<https://devhints.io/bash>

# "Hello, World!"

Create hello\_world.sh as follows:

#!/bin/sh

# This is a comment!

echo Hello, World!

* First line tells Linux that the file is to be executed by /bin/sh. This is the standard location of the Bourne shell on every Unix system. If you're using GNU/Linux, /bin/sh is normally a symbolic link to bash (or, more recently, dash).
* Second line begins with a #. This marks the line as a comment, and it is ignored completely by the shell.

**Note**: The only exception is when the very first line of the file starts with #!. This is a special directive Unix treats specially. It means that even if you are using csh, ksh, or anything else as your interactive shell, that what follows should be interpreted by the Bourne shell.

* Third line runs command echo, with two parameters "Hello," and "World!". Note that echo will automatically put a single space between its parameters. To put simply, it takes any number of parameters.

**Note**: Despite beng a valid syntax, it’s not a good practice to use echo without the quotes because there can be escape characters which end the inputted string unexpectedly. So, it’s **recommend to always use echo with the quotes**.

To run the above script:

1. Run chmod 755 hello\_world.sh to make the text file executable
2. Run ./first.sh

The output to the screen will be:

Hello World

# Variables and Types

## Variables

Let's look back at our first hello\_world.sh. This could be done using variables:

#!/bin/sh

msg="Hello, World!"

echo $msg # This prints out "Hello, World!"

**Notes**:

* There must be **no spaces around the "=" sign**. That means VAR=value works, but VAR = value doesn't work.
* There must have the quotes around the string "Hello, World!". With the echo command, we can ignore these quotes because echo takes any number of parameters. But a variable can only hold one value.

## Types

**In shell script, we do not care much about types of variables**. They can store strings, integers, real numbers - anything you like.

**However, if you assign a string to a variable then try to add a number to it, you will get a syntax error**. For example, the external program expr only expects numbers:

$ x="hello"

$ expr $x + 1

expr: non-numeric argument

But there is no syntactic difference between:

MESSAGE="Hello World"

SHORT\_MESSAGE=hi

NUMBER=1

PI=3.142

OTHER\_PI="3.142"

MIXED=123abc

## Scope of Variables

Create a myvar.sh:

#!/bin/sh

echo "MYVAR is: $MYVAR"

MYVAR="Hi there"

echo "MYVAR is: $MYVAR"

Now run the script:

$ ./myvar.sh

MYVAR is:

MYVAR is: Hi there

MYVAR hasn't been set to any value, so it's blank.

**Exporting variables from the interactive shell to the script**

Now we give it a value:

$ MYVAR=Hello

$ ./myvar.sh

MYVAR is:

MYVAR is: Hi there

It's still not been set! Why? When you call myvar.sh from the interactive shell, a new shell is spawned to run the script. This is partly because of the #!/bin/sh line at the start of the script.

We need to export the variable for it to be inherited by another program - including a shell script:

$ MYVAR=Hello

$ export MYVAR

$ ./myvar.sh

MYVAR is: Hello

MYVAR is: Hi there

Although we changed the MYVAR from "Hello" to "Hi there". But there is no way that this will be passed back to the shell. Once the script exits, its environment is destroyed. Try reading the value of MYVAR:

$ echo $MYVAR

Hello

**Sourcing the script**

In order to receive environment changes back from the script, we must *source* the script - this effectively runs the script within our own interactive shell, instead of spawning another shell to run it.  
We can source a script via the "." command:

$ MYVAR=Hello

$ . ./myvar.sh

MYVAR is: Hello

MYVAR is: Hi there

$ echo $MYVAR

Hi there

The change has now made it out into the interactive shell. This is how your .profile or .bash\_profile file works, for example.

## Common Mistakes

**1. Lacking of a { } bracket in some cases:**

Consider the following shell script:

#!/bin/sh

echo "What is your name?"

read USER\_NAME

echo "Hello $USER\_NAME"

echo "I will create for you a file called $USER\_NAME\_file"

touch $USER\_NAME\_file

If you enter "steve" as USER\_NAME, should the script create steve\_file? No! This will cause an error unless there is a variable called USER\_NAME\_file. The shell does not know where the variable starts and ends. How can we define this? The answer is to enclose the variable itself in curly brackets { }:

#!/bin/sh

echo "What is your name?"

read USER\_NAME

echo "Hello $USER\_NAME"

echo "I will create for you a file called ${USER\_NAME}\_file"

touch "${USER\_NAME}\_file"

The shell now knows that we are referring to the variable USER\_NAME and that we want it suffixed with "\_file".

Also note the quotes around "${USER\_NAME}\_file". If the user entered a name with space, e.g.: "Steve Parker" then without the quotes, the arguments passed to touch would be Steve and Parker\_file, which is two files to be touched, not one. So we need the quotes avoid this.

# Loops

## The 'for' Loop

**Example 1:**

#!/bin/sh

for i in {1..5}

do

    echo "Number $i"

done

Output:

Number 1

Number 2

Number 3

Number 4

Number 5

**Example 2:**

#!/bin/sh

for i in hello 1 \* 2 goodbye

do

    echo "Looping i is set to $i"

done

Output:

Looping i is set to hello

Looping i is set to 1

Looping i is set to 2

Looping i is set to goodbye

**Example 3:**

for i in \*

do

    echo "Directories are $i"

done

Output:

Directories are ggcs\_pproxy.sh

Directories are script.sh

Directories are Setup\_Environment

Directories are snmpd.conf

Directories are snmpd\_original.conf

## The 'while' Loop

**Example 1:**

#!/bin/sh

MSG="hi"

while ["$MSG" != "bye"]

do

    echo "Please type something in (bye to quit)"

    read MSG

done

The loop will end until the inputted MSG is "bye".

# Conditions

## Syntax of an if Statement

The basic syntax of an if … then statement is like this:

if <condition>; then

<commands>

fi

The condition is, depending on its type, surrounded by certain brackets (e.g. [ ]). You can add commands to be executed when the condition is false using the else keyword, and use the elif (elseif) keyword to execute commands on another condition if the primary condition is false. The else keyword always comes last. Example:

if [ -r somefile ]; then

...

elif [ -f somefile ]; then

...

else

...

fi

## Basic Rules of Conditions in Shell Scripts

**1. Always keep spaces between the brackets and the actual check/comparison**

The following won’t work:

if [$foo -ge 3]; then

Bash will complain about a “missing `]'”.

You must write:

if [ $foo -ge 3 ]; then

**2. Always terminate the line before putting a new keyword like “then”**

The words if, then, else, elif and fi are shell keywords, meaning that they cannot share the same line. Put a “;” between the previous statement and the keyword or place the keyword on the start of a new line. Or else, bash will throw errors like “syntax error near unexpected token `fi'”.

**3. It is a good habit to quote string variables if you use them in conditions**

If you don’t, they are likely to give trouble if they contain spaces and/or newlines. By quoting I mean:

if [ "$stringvar" == "tux" ]; then

There are a few cases in which you should not quote, but they are rare. You will see one of them further on in the tutorial.

Also, there are two things that may be useful to know:

1. You can invert a condition by putting an “!” in front of it. Example:

if [ ! -f regularfile ]; then

2. You can combine conditions by using certain operators. For the single-bracket syntax that we’ve been using so far, you can use “-a” for and and “-o” for or. Example:

if [ $foo -ge 3 -a $foo -lt 10 ]; then

**Note**: The square brackets ( [ ] ) in the if statement above are actually a reference to the command test. This means that all of the operators that test allows may be used here as well.

## Different Condition Syntaxes

### 1. Single-Bracket Syntax

This is the condition syntax you have already seen in the previous paragraphs; it’s the oldest supported syntax. It supports three types of conditions:

**Arithmetic (number-based) conditions**

Allows comparing integer numbers. Example:

if [ $num -lt 1 ]; then

The above condition returns true if $num is less than 1.

For more arithmetic conditions see the table below.

**File-based conditions**

Allows different kinds of checks in a file. Example:

if [ -L symboliclink ]; then

The above condition is true if the file ‘symboliclink’ exists and is a symbolic link.

For more file-based conditions see the table below.

**String-based conditions**

Allows checks on a string and comparing of strings. Example:

if [ -z "$emptystring" ]; then

The above condition is true if $emptystring is an empty string or an uninitialized variable.

Another example:

if [ "$stringvar1" == "cheese" ]; then

The above condition is true if $stringvar1 contains just the string “cheese”.

For more string-based conditions see the table below.

### 2. Double-Bracket Syntax

You may have encountered conditions enclosed in double square brackets already, which look like this:

if [[ "$stringvar" == \*string\* ]]; then

The double-bracket syntax serves as an enhanced version of the single-bracket syntax; it mainly has the same features, but also some important differences with it.

**First difference**

The first difference can be seen in the above example; when comparing strings, the double-bracket syntax features shell globbing. This means that an asterisk (“\*”) will expand to literally anything, just as you probably know from normal command-line usage. Therefore, if $stringvar contains the phrase “string” anywhere, the condition will return true. Other forms of shell globbing are allowed, too. If you’d like to match both “String” and “string”, you could use the following syntax:

if [[ "$stringvar" == \*[sS]tring\* ]]; then

Note that only general shell globbing is allowed. Bash-specific things like {1..4} or {foo,bar} will not work. Also note that the globbing will not work if you quote the right string. In this case you should leave it unquoted.

**Second difference**

The second difference is that word splitting is prevented. Therefore, you could omit placing quotes around string variables and use a condition like the following without problems:

if [[ $stringvarwithspaces != foo ]]; then

Nevertheless, the quoting string variables remains a good habit, so I recommend just to keep doing it.

**Third difference**

The third difference consists of not expanding filenames. I will illustrate this difference using two examples, starting with the old single-bracket situation:

if [ -a \*.sh ]; then

The above condition will return true if there is one single file in the working directory that has a .sh extension. If there are none, it will return false. If there are several .sh files, bash will throw an error and stop executing the script. This is because \*.sh is expanded to the files in the working directory. Using double brackets prevents this:

if [[ -a \*.sh ]]; then

The above condition will return true only if there is a file in the working directory called “\*.sh”, no matter what other .sh files exist. The asterisk is taken literally, because the double-bracket syntax does not expand filenames.

**Fourth difference**

The fourth difference is the addition of more generally known combining expressions, or, more specific, the operators “&&” and “||”. Example:

if [[ $num -eq 3 && "$stringvar" == foo ]]; then

The above condition returns true if $num is equal to 3 and $stringvar is equal to “foo”. The -a and -o known from the single-bracket syntax is supported, too.

Note that the and operator has precedence over the or operator, meaning that “&&” or “-a” will be evaluated before “||” or “-o”.

**Fifth difference**

The fifth difference is that the double-bracket syntax allows regex pattern matching using the “=~” operator. See the table for more information.

### 3. Double-Parenthesis Syntax

There also is another syntax for arithmetic (number-based) conditions, most likely adopted from the Korn shell:

if (( $num <= 5 )); then

The above condition is true if $num is less than or equal to 5. This syntax may seem more familiar to programmers. It features all the ‘normal’ operators, like “==”, “<” and “>=”. It supports the “&&” and “||” combining expressions (but not the -a and -o ones!). It is equivalent to the built-in let command.

## Table of Conditions

The following table list the condition possibilities for both the single- and the double-bracket syntax.

|  |  |  |
| --- | --- | --- |
| **1. File-based conditions** | | |
| **Condition** | **True if** | **Example/explanation** |
| [ -a existingfile ] | file ‘existingfile’ exists. | if [ -a tmp.tmp ]; then rm -f tmp.tmp # *Make sure we’re not bothered by an old temporary file* fi |
| [ -b blockspecialfile ] | file ‘blockspecialfile’ exists and is block special. | Block special files are special kernel files found in /dev, mainly used for ATA devices like hard disks, cd-roms and floppy disks.  if [ -b /dev/fd0 ]; then dd if=floppy.img of=/dev/fd0 #*Write an image to a floppy* fi |
| [ -c characterspecialfile ] | file ‘characterspecialfile’ exists and is character special. | Character special files are special kernel files found in /dev, used for all kinds of purposes (audio hardware, tty’s, but also /dev/null).  if [ -c /dev/dsp ]; then cat raw.wav > /dev/dsp #*This actually works for certain raw wav files* fi |
| [ -d directory ] | file ‘directory’ exists and is a directory. | In UNIX-style, directories are a special kind of file.  if [ -d ~/.kde ]; then echo “You seem to be a kde user.” fi |
| [ -e existingfile ] | file ‘existingfile’ exists. | (same as -a, see that entry for an example) |
| [ -f regularfile ] | file ‘regularfile’ exists and is a regular file. | A regular file is neither a block or character special file nor a directory.  if [ -f ~/.bashrc ]; then source ~/.bashrc fi |
| [ -g sgidfile ] | file ‘sgidfile’ exists and is set-group-ID. | When the SGID-bit is set on a directory, all files created in that directory will inherit the group of the directory.  if [ -g . ]; then echo “Created files are inheriting the group ‘$(ls -ld . | awk ‘{ print $4 }’)’ from the working directory.” fi |
| [ -G fileownedbyeffectivegroup ] | file ‘fileownedbyeffectivegroup’ exists and is owned by the effective group ID. | The effective group id is the primary group id of the executing user.  if [ ! -G file ]; then #*An exclamation mark inverts the outcome of the condition following it* chgrp $(id -g) file #*Change the group if it’s not the effective one* fi |
| [ -h symboliclink ] | file ‘symboliclink’ exists and is a symbolic link. | if [ -h $pathtofile ]; then pathtofile=$(readlink -e $pathtofile) #*Make sure $pathtofile contains the actual file and not a symlink to it* fi |
| [ -k stickyfile ] | file ‘stickyfile’ exists and has its sticky bit set. | The sticky bit has got [quite a history](http://en.wikipedia.org/wiki/Sticky_bit), but is now used to prevent world-writable directories from having their contents deletable by anyone.  if [ ! -k /tmp ]; then #*An exclamation mark inverts the outcome of the condition following it* echo “Warning! Anyone can delete and/or rename your files in /tmp!” fi |
| [ -L symboliclink ] | file ‘symboliclink’ exists and is a symbolic link. | (same as -h, see that entry for an example) |
| [ -N modifiedsincelastread ] | file ‘modifiedsincelastread’ exists and was modified after the last read. | if [ -N /etc/crontab ]; then killall -HUP crond #*SIGHUP makes crond reread all crontabs* fi |
| [ -O fileownedbyeffectiveuser ] | file ‘fileownedbyeffectiveuser’ exists and is owned by the user executing the script. | if [ -O file ]; then chmod 600 file #*Makes the file private, which is a bad idea if you don’t own it* fi |
| [ -p namedpipe ] | file ‘namedpipe’ exists and is a named pipe. | A named pipe is a file in /dev/fd/ that can be read just once. See [my bash tutorial](http://www.linuxtutorialblog.com/post/tutorial-the-best-tips-tricks-for-bash#using-several-ways-of-substitution) for a case in which it’s used.  if [ -p $file ]; then cp $file tmp.tmp #*Make sure we’ll be able to read* file=”tmp.tmp”    #*the file as many times as we like* fi |
| [ -r readablefile ] | file ‘readablefile’ exists and is readable to the script. | if [-r file ]; then content=$(cat file) #*Set $content to the content of the file* fi |
| [ -s nonemptyfile ] | file ‘nonemptyfile’ exists and has a size of more than 0 bytes. | if [ -s logfile ]; then gzip logfile    #*Backup the old logfile* touch logfile #*before creating a fresh one.* fi |
| [ -S socket ] | file ‘socket’ exists and is a socket. | A socket file is used for inter-process communication, and features an interface similar to a network connection.  if [ -S /var/lib/mysql/mysql.sock ]; then mysql –socket=/var/lib/mysql/mysql.sock #*See*[*this MySQL tip*](http://www.tech-recipes.com/mysql_tips762.html)fi |
| [ -t openterminal ] | file descriptor ‘openterminal’ exists and refers to an open terminal. | Virtually everything is done using files on Linux/UNIX, and the terminal is no exception.  if [ -t /dev/pts/3 ]; then echo -e “nHello there. Message from terminal $(tty) to you.” > /dev/pts/3 #*Anyone using that terminal will actually see this message!* fi |
| [ -u suidfile ] | file ‘suidfile’ exists and is set-user-ID. | Setting the suid-bit on a file causes execution of that file to be done with the credentials of the owner of the file, not of the executing user.  if [ -u executable ]; then echo “Running program executable as user $(ls -l executable | awk ‘{ print $3 }’).” fi |
| [ -w writeablefile ] | file ‘writeablefile’ exists and is writeable to the script. | if [ -w /dev/hda ]; then grub-install /dev/hda fi |
| [ -x executablefile ] | file ‘executablefile’ exists and is executable for the script. | Note that the execute permission on a directory means that it’s searchable (you can see which files it contains).  if [ -x /root ]; then echo “You can view the contents of the /root directory.” fi |
| [ newerfile -nt olderfile ] | file ‘newerfile’ was changed more recently than ‘olderfile’, or if ‘newerfile’ exists and ‘olderfile’ doesn’t. | if [ story.txt1 -nt story.txt ]; then echo “story.txt1 is newer than story.txt; I suggest continuing with the former.” fi |
| [ olderfile -ot newerfile ] | file ‘olderfile’ was changed longer ago than ‘newerfile’, or if ‘newerfile’ exists and ‘olderfile’ doesn’t. | if [ /mnt/remote/remotefile -ot localfile ]; then cp -f localfile /mnt/remote/remotefile #*Make sure the remote location has the newest version of the file, too* fi |
| [ same -ef file ] | file ‘same’ and file ‘file’ refer to the same device/inode number. | if [ /dev/cdrom -ef /dev/dvd ]; then echo “Your primary cd drive appears to read dvd’s, too.” fi |
| **2. String-based conditions** | | |
| **Condition** | **True if** | **Example/explanation** |
| [ STRING1 == STRING2 ] | STRING1 is equal to STRING2. | if [ “$1” == “moo” ]; then echo $cow #*Ever tried executing ‘apt-get moo’?* fiNote: you can also use a single “=” instead of a double one. |
| [ STRING1 != STRING2 ] | STRING1 is not equal to STRING2. | if [ “$userinput” != “$password” ]; then echo “Access denied! Wrong password!” exit 1 #*Stops script execution right here* fi |
| [ STRING1 > STRING2 ] | STRING1 sorts after STRING2 in the current locale (lexographically). | The backslash before the angle bracket is there because the bracket needs to be escaped to be interpreted correctly. As an example we have a basic [bubble sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Bubble_sort):  *(Don’t feel ashamed if you don’t understand this, it is a more complex example)*  array=( linux tutorial blog ) swaps=1 while (( swaps > 0 )); do  swaps=0 for (( i=0; i < (( ${#array[@]} – 1 )) ; i++ )); do if [ “${array[$i]}” > “${array[$(( i + 1 ))]}” ]; then #*Here is the sorting condition* tempstring=${array[$i]} array[$i]=${array[$(( i + 1 ))]} array[$(( i + 1 ))]=$tempstring (( swaps=swaps + 1 )) fi done done echo ${array[@]} #*Returns “blog linux tutorial”* |
| [ STRING1 < STRING2 ] | STRING1 sorts before STRING2 in the current locale (lexographically). |
| [ -n NONEMPTYSTRING ] | NONEMPTYSTRING has a length of more than zero. | This condition only accepts valid strings, so be sure to quote anything you give to it.  if [ -n “$userinput” ]; then userinput=parse($userinput) # *Only parse if the user actually gave some input.* fi  Note that you can also omit the “-n”, as brackets with just a string in it behave the same. |
| [ -z EMPTYSTRING ] | EMPTYSTRING is an empty string. | This condition also accepts non-string input, like an uninitialized variable:  if [ -z $uninitializedvar ]; then uninitializedvar=”initialized” #*-z returns true on an uninitialized variable, so we initialize it here.* fi |
| *Double-bracket syntax only:* [[ STRING1 =~ REGEXPATTERN ]] | STRING1 matches REGEXPATTERN. | If you are familiar with Regular Expressions, you can use this conditions to perform a regex match.  if [[ “$email” =~ “b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4}b” ]]; then echo “$email contains a valid e-mail address.” fi |
| **3. Arithmetic (number-based) conditions** | | |
| **Condition** | **True if** | **Example/explanation** |
| [ NUM1 -eq NUM2 ] | NUM1 is **EQ**ual to NUM2. | These conditions only accept integer numbers. Strings will be converted to integer numbers, if possible. Some random examples:  if [ $? -eq 0 ]; then #*$? returns the exit status of the previous command* echo “Previous command ran succesfully.” fi  if [ $(ps -p $pid -o ni=) -ne $(nice) ]; then echo “Process $pid is running with a non-default nice value” fi  if [ $num -lt 0 ]; then echo “Negative numbers not allowed; exiting…” exit 1 fi |
| [ NUM1 -ne NUM2 ] | NUM1 is **N**ot **E**qual to NUM2. |
| [ NUM1 -gt NUM2 ] | NUM1 is **G**reater **T**han NUM2. |
| [ NUM1 -ge NUM2 ] | NUM1 is **G**reater than or **E**qual to NUM2. |
| [ NUM1 -lt NUM2 ] | NUM1 is **L**ess **T**han NUM2. |
| [ NUM1 -le NUM2 ] | NUM1 is **L**ess than or **E**qual to NUM2. |
| **4. Miscellaneous conditions** | | |
| **Condition** | **True if** | **Example/explanation** |
| [ -o shelloption ] | shell option ‘shelloption’ is enabled. | Shell options modify the behaviour of bash, except a few unmodifiable ones that indicate the shell status.  if [ ! -o checkwinsize ] #*An exclamation mark inverts the outcome of the condition following it* echo “Shell option checkwinsize is disabled; enabling it so you can resize you terminal window without problems.” shopt -s checkwinsize #*This shell option is modifiable* fi  if [ -o login\_shell ]; then echo “This a a login shell.” #*This shell option is not modifiable*  fi |

With the double-parenthesis syntax, you can use the following conditions:

|  |  |  |
| --- | --- | --- |
| **5. Double-parenthesis syntax conditions** | | |
| **Condition** | **True if** | **Example/explanation** |
| (( NUM1 == NUM2 )) | NUM1 is equal to NUM2. | These conditions only accept integer numbers. Strings will be converted to integer numbers, if possible. Some random examples:  if (( $? == 0 )); then #*$? returns the exit status of the previous command* echo “Previous command ran succesfully.” fi  if (( $(ps -p $pid -o ni=) != $(nice) )); then echo “Process $pid is running with a non-default nice value” fi  if (( $num < 0 )); then echo “Negative numbers not allowed; exiting…” exit 1 fi |
| (( NUM1 != NUM2 )) | NUM1 is not equal to NUM2. |
| (( NUM1 > NUM2 )) | NUM1 is greater than NUM2. |
| (( NUM1 >= NUM2 )) | NUM1 is greater than or equal to NUM2. |
| (( NUM1 < NUM2 )) | NUM1 is less than NUM2. |
| (( NUM1 <= NUM2 )) | NUM1 is less than or equal to NUM2. |

After this dry information load, here’s a bit of explanation for those who want to know more…

Ref: <https://linuxacademy.com/blog/linux/conditions-in-bash-scripting-if-statements/>

## Case Statements

Read <https://ryanstutorials.net/bash-scripting-tutorial/bash-if-statements.php>

# Table of Shell Scripts Arguments and Operators

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| && | Logical AND | if [ "$foo" -ge "0" ] && [ "$foo" -le "9"] |
| || | Logical OR | if [ "$foo" -lt "0" ] || [ "$foo" -gt "9" ] (not in Bourne shell) |
| ^ | Start of line | grep "^foo" |
| $ | End of line | grep "foo$" |
| = | String equality (cf. -eq) | if [ "$foo" = "bar" ] |
| ! | Logical NOT | if [ "$foo" != "bar" ] |
| $$ | PID of current shell | echo "my PID = $$" |
| $! | PID of last background command | ls & echo "PID of ls = $!" |
| $? | exit status of last command | ls ; echo "ls returned code $?" |
| $0 | Name of current command (as called) | echo "I am $0" |
| $1 | Name of current command's first parameter | echo "My first argument is $1" |
| $9 | Name of current command's ninth parameter | echo "My ninth argument is $9" |
| $@ | All of current command's parameters (preserving whitespace and quoting) | echo "My arguments are $@" |
| $\* | All of current command's parameters (not preserving whitespace and quoting) | echo "My arguments are $\*" |
| -eq | Numeric equality | if [ "$foo" -eq "9" ] |
| -ne | Numeric inquality | if [ "$foo" -ne "9" ] |
| -lt | Less than | if [ "$foo" -lt "9" ] |
| -le | Less than or equal | if [ "$foo" -le "9" ] |
| -gt | Greater than | if [ "$foo" -gt "9" ] |
| -ge | Greater than or equal | if [ "$foo" -ge "9" ] |
| -z | String is zero length | if [ -z "$foo" ] |
| -n | String is not zero length | if [ -n "$foo" ] |
| -nt | Newer than | if [ "$file1" -nt "$file2" ] |
| -d | Is a directory | if [ -d /bin ] |
| -f | Is a File | if [ -f /bin/ls ] |
| -r | Is a readable file | if [ -r /bin/ls ] |
| -w | Is a writable file | if [ -w /bin/ls ] |
| -x | Is an executable file | if [ -x /bin/ls ] |
| ( ... ) | Function definition | function myfunc() { echo hello } |

# Importing Data from a Text File to a Bash Script

**Example 1:**

IFS stands for Input Internal Field Separator which is a character that separates fields. In the below example,

#!/bin/bash

IFS=$'\n' # make newlines the only separator

set -f # disable globbing

for line in $(cat < inputData.txt)

do

echo "$line"

done

IFS is set to new line character (\n). After you set it, for will process text line by line.

If the content of the inputData.txt is:

AMBI

CBNK

CCXI

CERE

CLACW

The output of the code above will be:

AMBI

CBNK

CCXI

CERE

CLACW

If we change IFS=$'\n' to IFS=$'\K', the output of the code above will change to:

AMBI

CBN

CCXI

CERE

CLACW

**Example 2: This code example gives the same result as the above example**

while IFS= read -r line

do

echo "$line"

done < inputData.txt

More details:

<https://unix.stackexchange.com/questions/26784/understanding-ifs>

<https://unix.stackexchange.com/questions/88100/importing-data-from-a-text-file-to-a-bash-script>

# Differences Between Shell Scripts and Bash Scripts

## Shell Scripts

"Shell" is a program, which facilitates the interaction between the user and operating system (kernel). There are many shells available, like Bourne shell (**sh**), Bourne again shell (**bash**), Korn shell (**ksh**), C shell (**csh**) and Z shell (**zsh**).

"Shell" scripting is a way of automating things, in the form of collection of commands. The control of execution is steered by the predefined control statements.

## Bash Scripts

“Bash” stands for "**B**ourne **A**gain **Sh**ell". And as its name suggest, “Bash shell" scripting is, a kind of shell scripting only. You can say, its a subset of "shell" scripting.

Bash is the most widely used shell. It comes with Linux by default, having backward compatibility with sh (though sh is also there). But, you can choose any shell you want.

In practice, "shell script" and "bash script" are often used interchangeably, unless the shell in question is not Bash.

More details:

<https://stackoverflow.com/questions/5725296/difference-between-sh-and-bash>

<https://medium.com/@varunkumar_53845/sh-vs-bash-a-summary-50f92a719e0d>