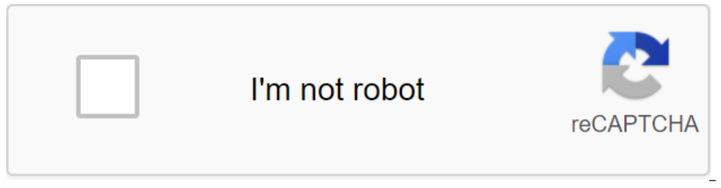

Continue

Bash shell script tutorial pdf

Bash Bash is a team language translator. It is widely available on various operating systems and is the default team translator on most GNU/Linux systems. The name is an acronym for Born-Again SHell. Shell Shell is a macroprocessor that allows you to carry out interactive or inactive commands. The script script allows you to perform automatic commands that would otherwise be performed online one by one. Don't despair if you don't understand any of the above bash Shell Scripting definitions. It's perfectly normal, in fact, which is why you're reading this Bash Script tutorial. In case you didn't know, Bash Scripting is a must-have skill for any Linux work administration system, even if it can't be indirectly requested by your employer. What is Shell, most likely your currently sitting in front of the computer, have the terminal window open and wondering: What do I do with this thing? Well, the terminal window in front of you contains a shell, and the shell allows you to use commands to interact with your computer, hence receive or store data, process information and various other simple or even extremely complex tasks. Try it now! Use the keyboard and type in some commands such as date, cal, pwd or ls followed by the ENTER key. What you just did was that with commands and shell you interacted with your computer to get the current date and time (date), looked at the calendar (cal), checked the location of the current work directory (pwd) and got a list of all the files and directories located within (ls). What is the scenario now, imagine that executing all of the above commands is your daily task. Every day you must complete all of the above commands and keep the information you are watching. Soon enough, it will be an extremely tedious task, doomed to failure. So the obvious concept is to think about some way of executing all these commands together. Here the script becomes your salvation. To see what the script means, use the shell in conjunction with your favorite text editor, for example. vi to create a new file called task.sh containing all of the above commands, each of which is on a separate line. Once ready, make your new file run using the chmod command with the option x. Finally, execute a new script by fixing its name with ./. As you can see, with the help of scripts, any interaction of the shell can be automated and scripted. In addition, you can now automatically execute our new shell script task.sh daily at any given time using a cron based on the work schedule and store output to file every time it runs. However, it's a fairy tale for another day, for now let's just focus on the task ahead. What is Bash So far we have reviewed the shells and scenarios. What about Bash? Where does the bash fit in? As already Bash is the default translator on many GNU/Linux systems, so we used it without even realizing it. That's why our previous shell script works even without us defining the bash as an interpreter. To see what your default translator perform echo command $SHELL: $ Echo $SHELL/bin/bash there are various other shell translators available such as The Korn Shell, C Shell and more. For this reason, it is good practice to identify the shell translator used explicitly to interpret the contents of the script. To identify your script's translator as Bash, first find the full path to its run binary file, with which the team, secure it with shebang! and insert it as the first line of the script. There are various other methods for determining the translator's shell, but this is a solid beginning. From now on, all of our scripts will include the definition of a shell translator No!/bin/bash. SUBSCRIBE TO NEWSLETTER Subscribe to Linux Career NEWSLETTER and receive the latest Linux news, jobs, career tips and tutorials. File names and permissions next, let's briefly discuss the resolution of files and file names. You may have already noticed that the shell script must be executed using chmod q FILENAME to run the shell script. By default, any newly created files are not executed regardless of the file extension suffix. In fact, extending the file to GNU/Linux systems basically makes no sense, except that when running a command ls to list all the files and directories immediately it is clear that a file with an extension .sh plausibly script shell and file with .jpg is likely to be a loss-making compressed image. A file command can be used in GNU/Linux systems to identify the type of file. As you can see in the example below, file extension doesn't matter, and the shell translator, in this case, has more weight. Thus, the name of the shell script 0_xyz is quite valid, but this should be avoided if possible. Follow the script, let's talk about an alternative way to run Bash scripts. In a very simplistic view, the bash script is nothing more than a text file containing instructions that must be executed from top to bottom. How the instructions are interpreted depends on a particular shebang or how the script is executed. Consider the following video example: Another way to perform bash scripts is to call the translator's bash for a clear example. The $ bash date.sh, hence running the script without having to do the shell scenario performed and without announcing the shebang directly in the script shell. Calling Bash performed binary explicitly, the contents of our file date.sh downloaded and interpreted as Bash Shell Script. Relative Vs. Absolute Path Finally Before We will program our first official shell script bash, let's briefly discuss the navigation of the shell and the difference between the relative and the absolute way of the file. Probably the analogy for explaining the relative and absolute path of the file is to visualize the GNU/Linux file system as a multi-storey building. The root directory (the building's front door), specified /, provides entry into the entire file system (building), hence providing access to all directories (levels/rooms) and files (people). To upgrade to number 1 at level 3, we first have to enter the main door/, then make our way to level 3 level3/ and from there enter room1. So the absolute path to this particular room in the building is /level3/room1. From here, if we want to visit room2 also at level3, we first have to leave our current location, which is room1, entering .. / and then include the name of room2. We took the relative path to number2, which in this case .. /room2. We were already on level 3 so there was no need to leave the entire building and walk the absolute way through the main entrance/level3/room2. Fortunately, GNU/Linux has a simple compass tool that will help you navigate the file system in the form of a pwd command. This command will always print the current location when performing. The following example will use the cd and pwd command to navigate the GNU/Linux file system using absolute and relative paths. Navigating the GNU/Linux file system is a simple and yet for many a very confusing topic. Check out the GNU/Linux file navigation before moving on to the following sections of this tutorial. Hello World Bash Shell Script Now, it's time to write our first, most basic script shell Bash. The whole purpose of this script is nothing more than printing Hello World using the Echo command to pull the terminal out. Using any text editor to create a new file called hello-world.sh containing the code below: !/bin/bash Echo: Hello World After How ready, make your script executed with thechmod command and execute it using a relative path./hello-world.sh: $ chmod x hello-world.sh is linuxconfig.org: hello-world.sh . . . . . hello-world.sh . . . . . He uses what command to print the full path to the translator Bash. This output is simultaneously redirected using the redirect mark when creating a new file hello-world.sh at the same time. A simple Bash Shell backup scenario, let's discuss the execution of the command line and how the GNU/Linux commands fit into the scripting process in more detail Any command that can be successfully executed directly through the bash shell terminal can be in the same shape as part of the bash shell script. In fact, there is no difference between executing a command directly through a terminal or in a shell script, except that the shell script offers an inactive multiple commands as a single process. In addition, most teams accept so-called options and arguments. Command settings are used to change the team's behavior for results and prefixed.. Arguments can specify the purpose of a command, such as file, catalog, text, and more. Each team comes with a manual page that can be used to study its function, as well as what options and arguments each particular team accepts. Use the man command to display the manual page of any desired command. For example, to display a manual page for the ls command, perform man ls. To get out of the manual page click q key. An example of the ls command shows the basic use of command line options and arguments. While our first Hello World shell script requires a clear understanding of file creation, editing, and script execution, its convenience can be clearly questioned. The following example offers a more practical application because it can be used to back up our home directory. To create a backup scenario, line 3we will use a resin command with a variety of -czf variants to create a compressed resin ball throughout the user's home/home/linuxconfig/. Insert the following code into a new file called backup.sh, make the script executed and run it: !/bin/bash tar-czf/tmp/myhome_directory.tar.gz/home/linuxconfig Variables Variables Variables are the essence of programming. Variables allow the programmer to store, modify, and reuse the data throughout the script. Create a new welcome.sh script with the following content: !/bin/bash greetingDo welcome user'$'$(date -%A) echo $greeting ago $user! Today $day, which is the best day of the week! Echo Your Bash Shell version: $BASH -VERSION. Enjoy yourselves! By now, you must have all the necessary skills needed to create a new script, making it executed and working on a command line. After launching the above welcome.sh, you'll see an outlet similar to the one below: $./welcome.sh Welcome back linuxconfig! Today is Wednesday, which is the best day of the week! Your Bash shell version: 4.4.12 (1)-release. Enjoy! Let's look at the script more closely. First, we announced a variable greeting and assigned the meaning of the line Welcome to it. The next variable user contains the username that the shell session starts. This is done with a method called team replacement. This means that the whoami team will be directly assigned to the user variable. The same goes for our next variable day, which contains the name of today, produced by the date of the %A. The second part of the script uses an echo command to print the message, replacing the variable names that are now stamped on the $, with their Values. In case you're wondering about the latest variable used $BASH THE VERSION know that this so-called internal variable is defined as part of your shell. Variables can also be used directly on the terminal command line. The following example announces variables A and B with integer data. Using an echo command, we can print out their values or or perform an arithmetic operation as shown in the following example: Now that we have a bash variable introduction behind us we can update our backup script to obtain a more meaningful file name output by including the date and time when the backup on our home directory was actually performed. In addition, the script will no longer communicate with a specific user. From now on, our backup.sh Bash script can be launched by any user, while backing up the user's correct home directory: !/bin/bash This bash script is used to back up the 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 $input completed! Details of the back-up inference file: ls-l $output you may have already noticed that the aforementioned script introduces two new concepts of the bash scenario. First, our backup.sh script contains a line of comments. Each line, starting with the q sign, except shebang, will not be interpreted by Bash and will serve only as an internal note of the programmer. Second, the script uses a new script shell trick $ (parameter) is called an extension option. In our case, braces are required because our variable $user is accompanied by symbols that are not part of its variable name. Below is the release of our recently revised backup scenario: $./backup.sh pitches: Removing the lead '/' from the names of members Backup/home/linuxconfig completed! Details of the backup file: -rw-r-r-r-r-- 1 linuxconfig 8778 Yul 27 12:30 /tmp/linuxconfig_home_2017-07-27-123043.tar.gz SUBSCRIBE TO NEWSLETTERSubribe to Linux Career NEWSLETTER and receive the latest Linux Career NEWS, career tips, career tips and tutorials. Redirecting input, output, and errors is usually done on the GNU/Linux command line, either exiting, requiring input, or causing an error message. This is a fundamental concept for shell scripts, as well as for working with the GNU/Linux command line in general. Each time you complete a command, there may be three possible outcomes. The first scenario is that the team will produce the expected output, secondly, the team will generate the error, and finally your team can not produce any output at all: What we are most interested in here is the output of both ls-l foobar teams. Both commands have produced an exit that is displayed on your terminal by default. However, both exits are fundamentally different. The first team tries to list a non-existent file foobar, which in turn produces a standard error output (stderr). Once created by a touch command, the second performance of the ls command produces a standard output (stdout). The difference between thick and thick output is an important concept because it allows us to threaten, that is, redirect each output separately. Notation is used to redirect thick to the file, while 2 notation is used for stderr and q is used to redirect both thick and stderr. The cat command is used to display the contents of any file. Consider the following example: Repeat the above video several times and make sure you understand the concept of redirecting shown. Let's go back to our backup.sh scenario. When performing our backup scenario, you may have noticed an additional display of message on the resin command: resin: Removing the lead '/' from the names of members Despite the informative nature of the message, it is sent to the stderr handle. In a nutshell, the message tells us that the absolute path has been removed so the extraction of the compressed file does not repurpose any existing files. Now that we have a basic understanding of output redirection, we can eliminate this unwanted stderr message by redirecting it from 2 notation to /dev/null. Imagine /dev/null as a data sink that discards any data redirected to it. For more information, run the person zero. Below is our new version of backup.sh, including stderr resin redirection: !/bin/bash This script is used to back up the 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/dev/null echo of

$input completed! Details of the back-up note: ls-l $output After a new version of our backup.sh script, the tar stderr message will not be displayed. The last concept that can be summarized in this section is the introduction of the shell. Aside from the aforementioned thick and thick descriptors the bash shell also has an input handle name stdin. Typically, the terminal input comes from the keyboard. Any keystroke you type in is accepted as a stdin. An alternative method is to take command input from the file using notation. Consider the following example where we first feed the cat command with the keyboard and redirect the output to file1.txt. Later, we allow the cat team to read input from file1.txt using notation: The function of the topic that we are going to discuss further, is the features. The features allow the programmer to organize and reuse the code, which increases the efficiency, speed of execution, and readability of the entire script. You can avoid using features and write any script without including any features. However, you will probably end up with chunky, inefficient and difficult troubleshooting code. You can think of the feature as a way to group different teams into one team. This can be extremely useful if the conclusion or calculation that you require consists of several commands, and this will be expected several times on the entire script. Features are defined by the keyword function, and then the function of the body is attached to the curly braces. The following video example identifies a simple shell function that will be used to print user data and make two functional calls, thus printing user data twice on the on Perform. The function name is user_details, and the body function encased inside the curly braces consists of a group of two echo commands. Each time a function is called using the function name, both echo commands are performed in determining our function. It's important to note that the function definition should precede the function call, otherwise the script will return a feature that didn't find an error: As the video above shows, the function user_details grouped by multiple commands in one new command user_details. A previous video example also introduced another method when writing scripts or any program, for that matter, a method called indentation. Echo commands in user_details functions were intentionally shifted to one right TAB, making our code more readable, easier to troubleshoot. With the indentation, it's much clearer to see that both echo commands below user_details defining function. There is no general convention on how to indent the bash scenario, so it is up to each person to choose their own path to indent. Tab is used in our example. However, it's fine, instead of one TAB using 4 spaces etc. Having a basic understanding of Bash script features up our sleeve, let's add a new feature for our existing backup.sh script. We're going to program two new features to report a number of directories and files that will be included as part of the release of a compressed backup file. In!/bin/bash - This script is used to back up the user's home directory on /tmp/. user $(whoami) input/home/$user output/tmp/$user (date :%Y-%m-%m-% d_%H%M%S).tar.gz The function of total_files reports the total number of files for this catalog. total_files and find $1-type f-wc-l - a feature total_directories reports the total number of directories for this catalog. feature 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 Catalogs to be included: Echo total_directories $input Backup $input completed! Echo Details about the release of the backup file: ls -l $output After reviewing the above backup.sh scenario, you'll notice the following changes in the code: we've identified a new feature called total_files. The feature used find and wc commands to determine the number of files in the directory provided to it during the function call, we identified a new feature called total_directories. Just like the above total_files feature he used to find and WC command however it reports a number of directories in the catalog comes to it during the call function After the script update to include the new features, the execution of the script will provide a similar output to the one below: $./backup.sh Files that are included:19 Catalogs to be inlcuded:2 inlcuded:2 /home / linuxconfig completed! Details of the backup file inference: -rw-r-r-r-r-r-- 1 linuxconfig 5520 August 16 11:01 /tmp/linuxconfig_home_2017-08-16-110121.tar.gz Numeric and String ComparisonsIn this section we are going to learn some of the basics of the number and string bash comparison. Using comparisons, we can compare lines (words, sentences) or price numbers, whether raw or variable. The following table lists rudimentary comparison operators for both numbers and lines: Bash Shell Numeric and String Comparisons Description Numeric Comparison String Comparison Shell Comparison Example: Echo $? GNU - UNIX; Echo $? smaller than -lt - more than -gt - equal -eq - not equal -ne!' smaller or equal -le N/A more or equal -ge N/A After considering the above table, say we would like to compare numerical values such as two integrators 1 and 2. The following video example will first identify two variables $a and $b keep our overall values. Next, we use brackets and numerical comparisons of operators to perform the actual assessment. Using The Echo $? commands, we check for the return value of the previously completed assessment. There's or two possible outcomes for each assessment, true or false. If the return value is 0, the comparison score is correct. However, if the return value is 1, the score leads to false. Using line comparison operators, we can also compare lines in the same way as when comparing numerical values. Consider the following example: If we translated the above knowledge into a simple bash shell script, the script would look like the one below. Using a string comparison operator, we compare two separate lines to see if they are equal. Similarly, we compare the two centners using a numerical comparison operator to determine whether they are equal in value. Remember that 0 signals are correct, while 1 points to the false: !/bin/bash string_aUNIX string_bGNU echoes Are $string and $string-B lines equal? $string th $string and Echo $? num_a 100 num_b 100 echoes $num equal $numb? - $num -eq $num th b and echo $? Save the above scenario, for example. comparison.sh file, make it executable and perform: $chmod x x compare.sh $./compare.sh Are UNIX and GNU lines equal? 1 Is 100 equal to 100? 0 In addition to educational value, the above script serves no other purpose. Comparison operations will make more sense once we learn about conditional applications such as if/else. Conditional applications will be covered in the next chapter, and this is where we put comparison operations for better use. SUBSCRIBE TO NEWSLETTER Subscribe to Linux Career NEWSLETTER and receive the latest Linux news, jobs, career tips and tutorials. Conditional statements now, it is time to give some logic to the backup scenario, including a few conditional statements. Conditional allow programmer programmer making decisions in the shell script based on certain conditions or events. Conditions that we mean, of course, if, then and again. For example, we can improve our backup script by implementing a sanity check to compare the number of files and directories in the source directory that we intend to back up and the resulting backup file. The pseudocode for this kind of implementation will be read as follows: if the number of files between the source code and the destination goal is equal, then print OK message, ELSE, seal ERROR. Let's start by creating a simple script bash featuring a major if/then/else build. 100 num_a 200 num_b if $num -lt $num-b; then the echo $num-a less than $num'b! Fi At this point still conditional was intentionally left, we will include it as soon as we understand the logic above the script. Save the script, for example. if_else.sh and run it: Lines 3 - 4 are used to initiate integer variables. On Line 6 we start if the conditional block. Next, we compare the two variables, and if the comparison score is true, the Echo command will tell us on line 7 that the value within the variable $num is smaller than the variable $num'b. Line 8 closes our if conditional block with a keyword fi. An important point to make of running a script is that in a situation where the variable $num more than $numb, our script does not respond. This is where the last piece of the puzzle, otherwise conditional will come in handy. Update the script by adding another block and execute it: !/bin/bash num_a400 num_b x 200 if $num -lt $num'b; then the echo $num'a less than $num'b! More echoes $num'a more than $num'b! Fi Line 8 now has another part of our conditional block. If the comparison score on line 6 says a false code below another statement, we'll have a line 9. Equipped with this basic knowledge of conditional statements, we can now improve our script to verify sanity by comparing the difference between the total number of files before and after the backup command. Here's a new updated script backup.sh: !/bin/bash user/whoami-input/home/$user output'/tmp/$-100/user'home $$(date:%Y-%-%-%d_%H%M%S). function tar.gz total_files - find the function $1-type f-l - function total_directories - find the function $total_archived_files1-type d wc-l - total_archived_directories - resin -tzf $1 resin -tzf $1 r 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 Echo Catalog files, to be included: $src-catalogues Echo Files archived: $arch files Echo Catalogs are archived: $arch-catalogues - $src -eq $arch files; Then the Backup of $input Echo echo is complete! Echo Details about the release of the backup file: ls-l $output still echo Backup of $input failed! Fi there are a few additions to the above scenario. The most important changes have been highlighted. Lines 15 - 21 are used to identify two new features that return the total number of files and directories included in the resulting compressed backup file. After running the backup line 23 on lines 25 to 29, we announce new variables to complete the total number of files and directories of sources and assignments. Backup variables are later used on lines 36 to 42 as part of our new if/then/else conditional statement returning a successful backup message on lines 37 to 39only if the total number of source and destination backup files is the same as indicated on line 36. Here is the execution of the script after the application of the above changes: $./backup.sh Files to be included: 24 Directories to be included: 4 Files archived: 24 Directories archived: 4 Backup/home/linuxconfig completed! Details of the backup file: -rw-r-r-r-r-- 1 linuxconfig 235569 Sep 12 12:43/tmp/linuxconfig_home_2017-09-12-124319.tar.gz Positional Settings So far our backup scenario looks great. We can count the number of files and directories included in the compressed backup file. In addition, our script also makes it easier to check sanity to confirm that all the files were properly backed up. The downside is that we always have to back up the current user's catalog. It would be great if the script was flexible enough to allow the system administrator to back up any selected system user's home directory by simply pointing the script to their home directory. When you use bash positional, this is a fairly simple task. Positional parameters are assigned using command line arguments and are available in the script as $1, $2...$N variables. During the script, any additional items that come after the program's name are considered arguments and are available during the script. Consider the following example: Let's look at the above-mentioned example of Bash's script in more detail: !/bin/bash echo $1 $2 $4 Echo $ Echo $ On Line 3 we print the 1st, 2nd and 4th positional parameters exactly as they come during the execution of the script. The third option is available, but is intentionally omitted on this line. Using line 4, we print the total number of arguments provided. This is useful when we need to check how many arguments the user has provided while running the script. Finally, to print all the arguments, $E is used on the 5. Armed with knowledge of positional parameters, let's now backup.sh our script to take arguments from the command line. What we are looking for here is to allow the user to decide which directory directory Backup. If the user does not present any arguments while running the script, the default script will be to back up the current user's home directory. The new script is below: !/bin/bash This script is used to back up the user's home directory to /tmp/. If - z $1; Then the user '$(whoami) still if th ! -d /home/$1 ; Then the Echo Requested $1 custom home directory does not exist. Exit 1 fi user'$1 fi input'/home/$user output'/tmp/$-user'home'$$(date:%Y-%%-%-%-d_%H%M%S).tar.gz function total_files - find the function $1-type f-wc-l - function total_directories - find the function $total_archived_files1-type d wc wc l - total_archived_directories - resin -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) which will be included: $srcfayles Echo Catalogs, that will be included: $src-catalogues Echo Files archived: $arch files Echo Catalogs are archived: $arch catalogs if $src $arch then Echo Backup $input completed! Echo Details about the backup inference file: ls-l $output still echo backup of $input failed! Fi Above backup.sh The Update Script introduces several new bash script methods, but the rest for the code between the lines 5 - 13 should be by now self-evident. Line 5 uses the -z bash option in combination with a conditional if statement to check whether the $1 positional option contains any value. -z just returns true if the length of the string, which in our case is variable $1 is zero. If so, we $user the variable in the current user's name. In another case, on line 8, we check whether the home directory of the requested user exists using the -d bash option. Note the exclamation point in front of the option -d. The exclamation point, in this case, acts as a denier. The default -D option returns correctly if the directory exists, hence ours ! just returns the logic and on line 9 we print an error message. Line 10 uses an exit command that causes the script to stop running. We also assigned a release value of 1 as opposed to 0, which means that the script was wrong. If the directory check is checked, on line 12 we assign $user variable positional option $1, as suggested by the user. Sample execution scenario: $./backup.sh Files to be included: 24 Directories to be included: 4 Files archived: 24 Directories archived: 4 Backup/home/linuxconfig completed! Details of the backup file: -rw-r-r-r-r--r-- 1 linuxconfig 235709 Sep 14 /tmp/linuxconfig_home_2017-09-14-114521.tar.gz$./backup.sh abc123 Requested abc123 user home catalog does not exist. $./backup.sh Damian Files to be included: 3 Directories to be included: 1 Files Archive: 3 Catalogues Archive: 1 1 /Home / Damian is finished! Details of the backup file: -rw-r-r-r-r-r-- 1 linuxconfig linuxconfig 2140 Sep 14 11:45/tmp/damian_home_2017-09-14-114534.tar.gz Bash Loops as expected, and its convenience was significantly increased compared to the original code presented at the beginning of this scenario. We can now easily back up any user directory by pointing the script to the user's home directory using positional settings while running the script. The problem only occurs when we need to back up multiple user directories on a daily basis. Therefore, this task very quickly will become tedious and time-consuming. At this point, it would be great to have the tools to back up any number of selected home directories of users with one backup.sh execute the script. Fortunately, the bash has us covered, since this task can be accomplished using loops. Loops are cyclical designs used to heerate through any specific number of tasks until all the elements in the listed are met or predetermined conditions are met. There are three main cycle types available for our removal. For Loop For, the loop is used to be iterationd through any code for any number of items supplied in the list. Let's start with a simple example: the above for the loop used the echo command to print all the elements 1, 2 and 3 in the list. Using a short-colon allows you to cycle on a single command line. If we were to pass the above for the loop in the bash scenario, the code would look like this: !/bin/bash for i in 1 2 3; Do echo $i made for a cycle comprised of four Shell Reserved Words: for, in, do, do. Thus, the above code can also be read as: FOReach INlist 1, 2 and 3 will temporarily assign each item to the i variable, after which DOecho $i in order to print the item as STDOUT and continue to print until all elements of the INthe are made. Printing numbers is undoubtedly fun, but let's try something more meaningful rather than. Using a team replacement, as explained earlier in this tutorial, we can create any list that is part of the cycle building. The next slightly more complex example for the cycle will be to take the characters of each line for any file: Yes, when mastering, the power of GNU Bash knows no bounds! Not the time to experiment before moving forward. While the loop is the next cycle design in our list while looping. This cycle operates in this state. Meaning, it will keep the execution code attached with DOand DONEwhile specified condition is true. As soon as the condition becomes false, the execution stops. Consider the following example: counter0, while $counter -lt 3; Let the counter No.1 echo $counter while the loop will keep running the closed code only while the variable counter is less than 3. This condition is set on Line 4. During each cycle cycle on lines 5, the variable counter is infused with one. Once the variable counter is 3, the condition defined on lines 4 becomes false and the cycle stops. SUBSCRIBE TO NEWSLETTER Subscribe to Linux Career NEWSLETTER and receive the latest Linux news, jobs, career tips and tutorials. Before the loop Last Loop we are going to cover in this tutorial scenarios before the cycle. As long as the cycle does the exact opposite while the cycle. As long as the cycle also acts in a given state. However, the code between DOand DONEis was repeatedly executed only until this condition changed from false to true. The pre-cycle run is illustrated with the example below: !/bin/bash counter6 until $counter -lt 3; Let the counter-No.1 echo $counter done if you understand the above while looping the script until the loop is somewhat self-evident. The script starts with a variable counter set at 6. A condition defined on line 4 of this particular until the cycle continues to run the closed code until the condition becomes true. At this point we can transform our understanding of loops into something tangible. Our current backup script is currently capable of backing up a single directory to run. It would be nice to be able to back up all the directories that are delivered to the script on the command line after it is executed. Browse the updated script below, which implements this new feature: !/bin/bash This bash script is used to back up the user's home catalog to /tmp/. Backup if z $1 Then user '$(whoami) still if th ! -d /home/$1 ; Then the Echo Requested $1 custom home directory does not exist. 1 fi user'$1 fi input/home/$user output'/tmp/$-user'home'$$(date :%Y-%-%-%d_%H%% M%S function).tar.g the total_files - find the function $1-type f-wc-l - function total_directories - find the function $1-type d -l - function total_archived_directories - tar -tzf $1 function total_archived_files resin -tzf $1 y grep -v /$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) I echo $user Echo Files, which will be included: $src to be included: $src-catalogues Echo Files are archived: $arch files Echo Catalogs in the archive: $arch-catalogues, If $src -eq $arch then echo backup $input completed! Echo Details about the backup file inference: ls-l $output still echo backup $input failed! Fi for the catalog in $; Make a backup $directory done; After seeing the script above, you may have noticed A new feature called backup on lines 5 - 57was created. This feature includes all of our previously written code. The function definition ends at line 57 after we introduced a new loop on the 59 - 51 to perform a new backup function for each user directory provided as an argument. If you remember, the $E variable contains all the arguments provided on the command line when running the script. In addition, cosmetic changes to the code on line 44 are better readability of the output script by separating each catalog of backup output information from the hash line. Let's see how it works: $./backup.sh the linuxconfig damian to be included: 4 Files archived: 27 directories archived: 4 Backup/home/linuxconfig completed! Details of the backup file: -rw-r-r-r-r-- 1 linuxconfig 236173 Oct 23 10:22/tmp/linuxconfig_home_2017-10-23-1022 29.tar.gz Damian and files, to be included: 3 directories to be included: 1 Files archived: 3 Directories archived: 1 Backup/Home/Damian Completed! Details of the back-up file: -rw-r-r-r-r-- 1 linuxconfig 2140 October 23 10:22/tmp/damian_home_2017-10-23-1022330.tar.gz Bash Bashs In the last section of this tutorial scripted, we will discuss some of the basics of bash arithmetic. The arithmetic in bashing scenarios will add another layer of complexity and flexibility to our scripts as it allows us to calculate numbers even with numerical accuracy. There are several ways to perform arithmetic operations in bash scripts. Let's look at some of them using a few simple examples. Arithmetic expansion Arithmetic expansion is probably the easiest method on how to achieve basic calculations. We just attach any mathematical expression inside the double brackets. Let's perform a few simple additions, subtraction, multiplication and dividing calculations with integers: expr commands Another alternative to arithmetic expansion expr team. Using the expr command allows us to perform arithmetic without even attaching our mathematical expression in brackets or quotes. However, be sure to avoid the asterisk multiplication sign to avoid expr: syntax error: let the team similarly, as with the expr team, we can perform bash arithmetic operations with let commands. let the team evaluate the mathematical expression and keep its result in a variable. We alredy faced let the team in one of our previous examples where we used it to perform an integrator increments. The following example shows some basic operations using let command as well as integrator increments and demonstration operations like x3:bc command After a few minutes of experimentation with the above bash arithmetic techniques, you may have noticed that they work perfectly with numbers, however when it comes to decimal numbers there is something wrong. we will need to use the bc team to take our arithmetic to a completely different level. BC team with proper syntax syntax for more than simple integer calculations. The operational command lead of the bc is quite extensive as it spans over more than 500 lines. However, it doesn't hurt to show some basic operations. The next example would be a division operation with 2 and 30 decimals and a square root of 50 with 50 decimal numbers. By default, the bc team will produce all results as an integer number. Use scale'x to instruct the B.C. team to show the real numbers: Let's put our new knowledge of Bash arithmetic to work and once again change our script backup.sh to implement the counter of all archive files and directories for all users: backup feature if - z $1; Then user '$(whoami) still if th ! -d /home/$1 ; Then the Echo Requested $1 custom home directory does not exist. 1 fi user'$1 fi input/home/$user output'/tmp/$-user'home'$$(date :%Y-%-%-%d_%H%% M%S function).tar.g the total_files - find the function $1-type f-wc-l - function total_directories - find the function $1-type d -l - function total_archived_directories - tar -tzf $1 function total_archived_files resin -tzf $1 y grep -v /$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) I echo $user Echo Files, which will be included: $src to be included: $src-catalogues Echo Files are archived: $arch files Echo Catalogs in the archive: $arch-catalogues, If $src -eq $arch then echo backup $input completed! Echo Details about the backup file inference: ls-l $output still echo backup $input failed! Fi for the catalog in $; Make a backup $directory let $all-$arch files $arch directories done; TOTAL FILES Echo AND DIRECTORIES: $all On Line 60 we used the supplement to add all the archival files using let the command resulting in a variable everything. Each iteration of the cycle adds a new score for each additional user. The result is then printed using the Echo command on line 62.Example execution scenario: $./backup.sh linuxconfig damian - Linuxconfig - Files to be included: 27 directories to be included: 6 files archived: 6 Directories archived: 6 Backups/home/linuxconfig completed! Details of the backup file: -rw-r-r-r-r-- 1 linuxconfig 237004 Dec 27 11:23/tmp/linuxconfig_home_2017-12-27-1123 59.tar.gz Damian and files, to be included: 3 directories to be included: 1 Files archived: 3 Directories archived: 1 Backup/Home/Damian Completed! Details of the backup file: -rw-r-r-r-r-r-- 1 linuxconfig linuxconfig 2139 December 27 11:23 TOTAL FILES AND DIRECTORIES: 37 Conclusion There Are Still Shell Bash than those covered in this textbook. However, before you move on, make sure you are comfortable with the topics discussed here. Aside from googling, there are numerous other resources available online to help you if you get stuck. The most notable and highly recommended of them all is the GNU Bash Handbook Guide. Manually. bash shell script tutorial for beginners. bash shell script tutorial pdf. bash shell script tutorialspoint. bash shell script tutorial for beginners pdf. advanced bash shell script tutorial. ubuntu bash shell script tutorial