shell script before we can use it.
* Arguments can be passed to the functions and accessed inside the function as $1, $2, etc.
* Local variables can be assigned within the function, and the scope of such variables will only be that particular function.
* Built-in commands of Bash shell can be overridden using functions.

1. The first method starts with the function name, followed by parentheses. It is the most preferred and commonly used method:

function\_name () {

commands

}

The second method starts with the function reserved word, followed by the function name:

function function\_name {

commands

}

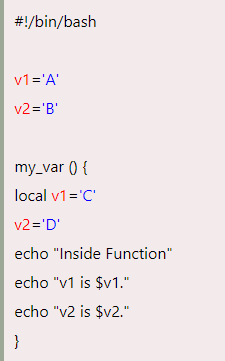
passing arguments to the bash functions

* The $\* and $@ variables are used to hold all the arguments/ parameters given to the function.
  + When $\* is used with double quotes (i.e., "$\*" ), it expands to a single string separated by the space. For example, "$1 $2 $n etc".
  + When $@ is used with double quotes (i.e., "$@" ), it expands to the separate string. For example, "$1" "$2" "$n" etc.
  + When $\* and $# are not used with the double quotes, they both are the same.

We can create local variables with the same name in different functions. To add a local variable, we can use the following syntax:

local var\_name=**<var\_value>**

it will have precedence over the global variable. Global variables can be modified within the function.



The return status can be indicated by using the 'return' keyword, and it is assigned to the variable $?. The return statement terminates the function and works as the function's exit status.

When a bash function completes, its return value is the status of the last executed statement in the function. It returns 0 for the success status and non-zero decimal number in the 1-255 range

for failure.

To perform arithmetic:

1.Using double brackets

$(( var + var2 ))

2.Using expr

$(expr $var + $var2 )

Decimal calculations are performed using bc command

Expression | bc

Considering decimals use scale=n, n= no of decimal places

Scale=2; expression | bc

<https://www.youtube.com/watch?v=zWVV31NYi1U&t=4502s>

2:23