Bash Scripting Tutorial for Beginners

Lubos Rendek | Programming & Scripting | 27 May 2020

Bash Shell Scripting Definition

Bash

Bash is a command language interpreter. It is widely available on various operating systems and is a default command interpreter on most GNU/Linux systems. The name is an acronym for the '**B**ourne-**A**gain **SH**ell'.

Shell

Contents

Shell is a macro processor which allows for an interactive or non-interactive command execution.

Scripting

Scripting allows for an automatic commands execution that would otherwise be executed interactively one-by-one.

Bash Shell Script Basics

Do not despair if you have not understood any of the above **Bash Shell Scripting** definitions. It is perfectly normal, in fact, this is precisely why you are reading this Bash Scripting tutorial.

In case you did not know, Bash Scripting is a must skill for any Linux system administration job even though it may not be implicitly requested by the employer.

What is Shell

Most likely, your are at the moment sitting in front of your computer, have a terminal window opened and wondering: "What should I do with this thing?"

Well, the terminal window in front of you contains shell, and shell allows you by use of commands to interact with your computer, hence retrieve or store data, process information and various other simple or even extremely complex tasks.

Try it now! Use your keyboard and type some commands such as `date`, `cal`, `pwd` or `ls` followed by the `ENTER` key.

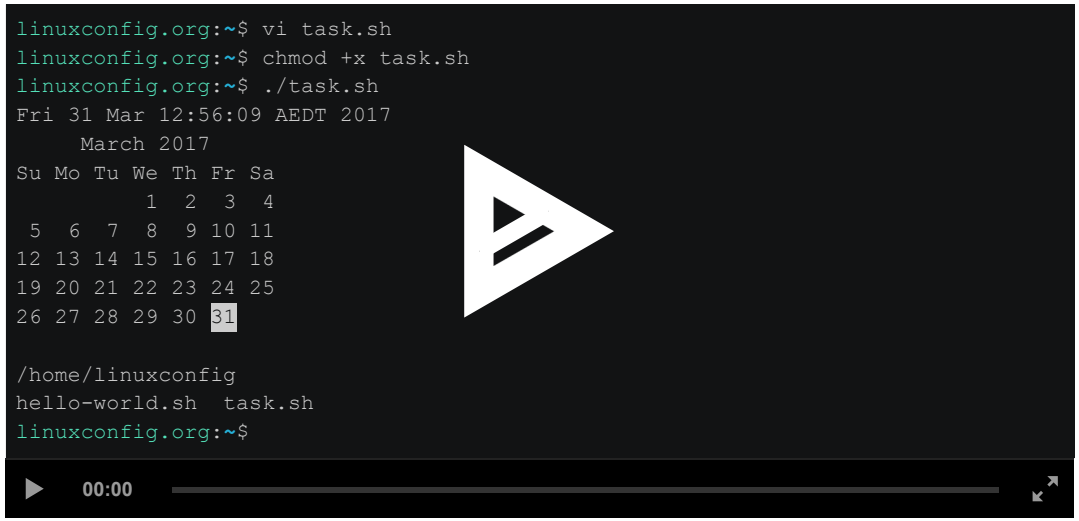
What you have just done, was that by use of commands and shell you interacted with your computer to retrieve a current date and time (`date`), looked up a calendar (`cal`), checked the location of your current working directory (`pwd`) and retrieved a list of all files and directories located within (`ls`).

What is Scripting

Now, imagine that the execution of all the above commands is your daily task. Every day you are required to execute all of the above commands without fail as well as store the observed information. Soon enough this will become an extremely tedious task destined for failure. Thus the obvious notion is to think of some way to execute all given commands together. This is where scripting becomes your salvation.

To see what is meant by scripting, use shell in combination with your favorite text editor eg. vi to create a new file called `task.sh` containing all the above

commands, each on a separate line. Once ready, make your new file executable using `chmod` command with an option `+x`. Lastly, execute your new script by prefixing its name with `./`.



As you can see, by use of scripting, any shell interaction can be automated and scripted. Furthermore, it is now possible to automatically execute our new shell script `task.sh` daily at any given time by use of cron time-based job scheduler and store the script's output to a file every time it is executed. However, this is a tale for an another day, for now let's just concentrate on a task ahead.

What is Bash

So far we have covered shell and scripting. What about Bash? Where does the bash fit in? As already mentioned, the bash is a default interpreter on many GNU/Linux systems, thus we have been using it even without realising. This is why our previous shell script works even without us defining bash as an interpreter. To see what is your default interpreter execute command `echo $SHELL`:

```
$ echo $SHELL
/bin/bash
```

There are various other shell interpreters available, such as Korn shell, C shell and more. From this reason, it is a good practice to define the shell interpreter to be used explicitly to interpret the script's content.

To define your script's interpreter as Bash, first locate a full path to its executable binary using `which` command, prefix it with a shebang `#!` and insert it as the first line of your script. There are various other techniques how to define shell interpreter, but this is a solid start.

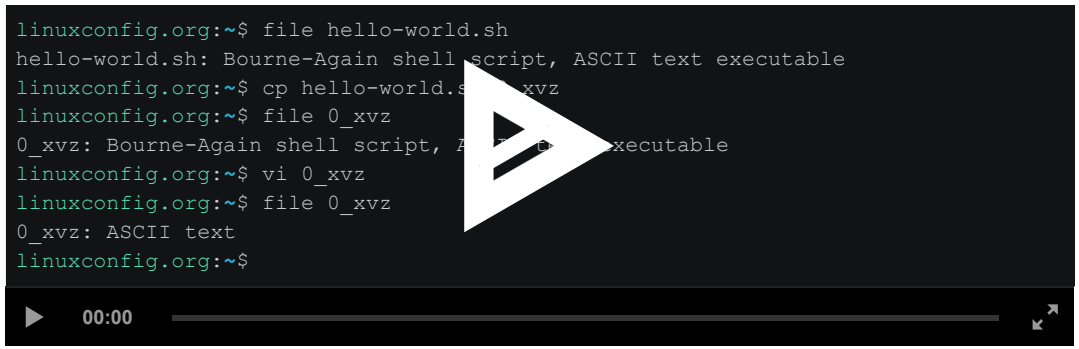
From now, all our scripts will include shell interpreter definition `#!/bin/bash`.

File Names and Permissions

Next, let's briefly discuss file permissions and filenames. You may have already noticed that in order to execute shell script the file needs to be made executable by use of `chmod +x FILENAME` command. By default, any newly created files are not executable regardless of its file extension suffix.

In fact, the file extension on GNU/Linux systems mostly does not have any meaning apart from the fact, that upon the execution of `ls` command to list all files and directories it is immediately clear that file with extension `.sh` is plausibly a shell script and file with `.jpg` is likely to be a lossy compressed image.

On GNU/Linux systems a `file` command can be used to identify a type of the file. As you can see on the below example, the file extension does not hold any value, and the shell interpreter, in this case, carries more weight.

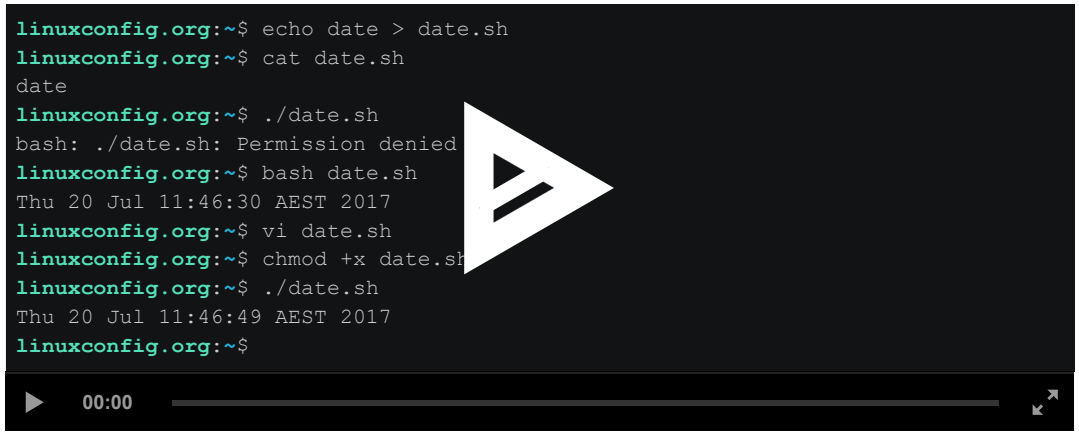


Thus, shell script name `0_xyz` is perfectly valid, but if possible it should be avoided.

Script Execution

Next, let's talk about an alternative way on how to run bash scripts. In a highly simplistic view, a bash script is nothing else just a text file containing

instructions to be executed in order from top to bottom. How the instructions are interpreted depends on defined shebang or the way the script is executed. Consider the following video example:



Another way to execute bash scripts is to call bash interpreter explicitly eg. `$ bash date.sh`, hence executing the script without the need to make the shell script executable and without declaring shebang directly within a shell script. By calling bash executable binary explicitly, the content of our file `date.sh` is loaded and interpreted as Bash Shell Script.

Relative vs Absolute Path

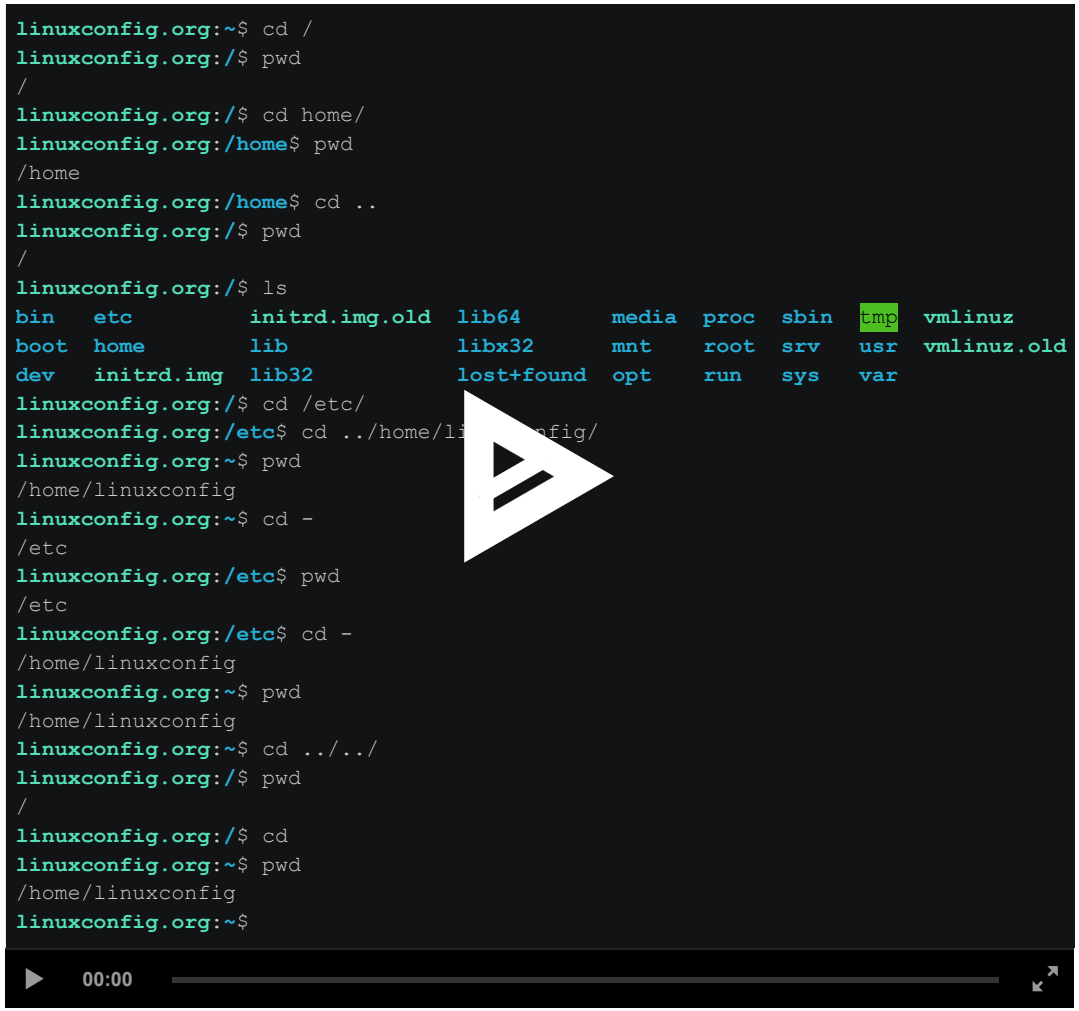
Lastly, before we program our first official bash shell script, let's briefly discuss shell navigation and the difference between a relative and absolute file path.

Probably the best analogy to explain a relative vs. absolute file path is to visualise GNU/Linux filesystem as a multiple storey building. The root directory (building's entrance door) indicated by `/` provides the entry to the entire filesystem (building), hence giving access to all directories (levels/rooms) and files (people).

To navigate to a room 1 on level 3 we first need to enter the main door `/`, then make our way to level 3 `level3/` and from there enter the `room1`.

Hence, the absolute path to this particular room within a building is `/level3/room1`. From here, if we wish to visit room2 also on level3 we first need to leave our current location that is room1 by entering `../` and then include room's name `room2`. We took a relative path to room2 which in this case is `../room2`. We were already on level 3, so there was no need to leave the entire building and take absolute path via main entrance `/level3/room2`.

Fortunately, GNU/Linux features a simple compass tool to help you navigate throughout the filesystem in the form of `pwd` command. This command, when executed, will always print your current location. The following example will use `cd` and `pwd` command to navigate GNU/Linux filesystem using absolute and relative paths.



Quick Tip:

Execute `cd` command without any arguments to instantly navigate to your user home directory from any location. Execute `cd -` to toggle between your last two visited locations. In what directory you end up after executing `cd ~` and `cd .` commands?

Navigation through GNU/Linux filesystem is a simple and yet to many a very confusing topic. Familiarise yourself with GNU/Linux filesystem navigation before you move on to next sections of this tutorial.

Hello World Bash Shell Script

Now, it is time to write our first, most basic bash shell script. The whole purpose of this script is nothing else but print "Hello World" using `echo` command to the terminal output. Using any text editor create a new file named `hello-world.sh` containing the below code:

```
#!/bin/bash

echo "Hello World"
```

Once ready, make your script executable with the `chmod` command and execute it using relative path `./hello-world.sh`:

```
$ chmod +x hello-world.sh
$ linuxconfig.org:~$ ./hello-world.sh
Hello World
$
```

The following video example offers an alternative way of creating the above `hello-world.sh` script. It uses `which` command to print a full path to the bash interpreter. This output is simultaneously redirected using `>` redirection sign while creating a new file `hello-world.sh` at the same time.

Simple Backup Bash Shell Script

Let's discuss a command line execution and how GNU/Linux commands fit into the shell script creation process in more detail.

Any command which can be successfully executed directly via bash shell terminal can be in the same form used as part of bash shell script. In fact, there is no difference between command execution directly via terminal or within a shell script apart from the fact that the shell script offers non-interactive execution of multiple commands as a single process.

> **Quick Tip:**
>
> Regardless of the script complexity, do not attempt to write your entire script in one go. Slowly develop your script by testing each core line by executing it first on the terminal command line. When successful, transfer it to your shell script.

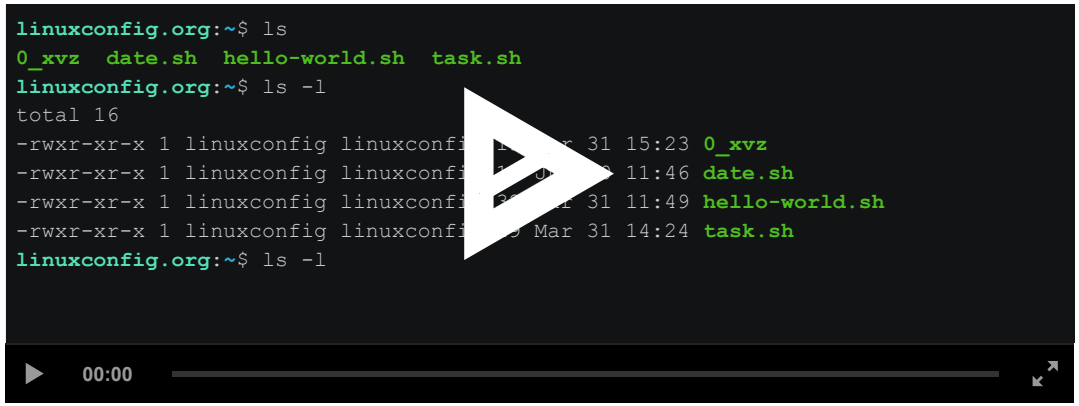
Additionally, most commands accept so called options and arguments. Command options are used to modify command's behaviour to produce alternative output results and are prefixed by `-`. Arguments may specify command's execution target such as file, directory, text and more.

Each command comes with a manual page which can be used to learn about its function as well as what options and arguments each specific command

accepts.

Use `man` command to display manual page of any desired command. For example to display a manual page for the `ls` command execute `man ls`. To quit from manual page press `q` key.

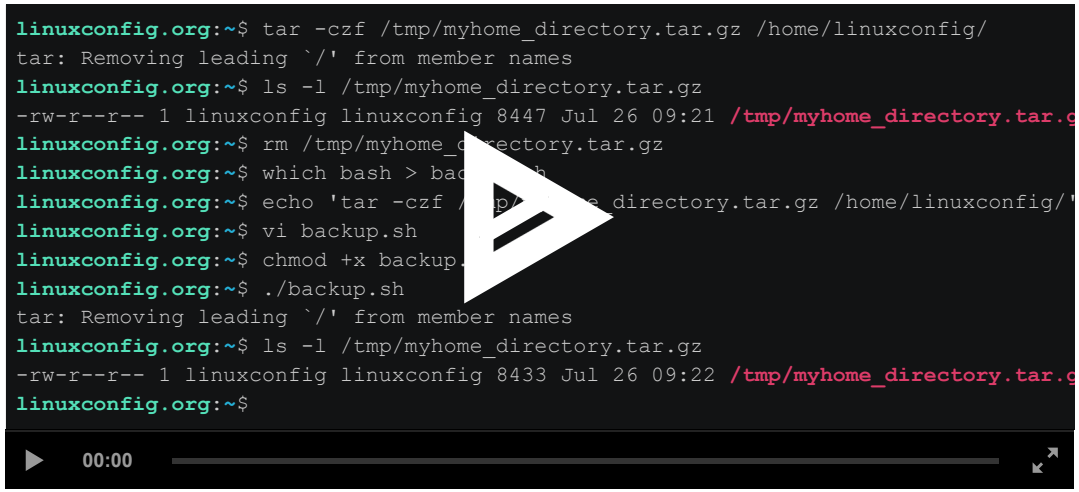
The below `ls` command example shows a basic use of command line options and arguments.



Although our first "Hello World" shell script requires a solid understanding of the file creation, editing and script execution, its usability can be clearly questioned.

The next example offers more practical application as it can be used to backup our user home directory. To create the backup script, on **Line 3** we will be using `tar` command with various options `-czf` in order to create a compressed tar ball of entire user home directory `/home/linuxconfig/`. Insert the following code into a new file called `backup.sh`, make the script executable and run it:

```
#!/bin/bash

tar -czf /tmp/myhome_directory.tar.gz /home/linuxconfig
```



Quick Tip:

Enter `man tar` command to learn more about all `tar` command line options used within the previous `backup.sh` script. Try to run the `tar` command without `-` option prefix! Does it work?

Variables

Variables are the essence of programming. Variables allow a programmer to store data, alter and reuse them throughout the script. Create a new script `welcome.sh` with the following content:

```
#!/bin/bash

greeting="Welcome"
user=$(whoami)
day=$(date +%A)
```

```
echo "$greeting back $user! Today is $day, which is the best day of the entire week!"
echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"
```

By now you should possess all required skills needed to create a new script, making it executable and running it on the command line. After running the above `welcome.sh` script, you will see an output similar to the one below:

```
$ ./welcome.sh
Welcome back linuxconfig! Today is Wednesday, which is the best da
Your Bash shell version is: 4.4.12(1)-release. Enjoy!
```

Let's look at the script more closely. First, we have declared a variable `greeting` and assigned a string value `Welcome` to it. The next variable `user` contains a value of user name running a shell session. This is done through a technique called command substitution. Meaning that the output of the `whoami` command will be directly assigned to the user variable. The same goes for our next variable `day` which holds a name of today's day produced by `date +%A` command.

The second part of the script utilises the `echo` command to print a message while substituting variable names now prefixed by `$` sign with their relevant values. In case you wonder about the last variable used `$BASH_VERSION` know that this is a so called internal variable defined as part of your shell.

Quick Tip:

Never name your private variables using UPPERCASE characters. This is because uppercase variable names are reserved for internal shell variables, and you run a risk of overwriting them. This may lead to the dysfunctional or misbehaving script execution.

Variables can also be used directly on the terminal's command line. The following example declares variables `a` and `b` with integer data. Using `echo`

command, we can print their values or even perform an arithmetic operation as illustrated by the following example:

Now that we have bash variable introduction behind us we can update our backup script to produce more meaningful output file name by incorporating a date and time when the backup on our home directory was actually performed.

Furthermore, the script will no longer be bind to a specific user. From now on our `backup.sh` bash script can be run by any user while still backing up a correct user home directory:

```
      #!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

tar -czf $output $input
echo "Backup of $input completed! Details about the output backup file:"
ls -l $output
```

You may have already noticed that the above script introduces two new bash scripting concepts. Firstly, our new `backup.sh` script contains comment line. Every line starting with `#` sign except shebang will not be interpreted by bash and will only serve as a programmer's internal note.

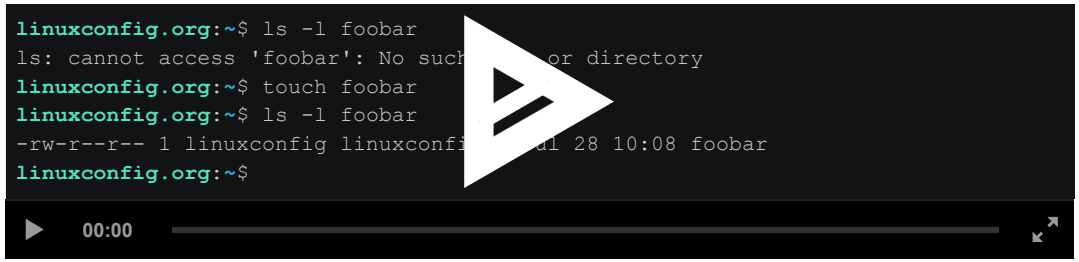
Secondly, the script uses a new shell scripting trick `${parameter}` called parameter expansion. In our case, curly braces `{}` are required because our variable `$user` is followed by characters which are not part of its variable name. Below is the output of our newly revised backup script:

```
$ ./backup.sh
tar: Removing leading `/' from member names
Backup of /home/linuxconfig completed! Details about the output ba
-rw-r--r-- 1 linuxconfig linuxconfig 8778 Jul 27 12:30 /tmp/linuxc
```


Input, Output and Error Redirections

Normally commands executed on GNU/Linux command line either produce output, require input or throw an error message. This is a fundamental concept for shell scripting as well as for working with GNU/Linux's command line in general.

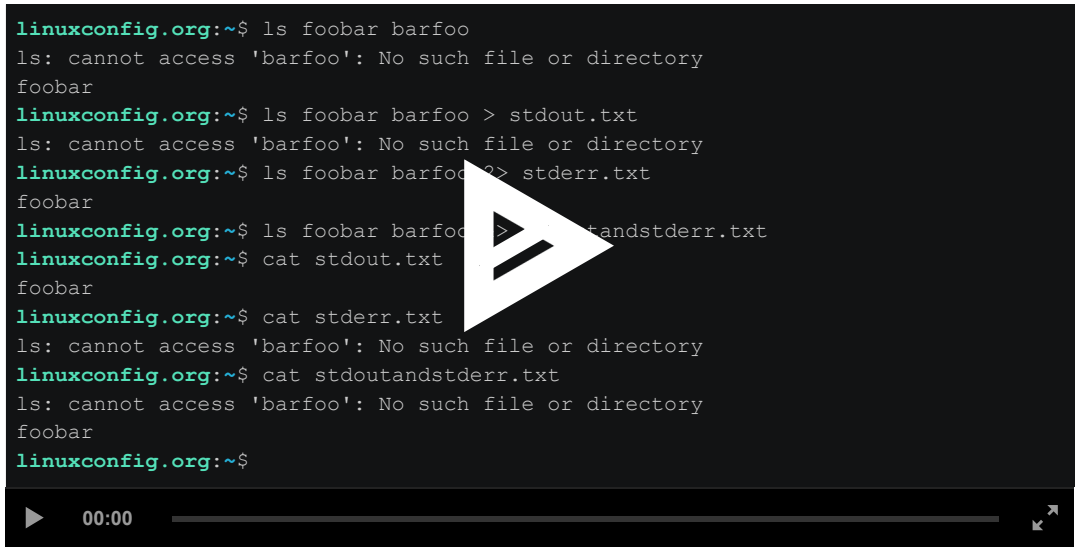
Every time, you execute a command, three possible outcomes might happen. The first scenario is that the command will produce an expected output, second, the command will generate an error, and lastly, your command might not produce any output at all:



What are we most interested in here is the output of both `ls -l foobar` commands. Both commands produced an output which by default is displayed on your terminal. However, both outputs are fundamentally different.

The first command tries to list non-existing file `foobar` which, in turn, produces a standard error output (stderr). Once the file is created by `touch` command, the second execution of the `ls` command produces standard output (stdout).

The difference between **stdout** and **stderr** output is an essential concept as it allows us to a threat, that is, to redirect each output separately. The `>` notation is used to redirect **stdout** to a file whereas `2>` notation is used to redirect **stderr** and `&>` is used to redirect both **stdout** and **stderr**. The `cat` command is used to display a content of any given file. Consider a following example:



Replay the above video few times and make sure that you understand the redirection concept shown.

> **Quick Tip:**
>
> When unsure whether your command produced **stdout** or **stderr** try to redirect its output. For example, if you are able to redirect its output successfully to a file with `2>` notation, it means that your command produced **stderr**. Conversely, successfully redirecting command output with `>` notation is indicating that your command produced **stdout**.

Back to our backup.sh script. When executing our backup script, you may have noticed an extra message display by tar command:

```
tar: Removing leading `/' from member names
```

Despite the message's informative nature, it is sent to **stderr** descriptor. In a nutshell, the message is telling us that the absolute path has been removed

thus extraction of the compressed file not overwrite any existing files.

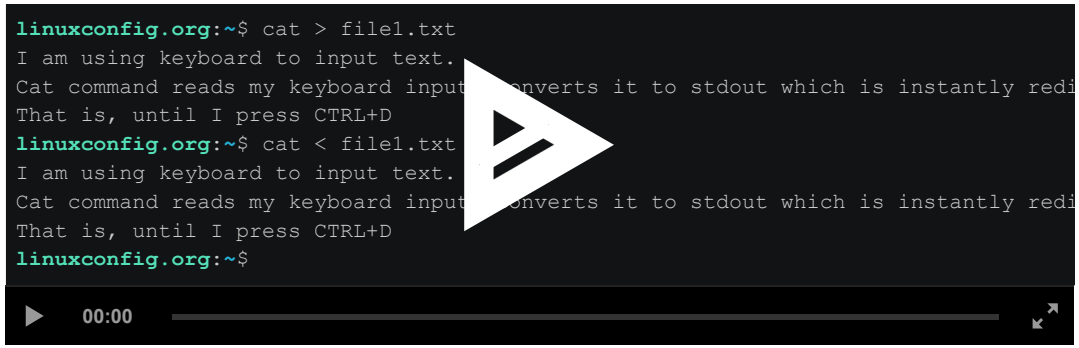
Now that we have a basic understanding of the output redirection we can eliminate this unwanted **stderr** message by redirecting it with `2>` notation to `/dev/null`. Imagine `/dev/null` as a data sink, which discards any data redirected to it. For more information run `man null`. Below is our new `backup.sh` version including tar's **stderr** redirection:

```
      #!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

tar -czf $output $input 2> /dev/null
echo "Backup of $input completed! Details about the output backup file:"
ls -l $output
```

After executing a new version of our `backup.sh` script, no tar **stderr** message will be displayed.

The last concept to briefly cover in this section is a shell input. Apart of the above **stdout** and **stderr** descriptors bash shell also features input descriptor name **stdin**. Generally, terminal input comes from a keyboard. Any keystroke you type is accepted as **stdin**.

The alternative method is to accept command input from a file using `<` notation. Consider the following example where we first feed cat command from the keyboard and redirecting the output to `file1.txt`. Later, we allow cat command to read the input from `file1.txt` using `<` notation:



Functions

The topic we are going to discuss next is functions. Functions allow a programmer to organize and reuse code, hence increasing the efficiency, execution speed as well as readability of the entire script.

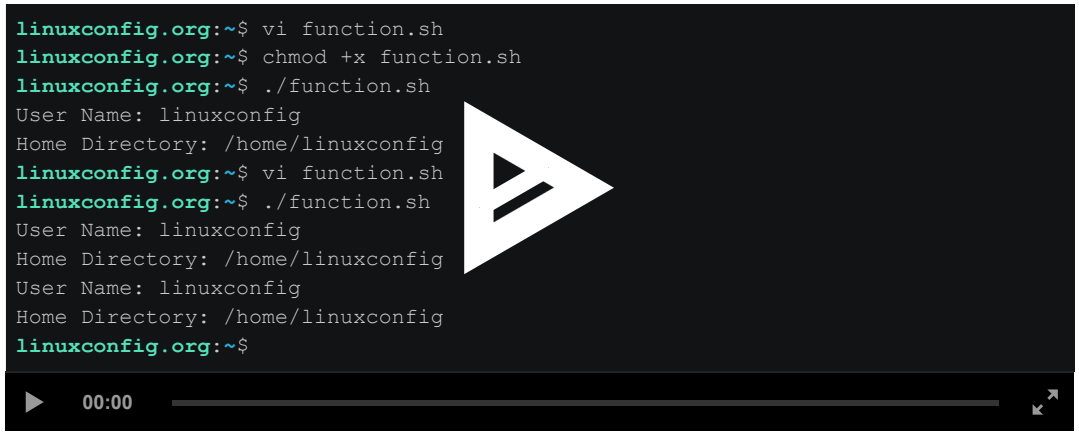
It is possible to avoid using functions and write any script without including a single function in it. However, you are likely to end up with a chunky, inefficient and hard to troubleshoot code.

> **Quick Tip:**
>
> The moment you notice that your script contains two lines of the same code, you may consider to enact a function instead.

You can think of the function as a way to the group number of different commands into a single command. This can be extremely useful if the output or calculation you require consists of multiple commands, and it will be expected multiple times throughout the script execution. Functions are defined by using the function keyword and followed by function body enclosed by curly brackets.

The following video example defines a simple shell function to be used to print user details and will make two function calls, thus printing user details twice upon a script execution.

The function name is `user_details`, and function body enclosed inside curly brackets consists of the group of two `echo` commands. Every time a function call is made by using the function name, both `echo` commands within our function definition are executed. It is important to point out that the function definition must precede function call, otherwise the script will return `function not found` error:



As illustrated by the above video example the `user_details` function grouped multiple commands in a single new command `user_details`.

The preceding video example also introduced yet another technique when writing scripts or any program for that matter, the technique called indentation. The `echo` commands within the `user_details` function definition were deliberately shifted one TAB right which makes our code more readable, easier to troubleshot.

With indentation, it is much clearer to see that both `echo` commands below to `user_details` function definition. There is no general convention on how

to indent bash script thus it is up to each individual to choose its own way to indent. Our example used TAB. However, it is perfectly fine to instead a single TAB use 4 spaces, etc.

Having a basic understanding of bash scripting functions up our sleeve, let's add a new feature to our existing backup.sh script. We are going to program two new functions to report a number of directories and files to be included as part of the output compressed the backup file.

```
	#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

# The function total_files reports a total number of files for a given directory.
function total_files {
	find $1 -type f | wc -l
}

# The function total_directories reports a total number of directories
# for a given directory.
function total_directories {
	find $1 -type d | wc -l
}

tar -czf $output $input 2> /dev/null

echo -n "Files to be included:"
total_files $input
echo -n "Directories to be included:"
total_directories $input

echo "Backup of $input completed!"

echo "Details about the output backup file:"
ls -l $output
```

After reviewing the above backup.sh script, you will notice the following changes to the code:

- we have defined a new function called `total_files`. The function utilized the `find` and `wc` commands to determine the number of files located within a directory supplied to it during the function call
- we have defined a new function called `total_directories`. Same as the above `total_files` function it utilized the `find` and `wc` commands however it reports a number of directories within a directory supplied to it during the function call

> **Quick Tip:**
>
> Read manual pages, if you wish to learn more about `find`, `wc` and `echo` command's options used by our `backup.sh` bash script. Example: `$ man find`

Once you update your script to include new functions, the execution of the script will provide a similar output to the one below:

```
$ ./backup.sh
Files to be included:19
Directories to be inlcuded:2
Backup of /home/linuxconfig completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 5520 Aug 16 11:01 /tmp/linuxcor
```

Numeric and String Comparisons

In this section, we are going to learn some basics of numeric and string bash shell comparisons. Using comparisons, we can compare strings (words,

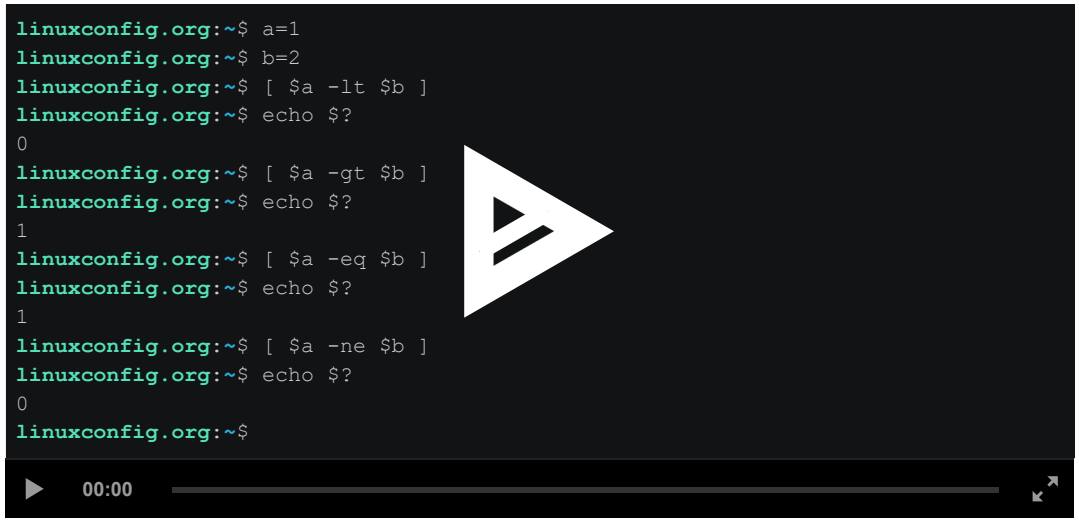
sentences) or integer numbers whether raw or as variables. The following table lists rudimentary comparison operators for both numbers and strings:

Bash Shell Numeric and String Comparisons

Description	Numeric Comparison	String Comparison
less than	-lt	<
greater than	-gt	>
equal	-eq	=
not equal	-ne	!=
less or equal	-le	N/A
greater or equal	-ge	N/A
Shell comparison example:	*[100 -eq 50]; echo $?*	*["GNU" = "UNIX"]; echo $?*

After reviewing the above table, let's say, we would like to compare numeric values like two integers `1` and `2`. The following video example will first define two variables `$a` and `$b` to hold our integer values.

Next, we use square brackets and numeric comparison operators to perform the actual evaluation. Using `echo $?` command, we check for a return value of the previously executed evaluation. There or two possible outcomes for every evaluation, true or false. If the return value is equal to `0`, then the comparison evaluation is true. However, if the return value is equal to `1`, the evaluation resulted as false.



Using string comparison operators we can also compare strings in the same manner as when comparing numeric values. Consider the following example:

If we were to translate the above knowledge to a simple bash shell script, the script would look as shown below. Using string comparison operator `=` we compare two distinct strings to see whether they are equal.

Similarly, we compare two integers using the numeric comparison operator to determine if they are equal in value. Remember, `0` signals true, while `1` indicates false:

```
#!/bin/bash
```

```
string_a="UNIX"
string_b="GNU"

echo "Are $string_a and $string_b strings equal?"
[ $string_a = $string_b ]
echo $?

num_a=100
num_b=100

echo "Is $num_a equal to $num_b ?"
[ $num_a -eq $num_b ]
echo $?
```

Save the above script as eg. `comparison.sh` file, make it executable and execute:

```
$ chmod +x compare.sh
$ ./compare.sh
Are UNIX and GNU strings equal?
1
Is 100 equal to 100 ?
0
```

Quick Tip:

Comparing strings with integers using numeric comparison operators will result in the error: `integer expression expected`. When comparing values, you may want to use `echo` command first to confirm that your variables hold expected values before using them as part of the comparison operation.

Apart from the educational value, the above script does not serve any other purpose. Comparisons operations will make more sense once we learn about conditional statements like if/else. Conditional statements will be covered in

the next chapter, and this is where we put comparison operations to better use.

Conditional Statements

Now, it is time to give our backup script some logic by including few conditional statements. Conditionals allow the programmer to implement decision making within a shell script based on certain conditions or events.

The conditionals we are referring to are of course, `if` , `then` and `else` . For example, we can improve our backup script by implementing a sanity check to compare the number of files and directories within a source directory we intend to backup and the resulting backup file. The pseudocode for this kind of implementation will read as follows:

IF the number of files between the source and destination target is equal THEN print the OK message, ELSE , print ERROR.

Let's start by creating a simple bash script depicting a basic `if/then/else` construct.

```
      #!/bin/bash

num_a=100
num_b=200

if [ $num_a -lt $num_b ]; then
```

```
    echo "$num_a is less than $num_b!"
fi
```

For now the `else` conditional was deliberately left out, we will include it once we understand the logic behind the above script. Save the script as, eg. `if_else.sh` and execute it:

Lines 3 - 4 are used to initialize an integer variables. On **Line 6** we begin an `if` conditional block. We further compare both variables and if the comparison evaluation yields true, then on **Line 7** the `echo` command will inform us, that the value within the variable `$num_a` is less when compared with the variable `$num_b`. **Lines 8** closes our `if` conditional block with a `fi` keyword.

The important observation to make from the script execution is that, in the situation when the variable `$num_a` greater than `$num_b` our script fails to react. This is where the last piece of the puzzle, `else` conditional comes in handy. Update your script by adding else block and execute it:

```
      #!/bin/bash

num_a=400
num_b=200

if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
else
    echo "$num_a is greater than $num_b!"
fi
```

The Line 8 now holds the `else` part of our conditional block. If the comparison evaluation on Line 6 reports false the code below `else` statement, in our case Line 9 is executed.

Exercise:

Can you rewrite the if_else.sh script to reverse the logic of its execution in a way that the else block gets executed if the variable `$num_a` is less than variable `$num_b` ?

Equipped with this basic knowledge about the conditional statements we can now improve our script to perform a sanity check by comparing the difference between the total number of the files before and after the backup command. Here is the new updated `backup.sh` script:

```
      #!/bin/bash

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

function total_files {
        find $1 -type f | wc -l
}

function total_directories {
        find $1 -type d | wc -l
}
```

```
function total_archived_directories {
        tar -tzf $1 | grep  /$ | wc -l
}

function total_archived_files {
        tar -tzf $1 | grep -v /$ | wc -l
}

tar -czf $output $input 2> /dev/null

src_files=$( total_files $input )
src_directories=$( total_directories $input )

arch_files=$( total_archived_files $output )
arch_directories=$( total_archived_directories $output )

echo "Files to be included: $src_files"
echo "Directories to be included: $src_directories"
echo "Files archived: $arch_files"
echo "Directories archived: $arch_directories"

if [ $src_files -eq $arch_files ]; then
        echo "Backup of $input completed!"
        echo "Details about the output backup file:"
        ls -l $output
else
        echo "Backup of $input failed!"
fi
```

There are few additions to the above script. Highlighted are the most important changes.

Lines 15 - 21 are used to define two new functions returning a total number of files and directories included within the resulting compressed backup file. After the backup **Line 23** is executed, on **Lines 25 - 29** we declare new variables to hold the total number of source and destination files and directories.

The variables concerning backed up files are later used on **Lines 36 - 42** as part of our new conditional if/then/else statement returning a message about the successful backup on **Lines 37 - 39** only if the total number of both, source and destination backup files is equal as stated on **Line 36** .

Here is the script execution after applying the above changes:

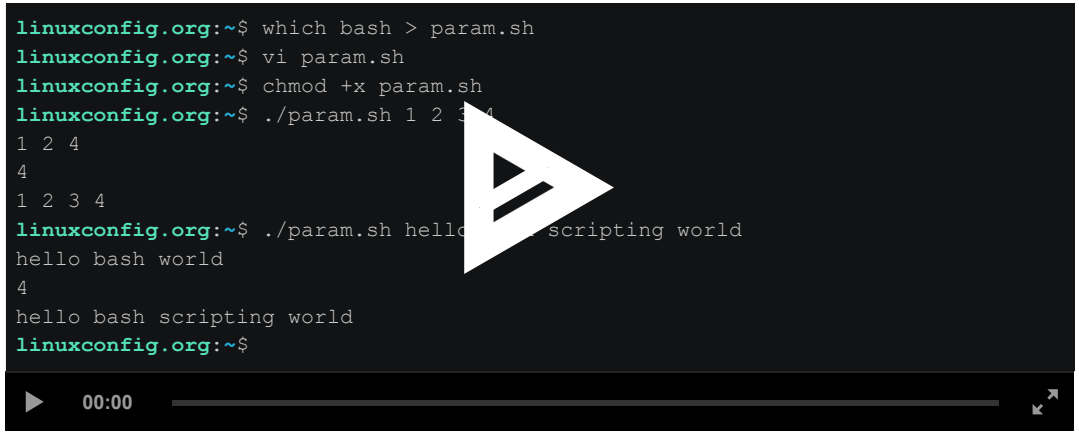
```
$ ./backup.sh
Files to be included: 24
Directories to be included: 4
Files archived: 24
Directories archived: 4
Backup of /home/linuxconfig completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 235569 Sep 12 12:43 /tmp/linuxc
```

Positional Parameters

So far our backup script looks great. We can count the number of files and directories included within the resulting compressed backup file. Furthermore, our script also facilitates a sanity check to confirm that all files have been correctly backed up. The disadvantage is that we are always forced to backup a directory of a current user. It would be great if the script would be flexible enough to allow the system administrator to backup a home directory of any selected system user by merely pointing the script to its home directory.

When using bash positional parameters, this is rather an easy task. Positional parameters are assigned via command line arguments and are accessible within a script as `$1, $2...$N` variables. During the script execution, any additional items supplied after the program name are considered arguments

and are available during the script execution. Consider the following example:



Let's look at the above-used bash example script in more detail:

```
1  #!/bin/bash
2
3  echo $1 $2 $4
4  echo $#
5  echo $*
```

On the **Line 3** we print 1st, 2nd and 4th positional parameters exactly in order as they are supplied during the script's execution. The 3rd parameter is available, but deliberately omitted on this line. Using `$#` on **Line 4**, we are printing the total number of supplied arguments. This is useful when we need to check how many arguments the user provided during the script execution. Lastly, the `$*` on **Line 5**, is used to print all arguments.

Armed with the positional parameters knowledge let's now improve our `backup.sh` script to accept arguments from a command line. What we are looking for here is to let the user decide what directory will be backed up. In case that no argument is submitted by the user during the script's execution, by default the script will backup a current user's home directory. The new script is below:

```
1  #!/bin/bash
2
```

```
# This bash script is used to backup a user's home directory to /tmp/.

if [ -z $1 ]; then
        user=$(whoami)
else
        if [ ! -d "/home/$1" ]; then
                echo "Requested $1 user home directory doesn't exist."
                exit 1
        fi
        user=$1
fi

input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

function total_files {
        find $1 -type f | wc -l
}

function total_directories {
        find $1 -type d | wc -l
}

function total_archived_directories {
        tar -tzf $1 | grep  /$ | wc -l
}

function total_archived_files {
        tar -tzf $1 | grep -v /$ | wc -l
}

tar -czf $output $input 2> /dev/null

src_files=$( total_files $input )
src_directories=$( total_directories $input )

arch_files=$( total_archived_files $output )
arch_directories=$( total_archived_directories $output )
```

```
echo "Files to be included: $src_files"
echo "Directories to be included: $src_directories"
echo "Files archived: $arch_files"
echo "Directories archived: $arch_directories"

if [ $src_files -eq $arch_files ]; then
        echo "Backup of $input completed!"
        echo "Details about the output backup file:"
        ls -l $output
else
        echo "Backup of $input failed!"
fi
```

The above `backup.sh` script update introduces few new bash scripting techniques but rest for the code between **Lines 5 - 13** should be by now self-explanatory. **Line 5** is using a `-z` bash option in combination with conditional if statement to check whether positional parameter `$1` contains any value. `-z` simply returns true if the length of the string which in our case is variable `$1` is zero. If this is the case, we set `$user` variable to a current user's name.

Else on **Line 8** , we check if the requested user's home directory exists by using `-d` bash option. Note the exclamation mark before the -d option. Exclamation mark, in this case, acts as a negator. By default `-d` option returns true if the directory exists, hence our `!` just reverts the logic and on **Line 9** we print an error message. **Line 10** uses `exit` command causing script execution termination. We have also assigned exit value `1` as opposed to `0` meaning that the script exited with an error. If the directory check passes validation, on **Line 12** we assign our `$user` variable to positional parameter `$1` as requested during by the user.

Example of script execution:

```
$ ./backup.sh
Files to be included: 24
Directories to be included: 4
Files archived: 24
```

```
Directories archived: 4
Backup of /home/linuxconfig completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 235709 Sep 14 11:45 /tmp/linuxc

$ ./backup.sh abc123
Requested abc123 user home directory doesn't exist.

$ ./backup.sh damian
Files to be included: 3
Directories to be included: 1
Files archived: 3
Directories archived: 1
Backup of /home/damian completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 2140 Sep 14 11:45 /tmp/damian
```

Quick Tip:

Check bash manual page with `$ man bash` command for more information about `-z`, `-d` and other bash options. Currently, the default storage directory is `/tmp`. Perhaps the script could be more flexible? Can you think of a way to use positional parameter `$2` to let the user to decide on which directory to use to store the resulting backup file?

Bash Loops

So far our backup script functions as expected and its usability has been substantially increased in comparison with the initial code introduced at the beginning of this scripting tutorial. We can now easily backup any user directory by pointing the script to user's home directory using positional parameters during the script's execution.

The trouble only arises when we need to backup multiple user directories on a daily basis. Hence this task will very quickly become tedious and time-

consuming. At this stage, it would be great to have the means to backup any number of selected user home directories with a single backup.sh script execution.

Fortunately, bash has us covered, as this task can be accomplished by use of loops. Loops are looping constructs used to iterate through any given number of tasks until all items in a specified list were completed or predefined conditions were met. There are three basic loop types available to our disposal.

For Loop

For loop is used to iterate through any given code for any number of supplied items in the list. Let's start with a simple for loop example:

The above for loop has used the `echo` command to print all items `1`, `2` and `3` in the list. Using a semicolon allows us to execute for loop on a single command line. If we were to transfer the above for loop into a bash script, the code would look like follows:

```
#!/bin/bash

for i in 1 2 3; do
    echo $i
done
```

The for loop consists of four Shell Reserved Words: for, in, do, done. The above code can therefore also be read as: **FOR** each item **IN** list `1`, `2` and

`3` assign each item temporarily into a variable `i` after which **DO** `echo $i` in order to print the item as STDOUT and keep printing until all items **IN** the list are **DONE** .

Printing numbers is undoubtedly fun but let's try something more meaningful instead. Using command substitution as explained earlier in this tutorial we can create any kind of list to be a part of for loop construct. The following slightly more sophisticated for loop example will count characters of each line for any given file:

Yes, when mastered, the power of GNU Bash knows no limits! Take your time to experiment before moving forward.

Exercise:

Rewrite the above character count for loop to print names of all files and directories inside your current working directory along with the number of characters each file and directory name consists from. The for loop output should look similar to:

```
0_xvz has 5
backup.sh has 9
compare.sh has 10
date.sh has 7
file1.txt has 9
foobar has 6
```

```
function.sh has 11
hello-world.sh has 14
if_else.sh has 10
items.txt has 9
```

While Loop

The next loop construct on our list is while loop. This particular loop acts on a given condition. Meaning, it will keep executing code enclosed withing **DO** and **DONE** while the specified condition is true. Once the specified condition becomes false, the execution will stop. Consider the following example:

```
      #!/bin/bash

counter=0
while [ $counter -lt 3 ]; do
    let counter+=1
    echo $counter
done
```

This particular while loop will keep executing the enclosed code only while the `counter` variable is less than 3. This condition is set on **Line 4** . During each loop iteration, on **Lines 5** the variable `counter` is incremented by one. Once the variable `counter` is equal 3, the condition defined on **Lines 4** becomes false and while loop execution is terminated.

Until Loop

The last loop we are going to cover in this scripting tutorial is until loop. The until loop does the exact opposite of the while loop. Until loop also acts on a preset condition. However, the code enclosed between **DO** and **DONE** is repeatedly executed only until this condition changes from false to true. The execution of until loop is illustrated using the below example:

```
#!/bin/bash

counter=6
until [ $counter -lt 3 ]; do
    let counter-=1
```

```
    echo $counter
done
```

If you understood the above while loop script, the until loop will be somewhat self-explanatory. The script starts with the variable `counter` set to `6`. The condition defined on **Line 4** of this particular until loop is to keep executing the enclosed code until the condition becomes true.

At this stage, we can convert our understanding of loops into something tangible. Our current backup script is currently capable to backup a single directory per execution. It would be nice to have the ability to backup all directories supplied to the script on a command line upon its execution. Review the updated script below which implements such a new feature:

```
      #!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

function backup {

```

```
    if [ -z $1 ]; then
        user=$(whoami)
    else
        if [ ! -d "/home/$1" ]; then
            echo "Requested $1 user home directory doesn't exist."
            exit 1
        fi
        user=$1
    fi

    input=/home/$user
    output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

    function total_files {
        find $1 -type f | wc -l
    }

    function total_directories {
        find $1 -type d | wc -l
    }

    function total_archived_directories {
        tar -tzf $1 | grep  /$ | wc -l
    }

    function total_archived_files {
        tar -tzf $1 | grep -v /$ | wc -l
    }

    tar -czf $output $input 2> /dev/null

    src_files=$( total_files $input )
    src_directories=$( total_directories $input )

    arch_files=$( total_archived_files $output )
    arch_directories=$( total_archived_directories $output )

    echo "########## $user ##########"
    echo "Files to be included: $src_files"
```

```
    echo "Directories to be included: $src_directories"
    echo "Files archived: $arch_files"
    echo "Directories archived: $arch_directories"

    if [ $src_files -eq $arch_files ]; then
        echo "Backup of $input completed!"
        echo "Details about the output backup file:"
        ls -l $output
    else
        echo "Backup of $input failed!"
    fi
}

for directory in $*; do
    backup $directory
done;
```

After reviewing the above script, you may have noticed that new function called `backup` on **Lines 5 - 57** was created. This function includes all of our previously written code. The function definition ends on **Line 57** after which we have implemented a new for loop on **Lines 59 - 51** to execute the newly defined `backup` function for each user directory supplied as an argument. If you recall, the `$*` variable contains all arguments supplied on a command line upon the script execution. Furthermore, a cosmetic change to the code on **Line 44** ensures a better readability of the script's output by separating each directory backup info output block with a hash line. Let's see how it works:

```
$ ./backup.sh linuxconfig damian
########## linuxconfig ##########
Files to be included: 27
Directories to be included: 4
Files archived: 27
Directories archived: 4
Backup of /home/linuxconfig completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 236173 Oct 23 10:22 /tmp/linuxc
########## damian ##########
Files to be included: 3
Directories to be included: 1
Files archived: 3
Directories archived: 1
```

```
Backup of /home/damian completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 2140 Oct 23 10:22 /tmp/damian_h
```

Exercise:

The current script does not check for the existence of user directories prior to the backup function execution. This can lead to unforeseen consequences. Do you think that you would be able to create your own improved copy of the backup script by defining a separate loop to check the existence of all user directories before the backup for loop is reached? You for loop will exit the script's execution if any of the user directories on the supplied list does not exist.

Bash Arithmetics

In the last section of this bash scripting tutorial, we will discuss some basics of bash arithmetics. Arithmetics in bash scripting will add another level of sophistication and flexibility to our scripts as it allows us to calculate numbers even with numeric precision. There are multiple ways on how to accomplish arithmetic operations within your bash scripts. Let's go through some of them using few simple examples.

Arithmetic Expansion

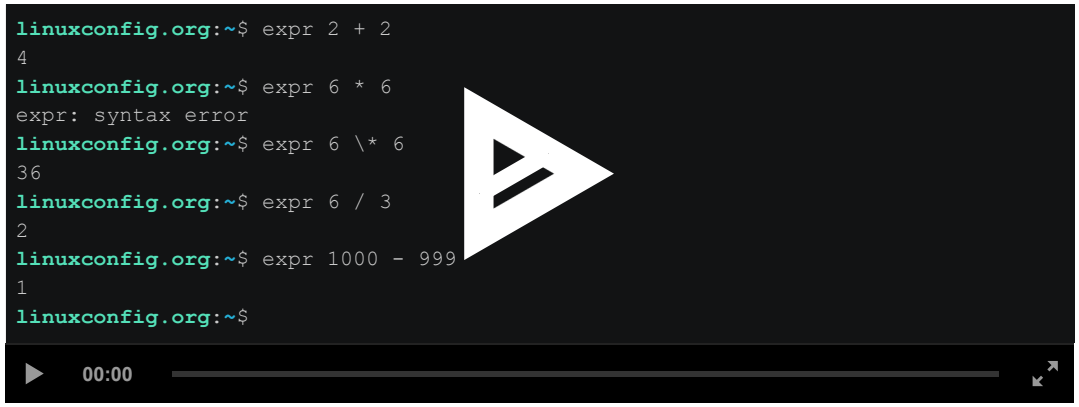
The arithmetic expansion is probably to the most simple method on how to achieve basic calculations. We just enclose any mathematical expression inside double parentheses. Let's perform some simple addition, subtraction, multiplication and division calculations with integers:

Exercise:

Can you use the arithmetic expansion to perform a modulus operation? For example what is the result of modulus operation `99 % 10` ?

expr command

Another alternative to arithmetic expansion is the `expr` command. Using the expr command allows us to perform an arithmetic operation even without enclosing our mathematical expression within brackets or quotes. However, do not forget to escape asterisk multiplication sign to avoid `expr: syntax error`
:



let command

Similarly, as with `expr` command, we can perform bash arithmetic operations with `let` command. `let` command evaluates a mathematical expression and stores its result into a variable. We have alredy encountered the `let` command in one of our previous examples where we have used it to perform integer increment. The following example shows some basic operations using `let` command as well as integer increment and exponent operations like x^3:

bc command

After few minutes of experimentation with the above bash arithmetic methods, you may have noticed that they work perfectly with integer numbers however when it comes to the decimal numbers there is something amiss. To take our bash arithmetic to an entirely different level, we will need to use `bc` command. `bc` command with a proper syntax allows for more than simple integer calculations.

Operational manual of the `bc` command is quite extensive as it spans over more than 500 lines. However, it does not hurt to show some basic operations. The following example will perform a division operation with 2 and 30 decimal numbers and the square root of 50 with 50 decimal numbers. By default, the `bc` command will produce all results as an integer number. Use `scale=x` to instruct the bc command to show real numbers:

Let's put our new bash arithmetic knowledge to work and once again change our backup.sh script to implement a counter of all archived files and directories for all users:

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.
function backup {
```

```
    if [ -z $1 ]; then
        user=$(whoami)
    else
        if [ ! -d "/home/$1" ]; then
                echo "Requested $1 user home directory doesn't exist."
                exit 1
        fi
        user=$1
    fi

    input=/home/$user
    output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

    function total_files {
        find $1 -type f | wc -l
    }

    function total_directories {
        find $1 -type d | wc -l
    }

    function total_archived_directories {
        tar -tzf $1 | grep  /$ | wc -l
    }

    function total_archived_files {
        tar -tzf $1 | grep -v /$ | wc -l
    }

    tar -czf $output $input 2> /dev/null

    src_files=$( total_files $input )
    src_directories=$( total_directories $input )

    arch_files=$( total_archived_files $output )
    arch_directories=$( total_archived_directories $output )

    echo "########## $user ##########"
```

```
    echo "Files to be included: $src_files"
    echo "Directories to be included: $src_directories"
    echo "Files archived: $arch_files"
    echo "Directories archived: $arch_directories"

    if [ $src_files -eq $arch_files ]; then
        echo "Backup of $input completed!"
        echo "Details about the output backup file:"
        ls -l $output
    else
        echo "Backup of $input failed!"
    fi
}

for directory in $*; do
    backup $directory
    let all=$all+$arch_files+$arch_directories
done;
    echo "TOTAL FILES AND DIRECTORIES: $all"
```

On **Line 60** we have used addition to add all archived files using `let` command to a resulting variable `all`. Each for loop iteration adds new count for every additional user. The result is then printed using `echo` command on **Line 62** .

Example script execution:

```
$ ./backup.sh linuxconfig damian
########## linuxconfig ##########
Files to be included: 27
Directories to be included: 6
Files archived: 27
Directories archived: 6
Backup of /home/linuxconfig completed!
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 237004 Dec 27 11:23 /tmp/linuxc
########## damian ##########
Files to be included: 3
Directories to be included: 1
Files archived: 3
Directories archived: 1
Backup of /home/damian completed!
```

```
Details about the output backup file:
-rw-r--r-- 1 linuxconfig linuxconfig 2139 Dec 27 11:23 /tmp/damian_h
TOTAL FILES AND DIRECTORIES: 37
```

Exercise:

Experiment with the backup.sh script. The script is far from being perfect, add new features or fix current features. Do not be afraid to break things as that is perfectly normal. Troubleshooting and fixing code is perhaps the best booster for you to enhance your understanding of bash scripting and to improve your ability to script beyond what has been discussed in this tutorial.

Conclusion

There is more to bash shell scripting than covered in this tutorial. However, before you move on, make sure that you are comfortable with topics discussed here. Apart from googling, there are myriad of other resources available online to help you out if you get stuck. The most prominent and highly recommended of them all is GNU's Bash Reference Manual.

Prev Next

latest news, jobs, career advice and tutorials.

FORUM or simply use comments below.

MORE ON LINUXCONFIG.ORG:

- Multi-threaded Bash scripting & process management at the command line
- How to Debug Bash Scripts
- How to create a selection menu using the select statement in Bash shell
- Advanced Bash regex with examples

YOU MAY ALSO BE INTERESTED IN:

Comments and Discussions

NEWSLETTER

Subscribe to **Linux Career Newsletter** to receive latest news, jobs, career advice and featured configuration tutorials.

WRITE FOR US

LinuxConfig is looking for a technical writer(s) geared towards GNU/Linux and FLOSS technologies. Your articles will feature various GNU/Linux configuration tutorials and FLOSS technologies

FEATURED LINUX TUTORIALS

- How To enable the EPEL Repository on RHEL 8 / CentOS 8 Linux
- Bash scripting Tutorial
- How to install VMware Tools on

LATEST ARTICLES

- How to use zip on Linux
- Useful Bash command line tips and tricks examples - Part 1
- Find a directory in Linux

Full Name

Email

GDPR permission: I give my consent to be in touch with me via email using the information I have provided in this form for the purpose of news and updates.

Subscribe to Newsletter

used in combination with GNU/Linux operating system.

When writing your articles you will be expected to be able to keep up with a technological advancement regarding the above mentioned technical area of expertise. You will work independently and be able to produce at minimum 2 technical articles a month.

APPLY NOW

CONTACT

web (at) linuxconfig (dot) org

- RHEL 8 / CentOS 8
- Howto mount USB drive in Linux
- How to install the NVIDIA drivers on Ubuntu 18.04 Bionic Beaver Linux
- How to update Kali Linux
- Ubuntu 20.04 Download
- How To Upgrade Ubuntu To 20.04 LTS Focal Fossa
- How to install node.js on RHEL 8 / CentOS 8 Linux
- How to check CentOS version
- How to Parse Data From JSON Into Python
- Check what Debian version you are running on your Linux system
- Bash Scripting Tutorial for Beginners
- Ubuntu 20.04 Guide
- How to stop/start firewall on RHEL 8 / CentOS 8
- Install gnome on RHEL 8 / CentOS 8
- Advanced Bash regex with examples
- Bash regexps for beginners with examples
- GNU/Linux General Troubleshooting Guide for Beginners
- How to create compressed encrypted archives with tar and gpg
- Best Multimedia Linux distributions
- How to Fix Grub error: no such partition Grub Rescue
- How to create incremental backups using rsync on Linux
- Introduction to the Systemd journal
- How to trace system calls made by a process with strace on Linux
- Formatting SD or USB disk under Linux
- apt vs apt-get - Advanced Package Tool
- Manjaro Linux Beginner's Guide

- Linux Download
- How To Upgrade from Ubuntu 18.04 and 19.10 To Ubuntu 20.04 LTS Focal Fossa
- Enable SSH root login on Debian Linux Server