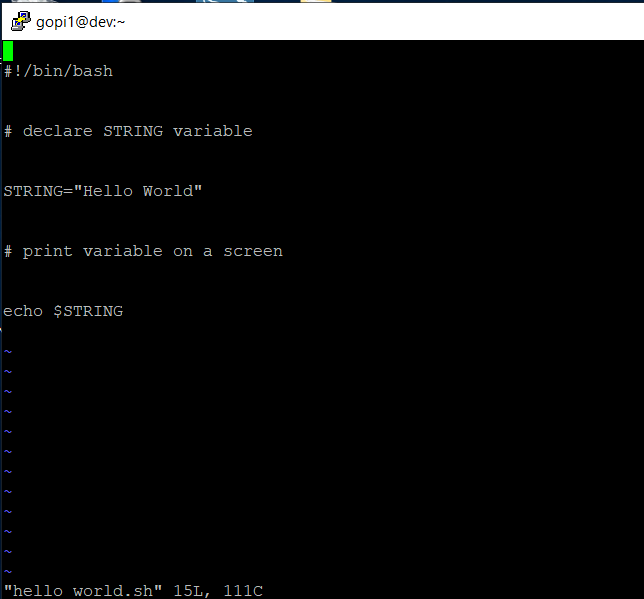
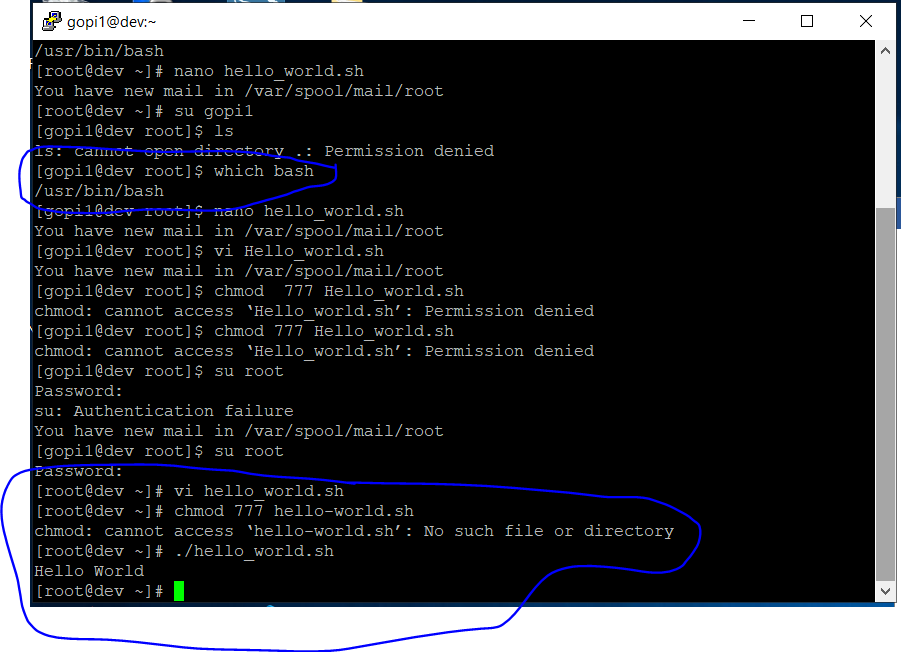
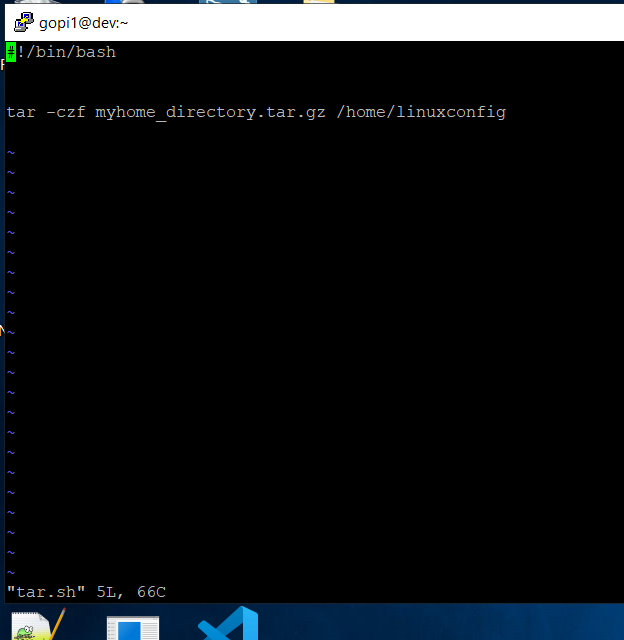
# Hello World Bash Shell Script – Bash Scripting Tutorial

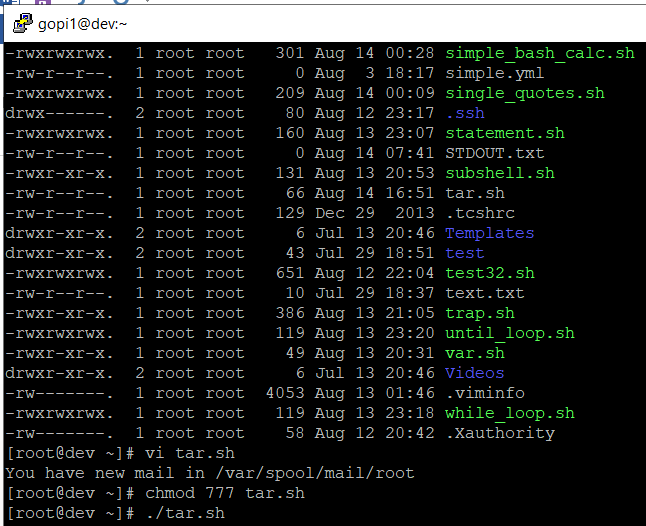
The next thing you need to do is open our favorite text editor and create a file called hello\_world.sh. We will use nano for this step.

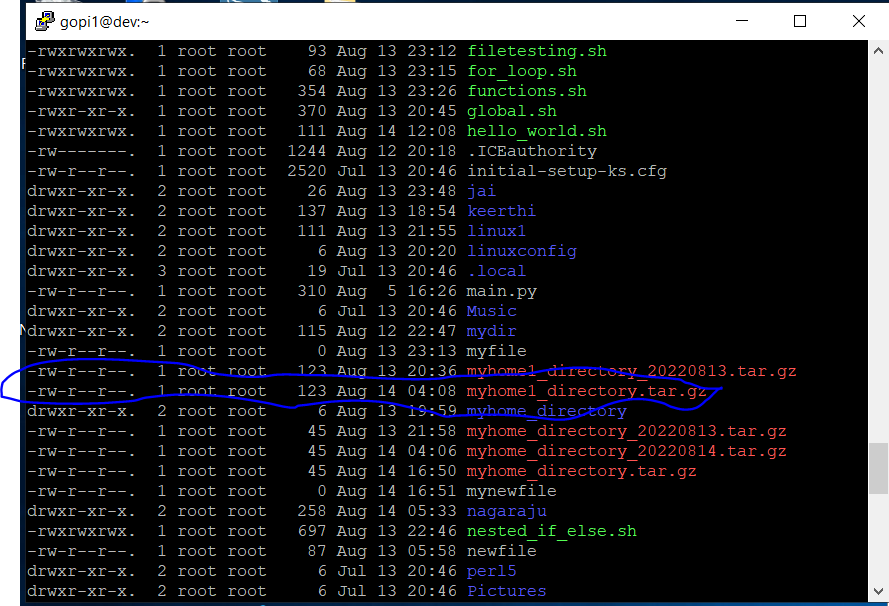


# Simple Backup bash shell script

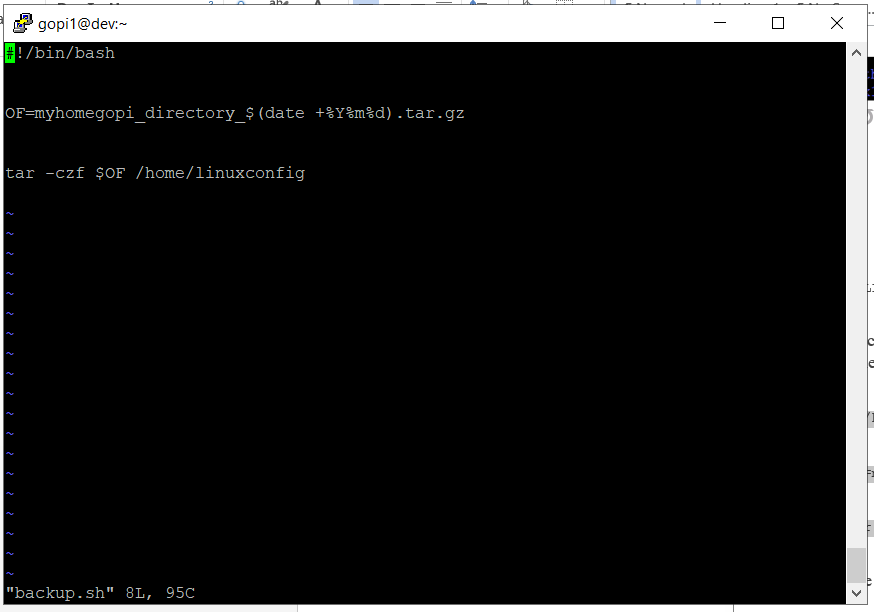
When writing a Bash script, you are basically putting into it the same commands that you could execute directly on the command line.

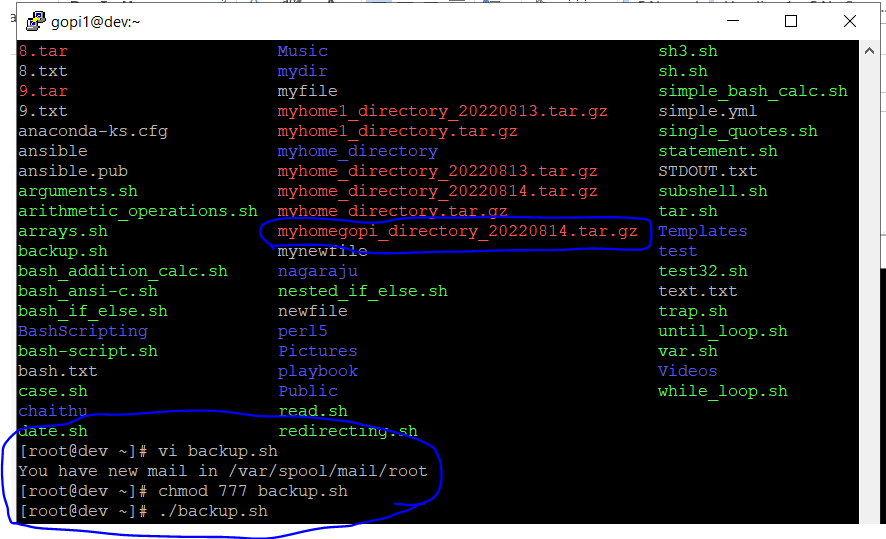






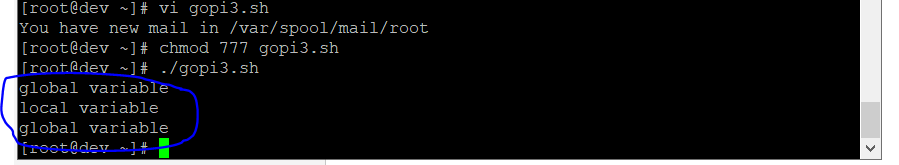
Circling back to our backup script example, let’s use a variable to name our backup file and put a time stamp in the file name by using the date command.

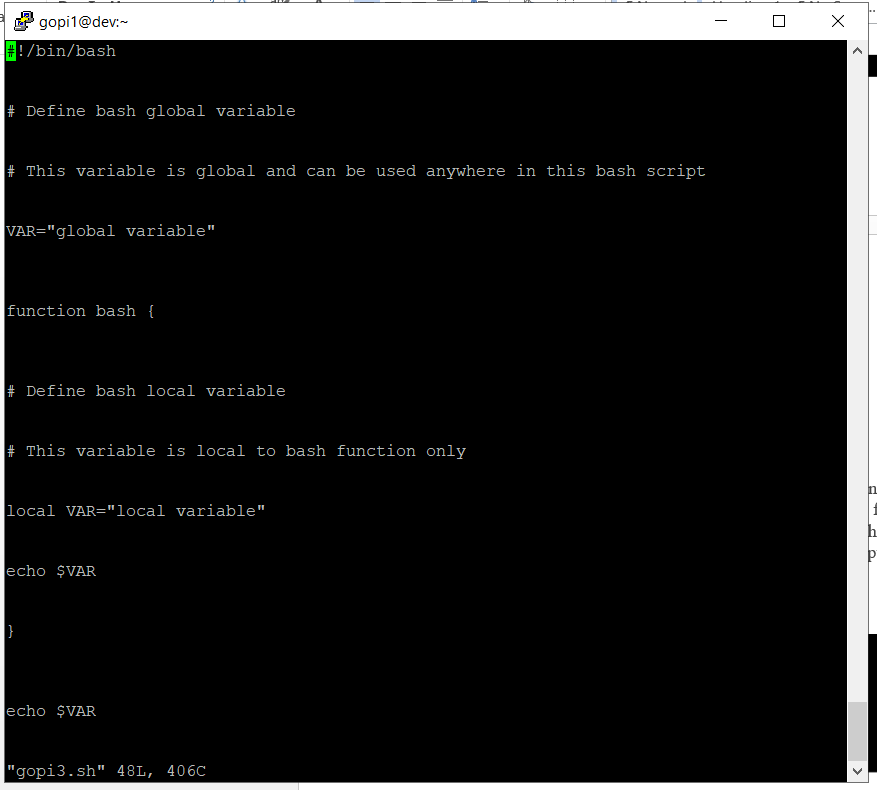




Global vs. Local variables

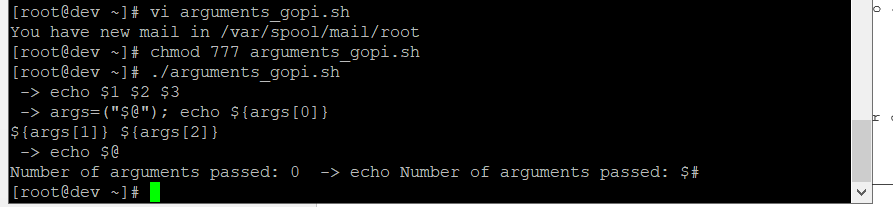
In Bash scripting, a global variable is a variable that can be used anywhere inside the script. A local variable will only be used within the function that it is declared in. Check out the example below where we declare both a global variable and local variable. We’ve made some comments in the script to make it a little easier to digest.

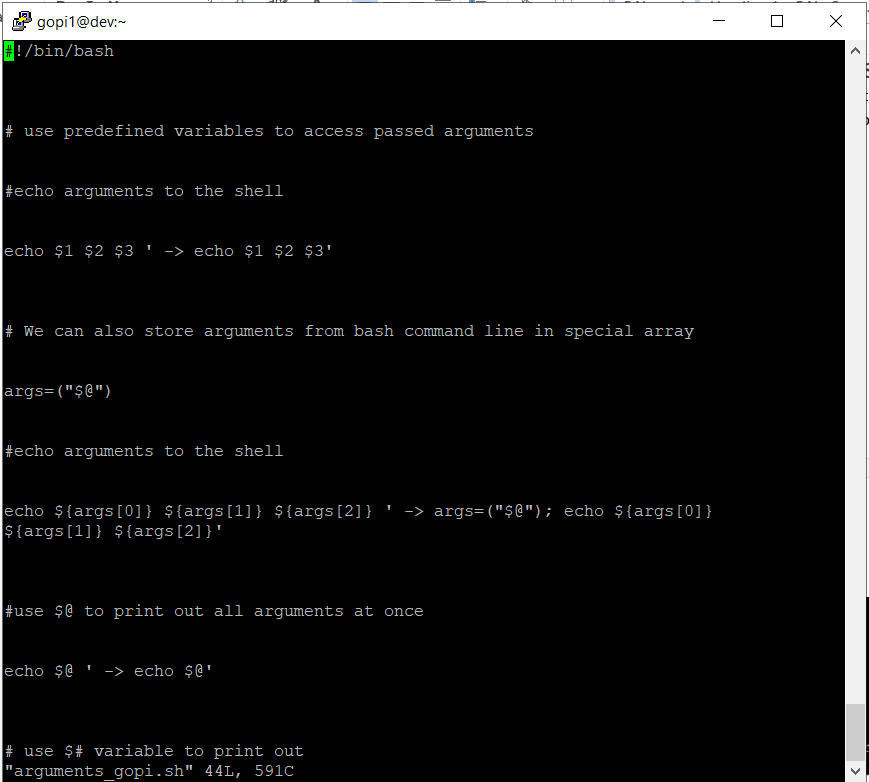




# Passing arguments to the bash script

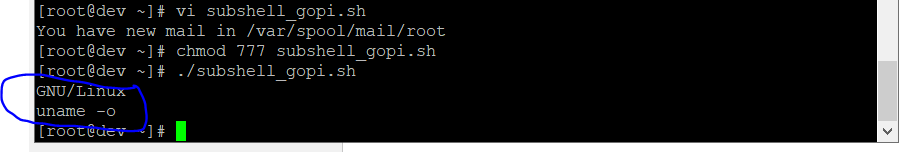
When executing a Bash script, it is possible to pass arguments to it in your command. As you can see in the example below, there are multiple ways that a Bash script can interact with the arguments we provide.

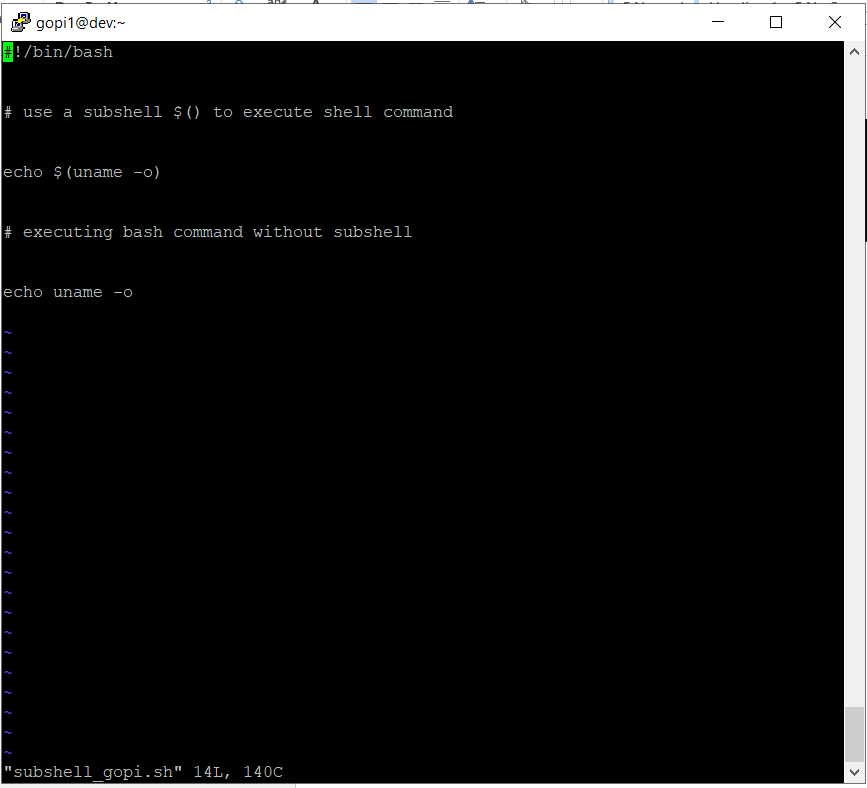




Executing shell commands with bash

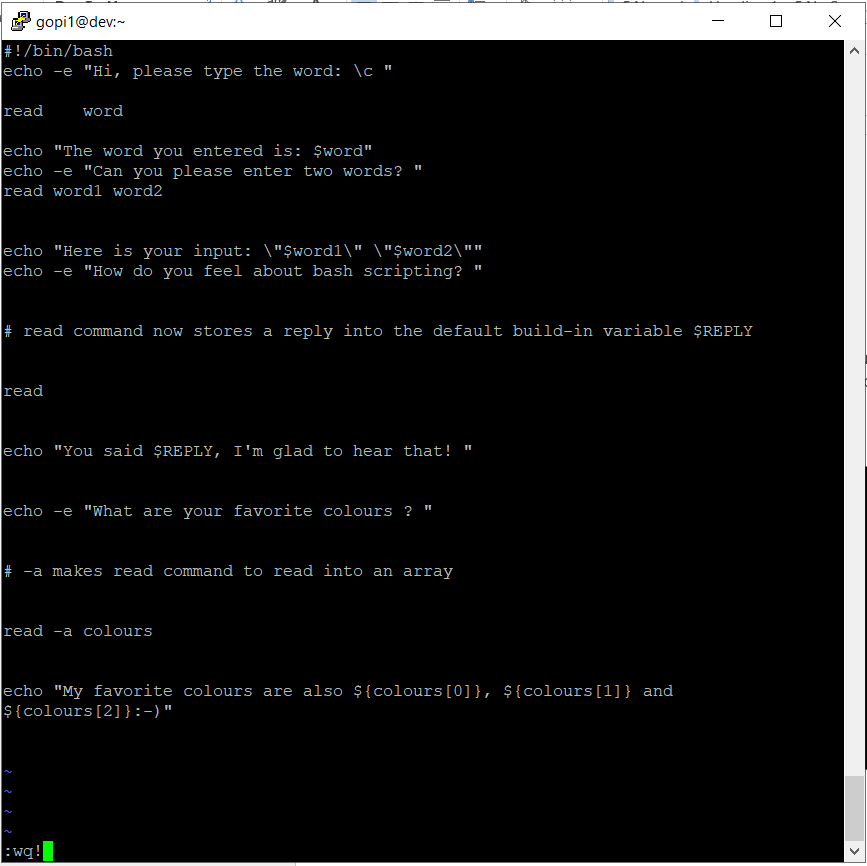
The best way to execute a separate shell command inside of a Bash script is by creating a new subshell through the $( ) syntax. Check the example below where we echo the result of running the uname -o command.

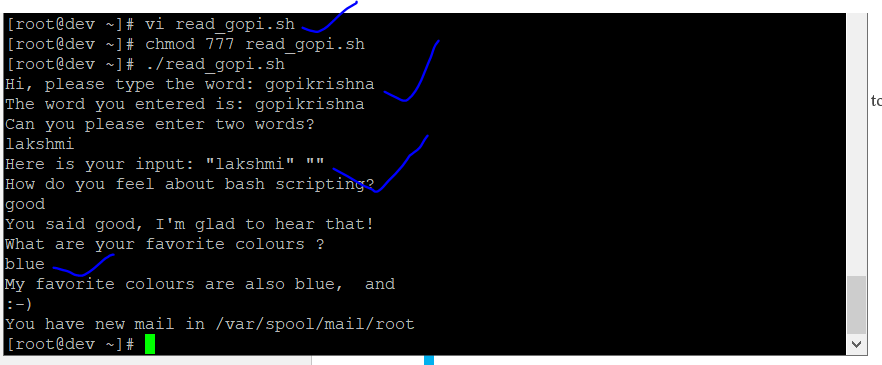




# Reading User Input

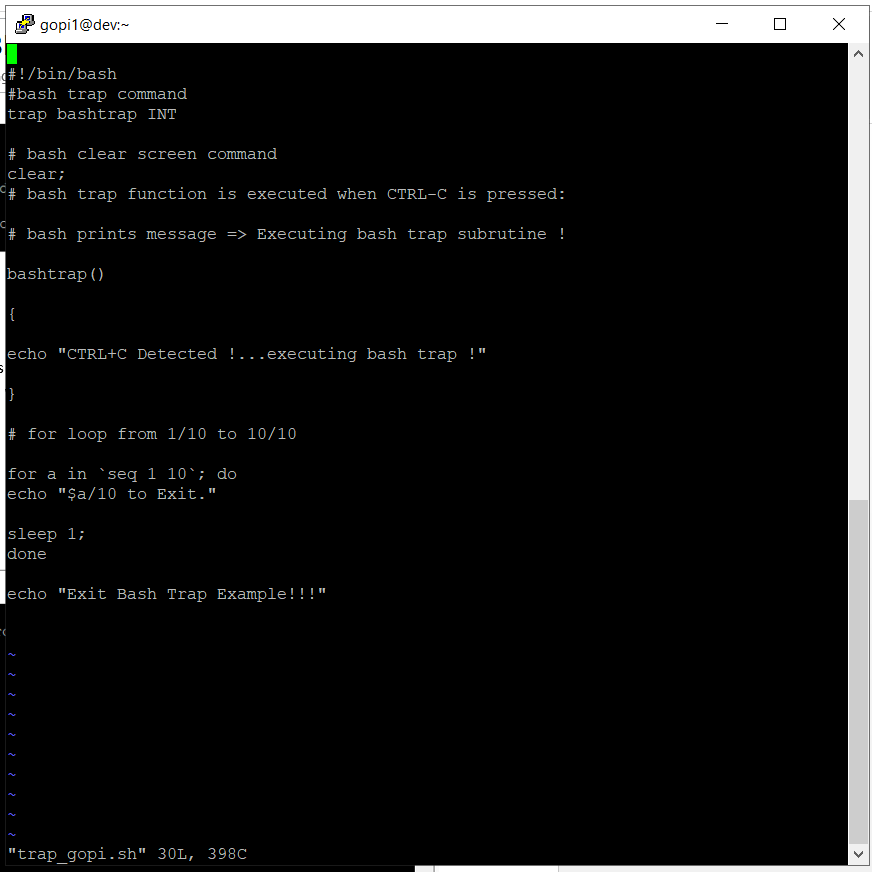
### We can use the **read** command to read input from the user. This allows a user to interact with a Bash script and help dictate the way it proceeds. Here’s an example:

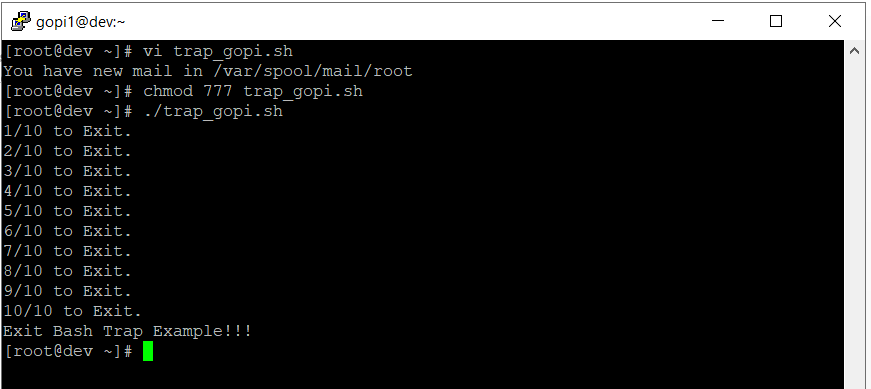




Bash Trap Command

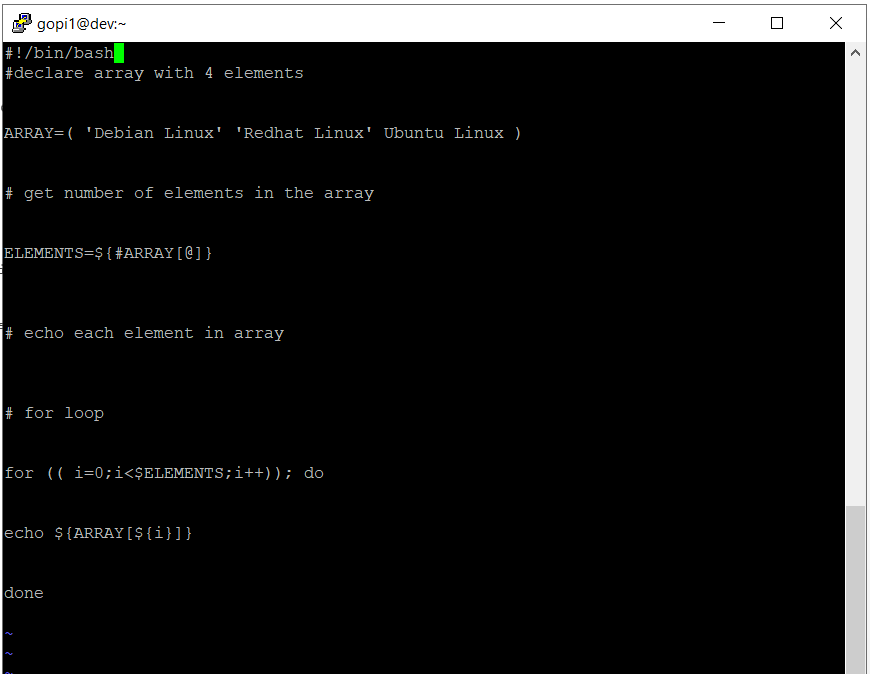
The trap command can be used in Bash scripts to catch signals sent to the script and then execute a subroutine when they occur. The script below will detect a Ctrl + C interrupt

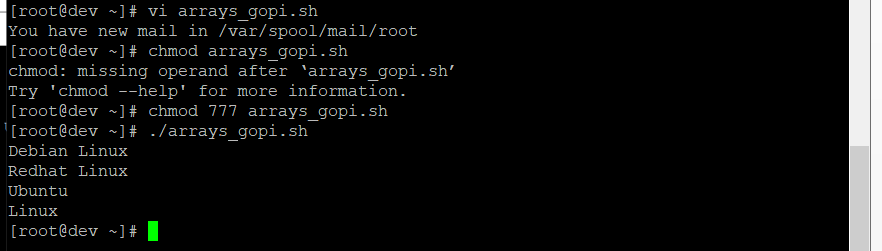




Arrays

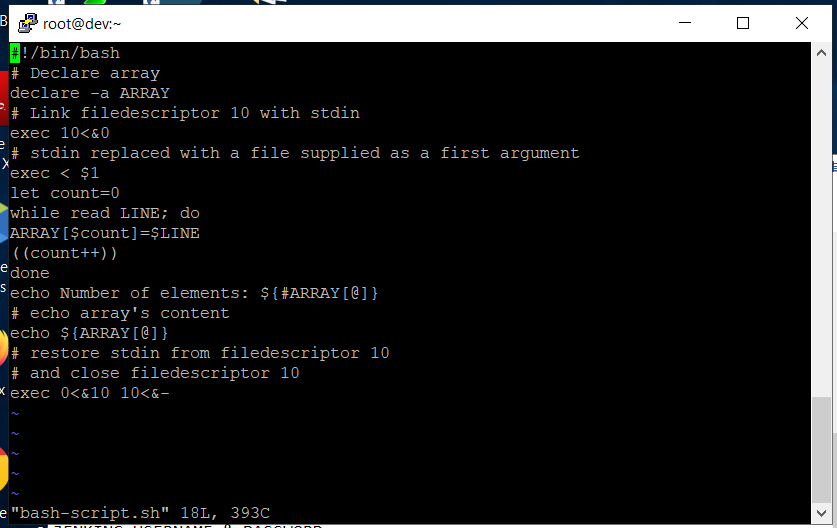
Bash is capable of storing values in arrays. Check the sections below for two different examples.



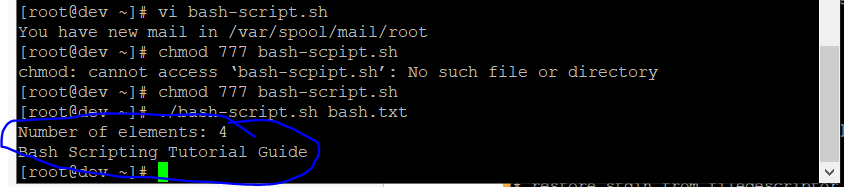


Read file into bash array

### Rather than filling out all of the elements of our array in the Bash script itself, we can program our script to read input and put it into an array.

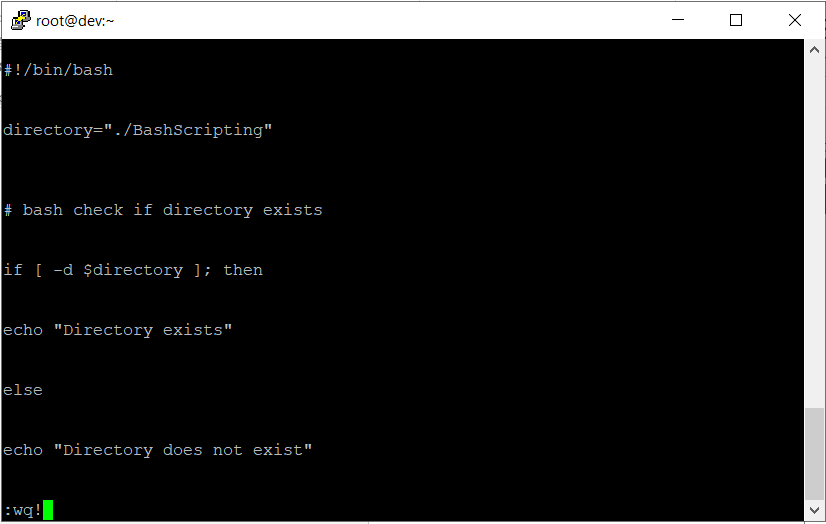


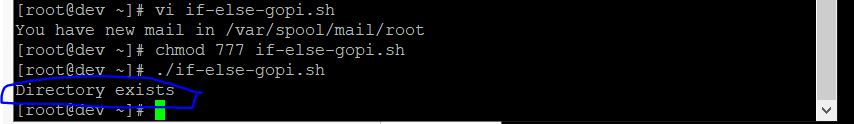




# Bash if / else / fi statements

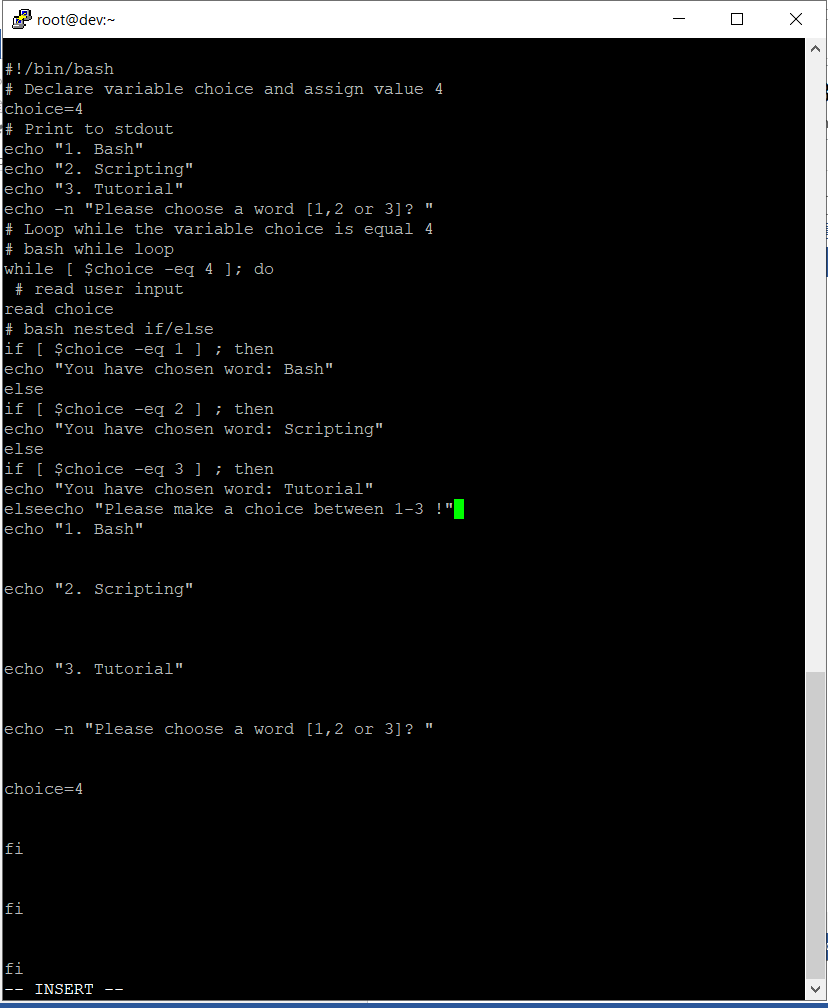
### Here is a simple **if** statement that check to see if a directory exists or not. Depending on the result, it will do one of two things. Please note the spacing inside the **[** and **]** brackets! Without the spaces, it won’t work!

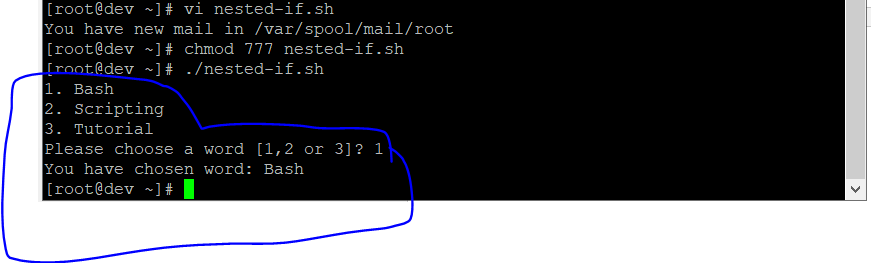




Nested if/else

### It is possible to place an **if** statement inside yet another **if** statement. This is called nesting. Scripts can get a bit complex depending on how many **if** statements deep it is.





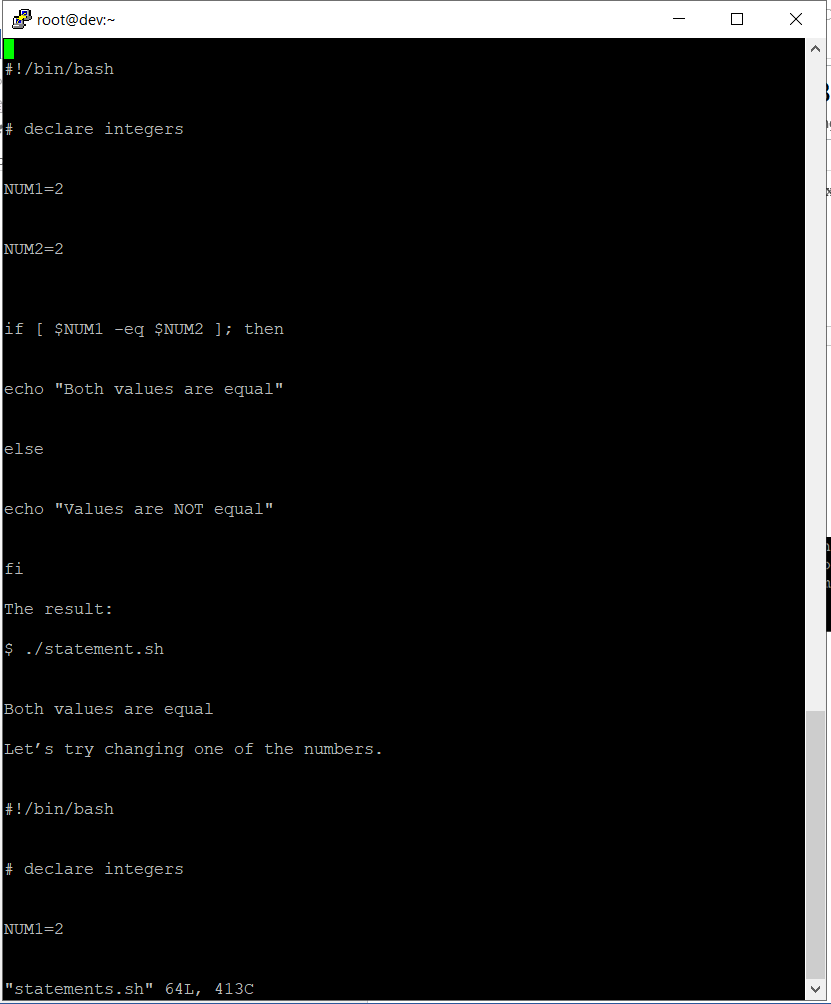
# Bash Comparisons

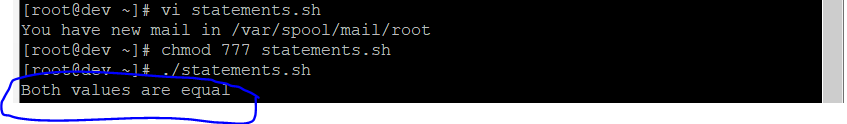
### Bash can compare two or more values, either integers or strings, to determine if they are equal to each other, or one is greater than the other, etc.

Arithmetic Comparisons

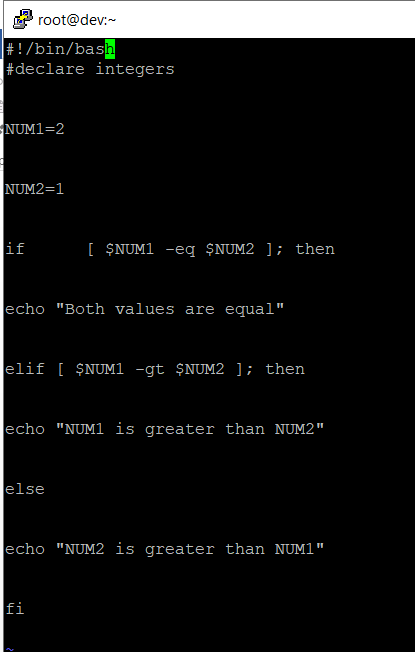
|  |  |
| --- | --- |
| -lt | < |
| -gt | > |
| -le | <= |
| -ge | >= |
| -eq | == |
| -ne | != |

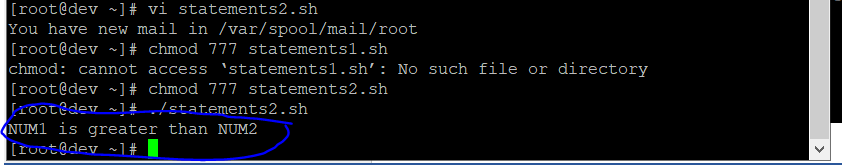
Now let’s use these operators in some examples.





### Let’s add a little more complexity by including an **elif** statement and determing which number is larger.

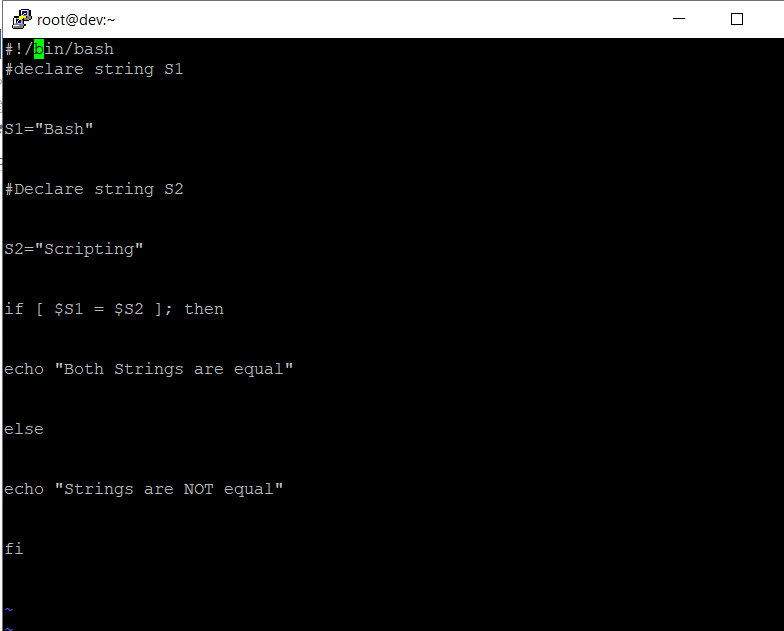


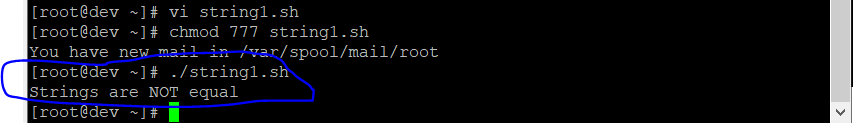


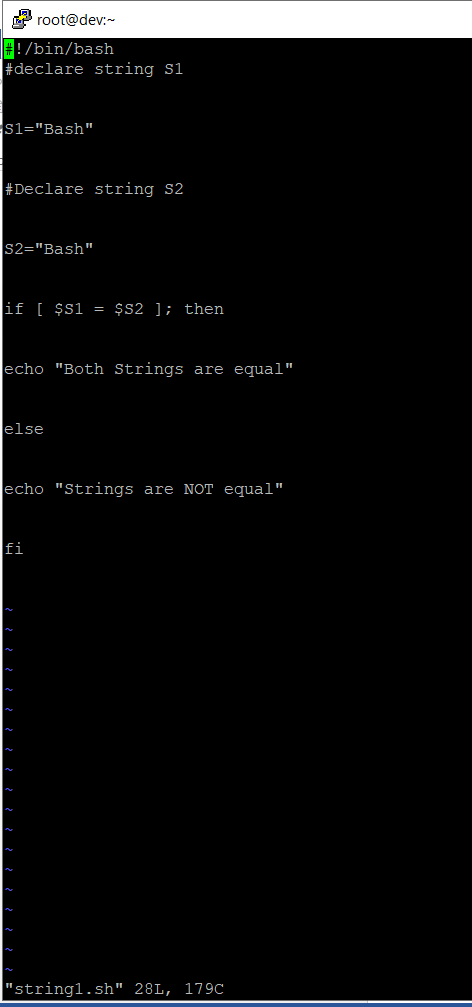
String Comparisons

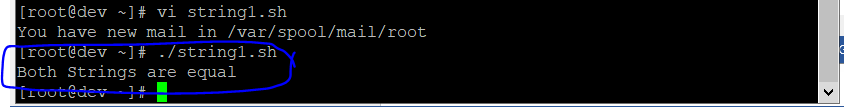
### Let’s try comparing two strings to see if they are equal.

|  |  |
| --- | --- |
| = | equal |
| != | not equal |
| < | less then |
| > | greater then |
| -n s1 | string s1 is not empty |
| -z s1 | string s1 is empty |









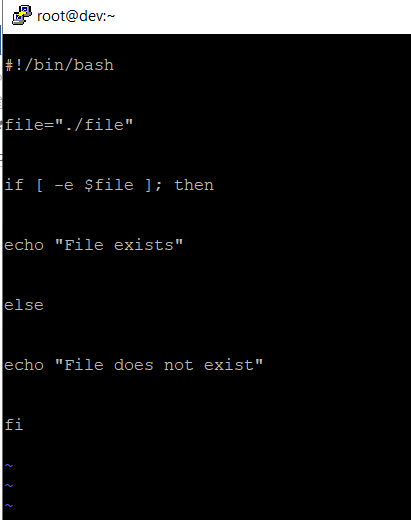
# Bash File Testing

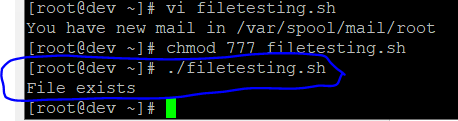
### In Bash, we can test to see different characteristics about a file or directory. See the table below for a full list.

|  |  |
| --- | --- |
| -b filename | Block special file |
| -c filename | Special character file |
| -d directoryname | Check for directory existence |
| -e filename | Check for file existence |
| -f filename | Check for regular file existence not a directory |
| -G filename | Check if file exists and is owned by effective group ID. |
| -g filename | true if file exists and is set-group-id. |
| -k filename | Sticky bit |
| -L filename | Symbolic link |
| -O filename | True if file exists and is owned by the effective user id. |
| -r filename | Check if file is a readable |
| -S filename | Check if file is socket |
| -s filename | Check if file is nonzero size |

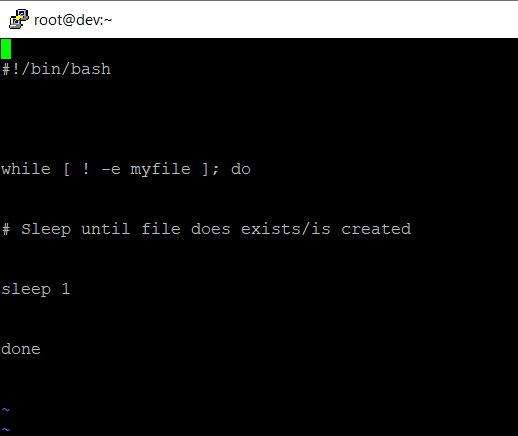
The following script will check to see if a file exists or not.

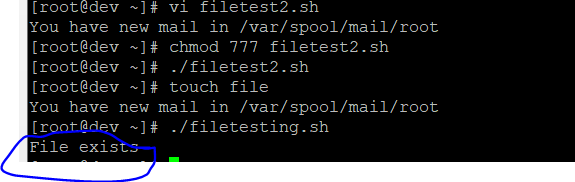
|  |  |
| --- | --- |
| -u filename | Check if file set-ser-id bit is set |
| -w filename | Check if file is writable |
| -x filename | Check if file is executable |





### Similarly for example we can use **while** loop to check if file does not exist. This script will sleep until file does exist. Note bash negator **!** which negates the **-e** option.



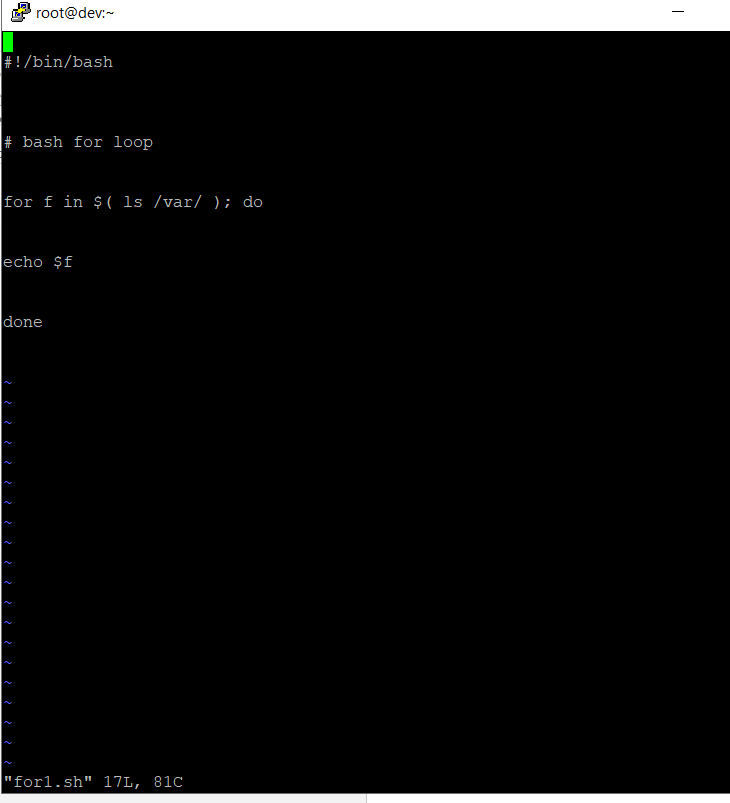


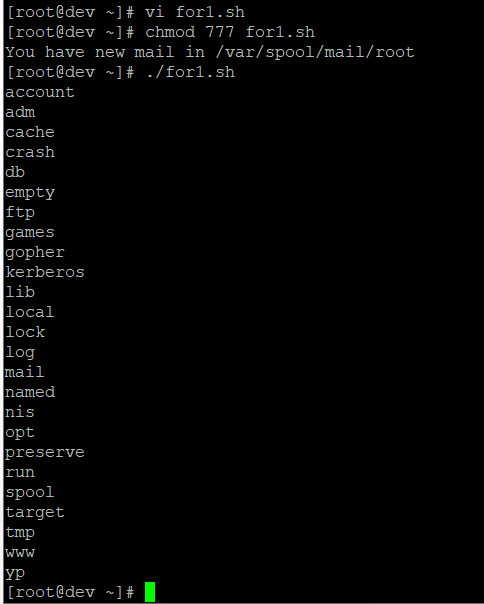
# Loops

### There are multiple types of loops that can be used in Bash, including **for**, **while**, and **until**. See some of the examples below to learn how to use.

Bash for loop

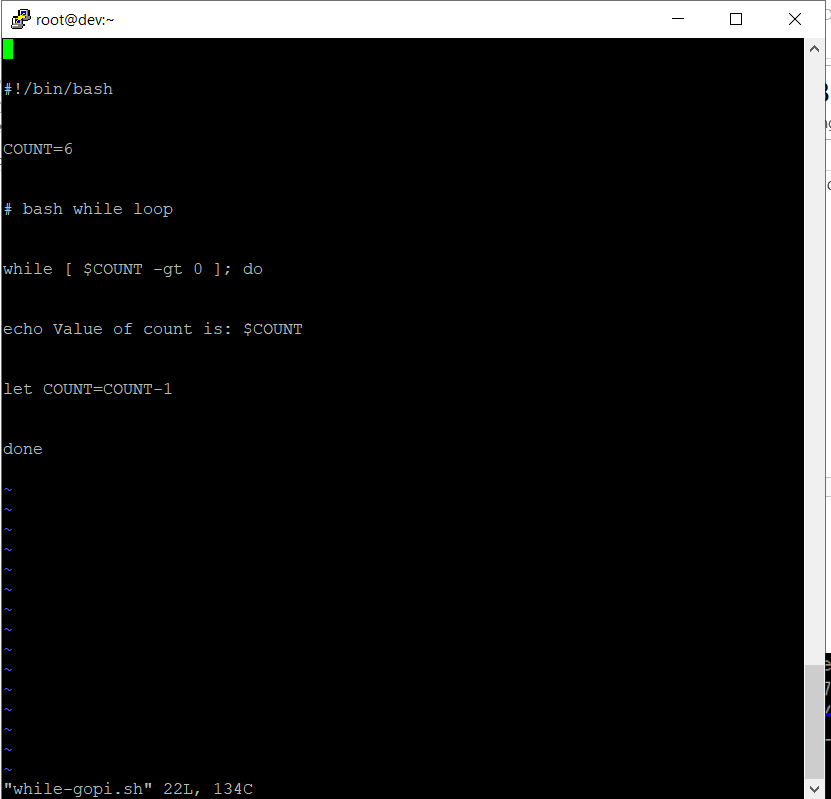
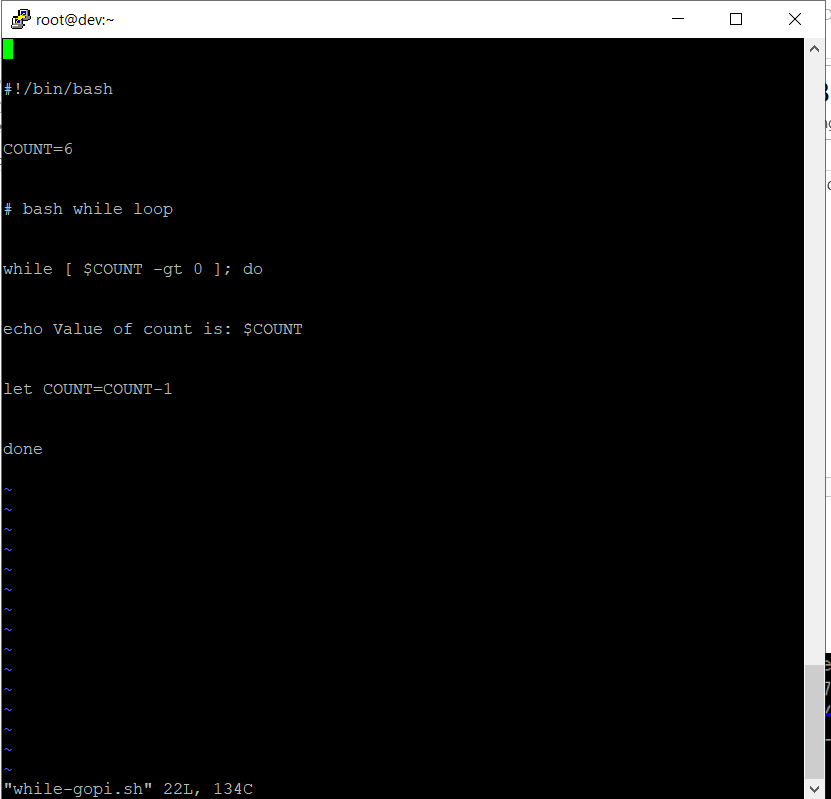
This script will list every file or directory it finds inside the **/var/** directory.

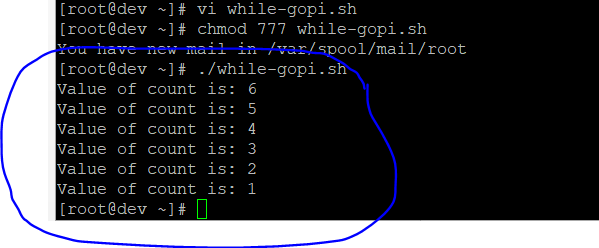




Bash while loop

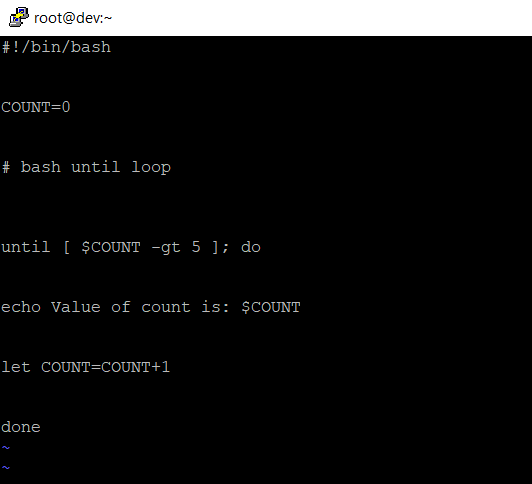
### This **while** loop will continue to loop until our variable reaches a value of 0 or less.

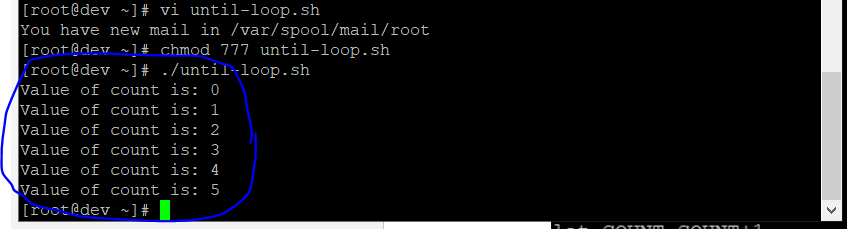




## Bash until loop

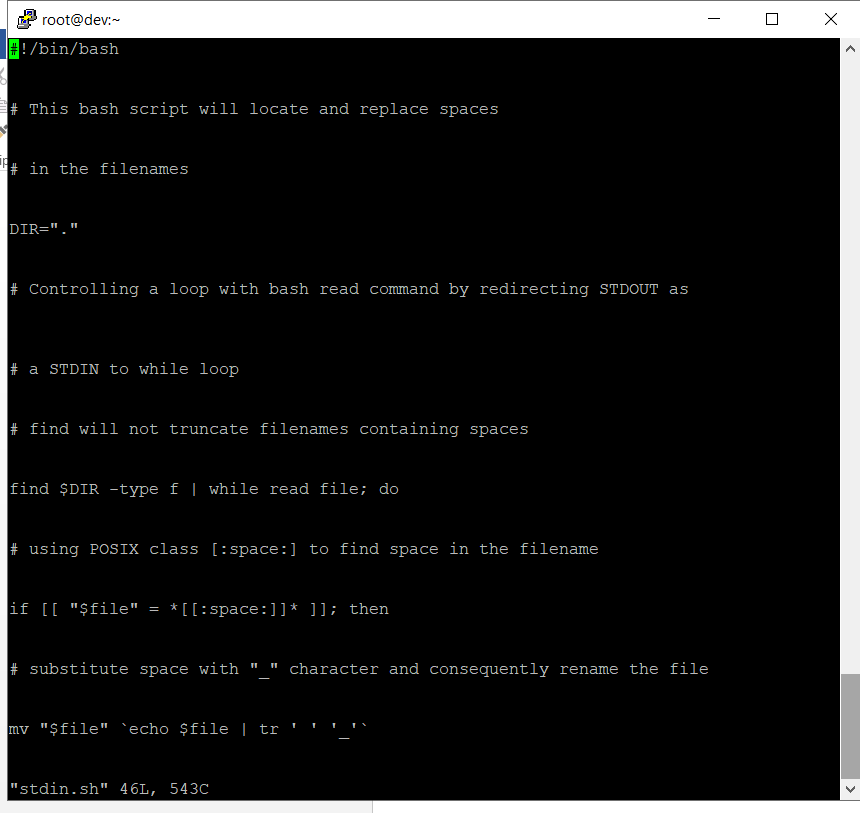
An **until** loop works similarly to **while**.

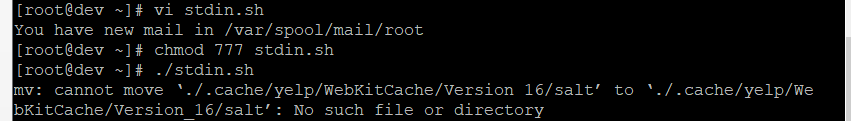




## Control bash loop with input

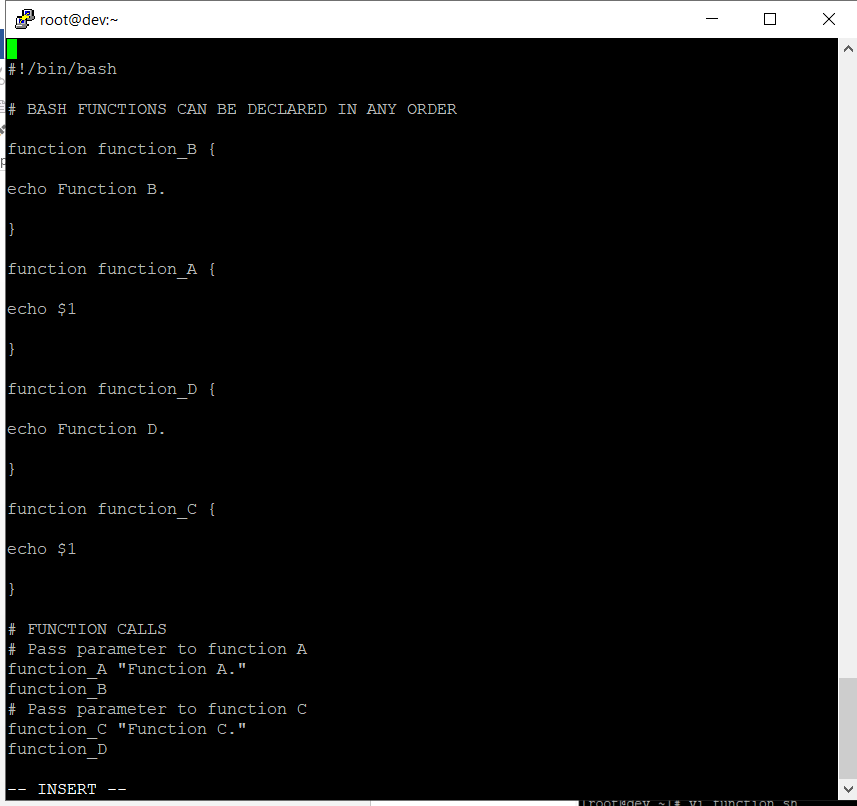
### Here is a example of **while** loop controlled by standard input. Until the redirection chain from STDOUT to STDIN to the **read** command exists the **while** loop continues.

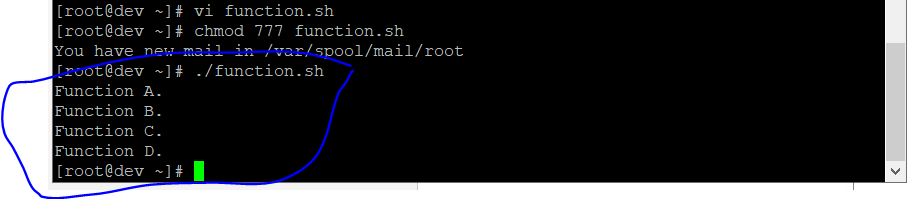




# Bash Functions

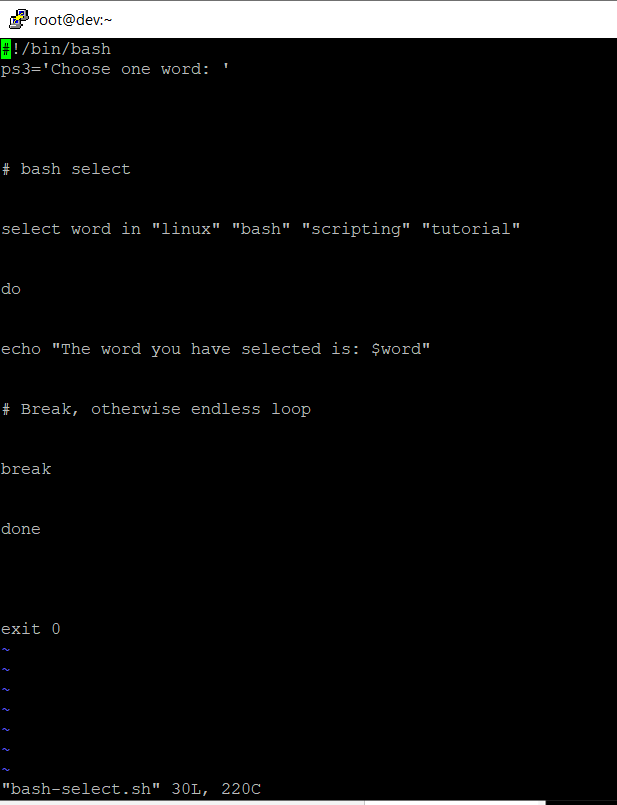
### This example shows how to declare a function and call back to it later in the script.

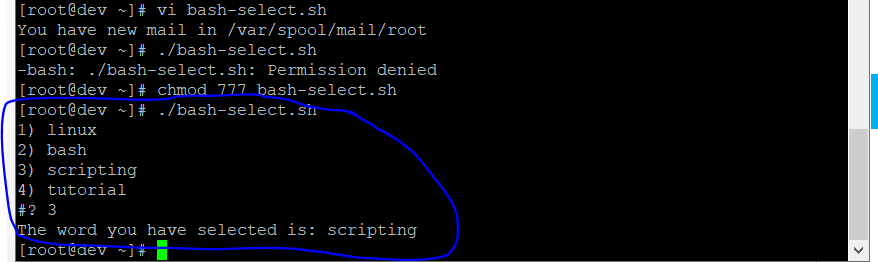




# Bash Select

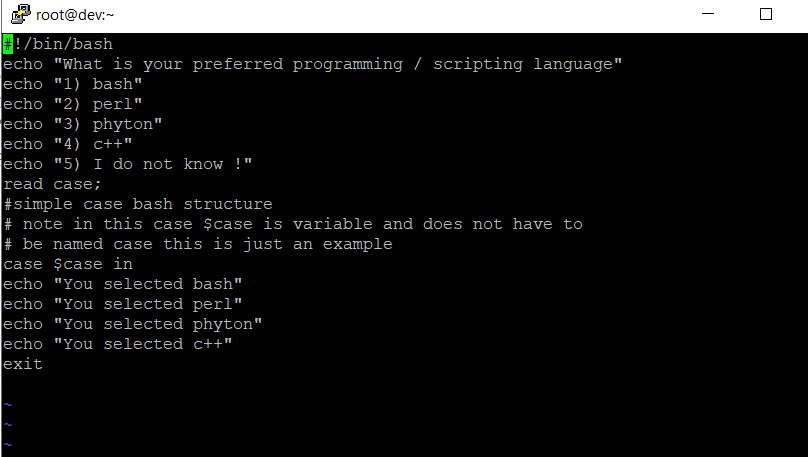
### The **select** command allows us to prompt the user to make a selection.

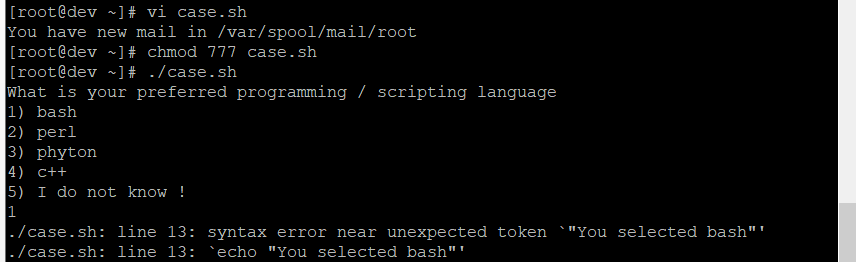




# Case statement conditional

### The **case** statement makes it easy to have many different possibilities, whereas an **if** statement can get lengthy very quickly if you have more than a few possibilities to account for.





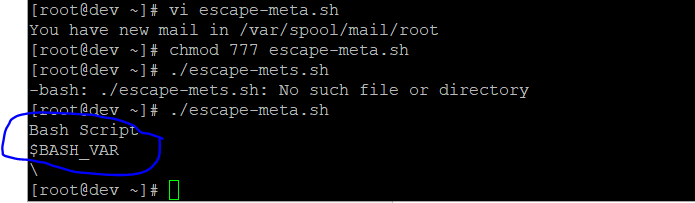
# Bash quotes and quotations

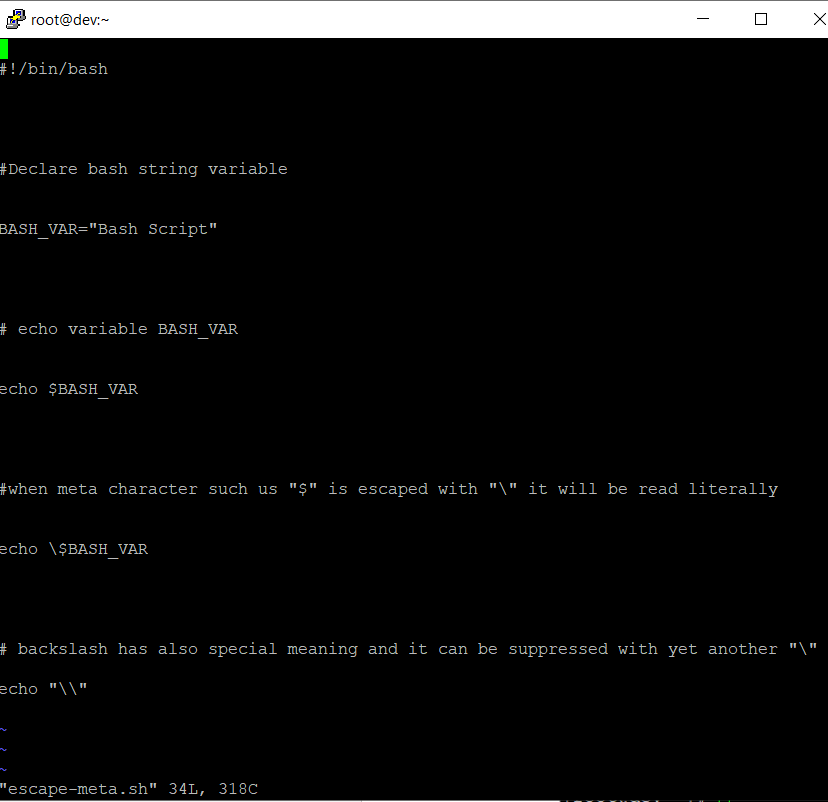
### Quotations and quotes are important part of bash and bash scripting. Here are some bash quotes and quotations basics.

Escaping Meta characters

Before we start with quotes and quotations we should know something about escaping meta characters. Escaping will suppress a special meaning of meta characters and therefore meta characters will be read by bash literally. To do this we need to use backslash **\** character.

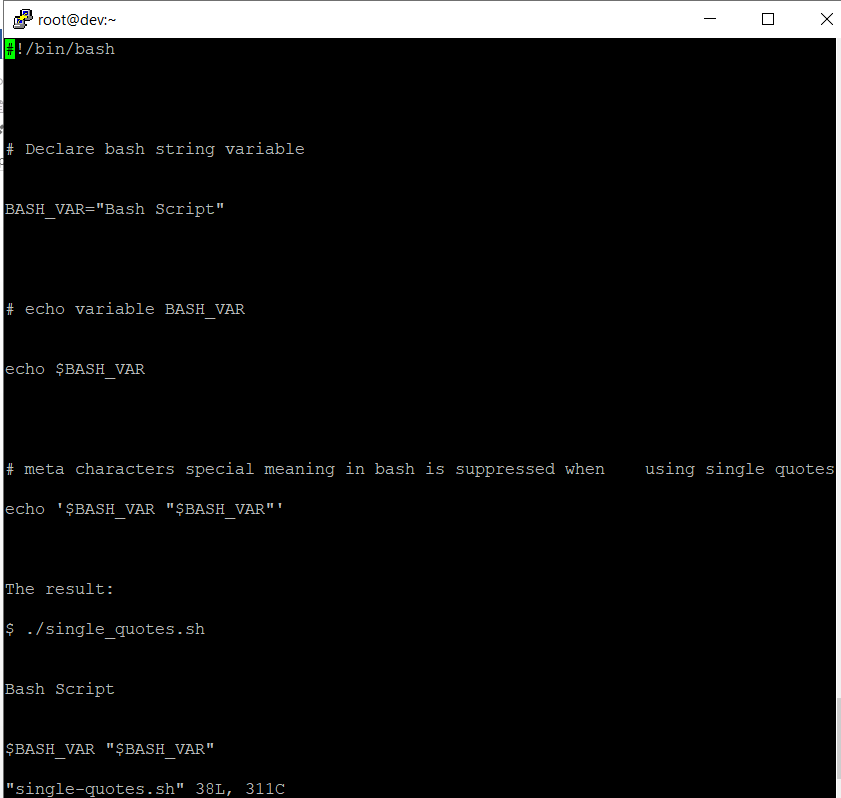
Example:

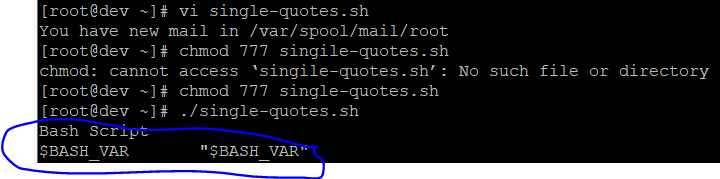




Single quotes

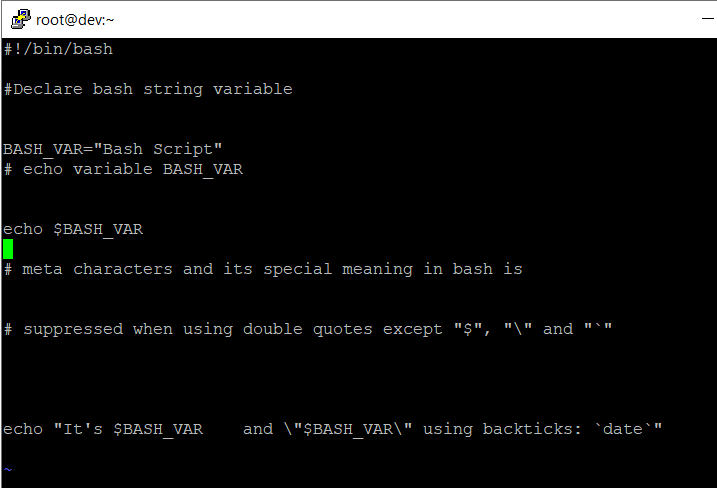
### Single quotes in bash will suppress special meaning of every meta characters. Therefore meta characters will be read literally. It is not possible to use another single quote within two single quotes not even if the single quote is escaped by backslash.

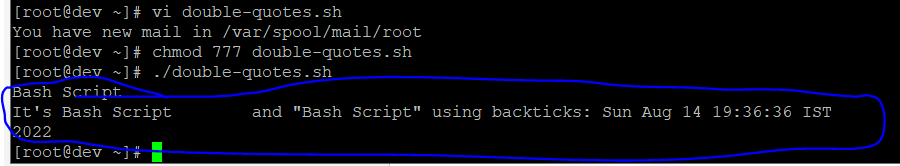




## Double quotes

### Double quotes in bash will suppress special meaning of every meta characters except **$**, **\** and **`**. Any other meta characters will be read literally. It is also possible to use single quote within double quotes. If we need to use double quotes within double quotes bash can read them literally when escaping them with **\**. Example:



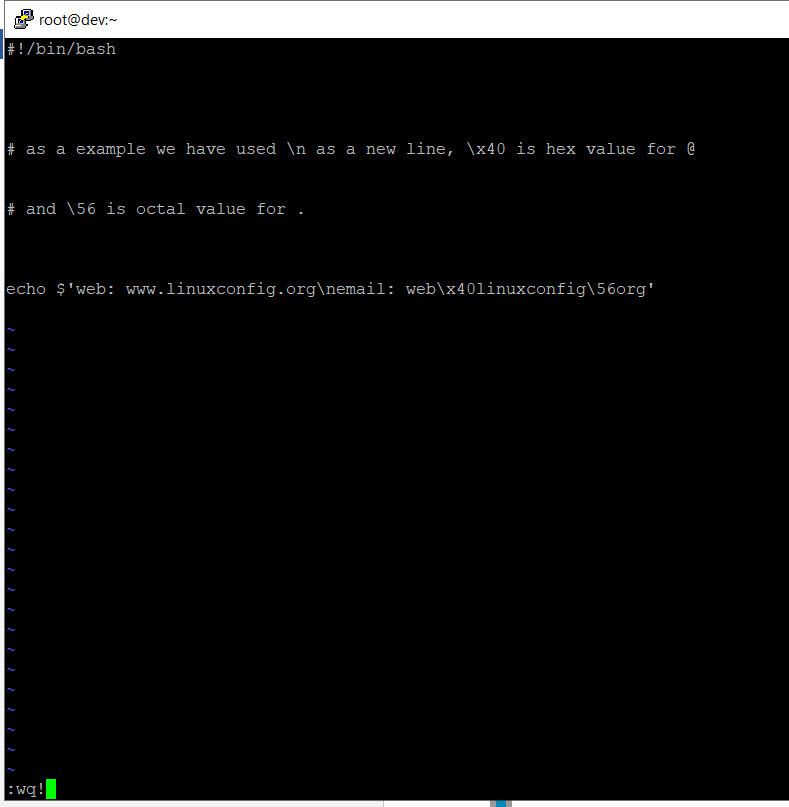


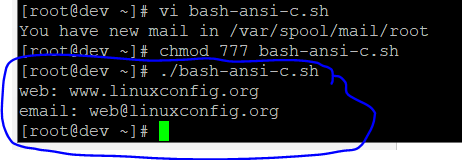
## Bash quoting with ANSI-C style

### There is also another type of quoting and that is ANSI-C. In this type of quoting characters escaped with **\** will gain special meaning according to the ANSI-C standard.

|  |  |  |  |
| --- | --- | --- | --- |
| \a | alert (bell) | \b | backspace |
| \e | an escape character | \f | form feed |
| \n | newline | \r | carriage return |
| \t | horizontal tab | \v | vertical tab |
| \\ | backslash | \` | single quote |
| \nnn | octal value of characters ( see [<http://www.asciitable.com/>ASCII table] ) | \xnn | hexadecimal value of characters ( see [<http://www.asciitable.com/>ASCII ta |

The syntax for ansi-c bash quoting is: **$' '** .

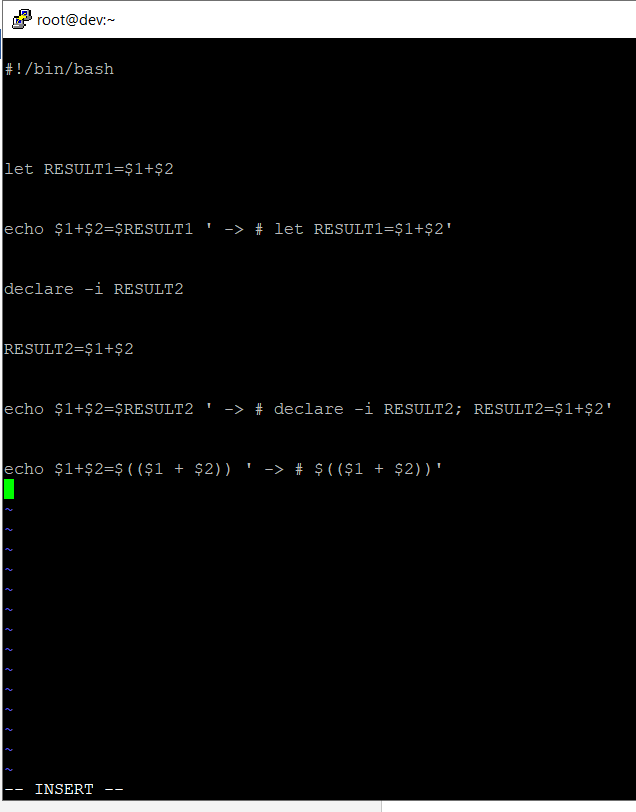


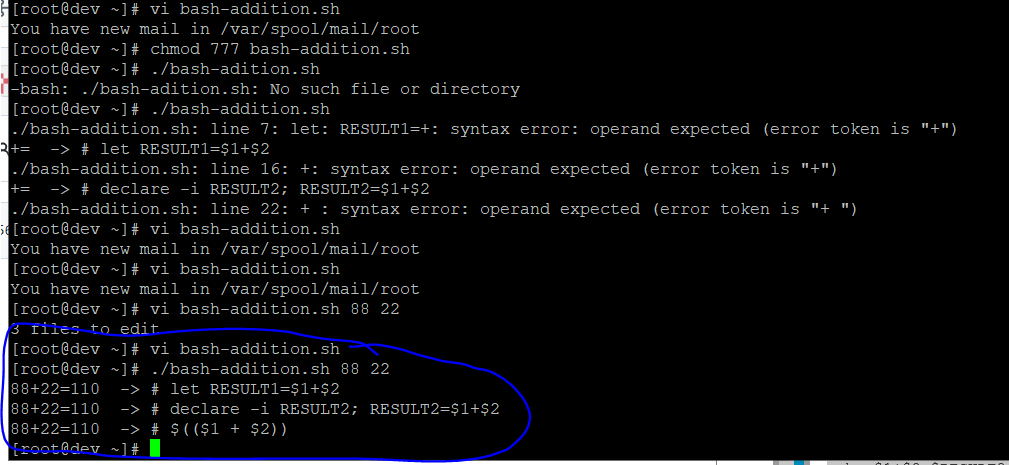


# Arithmetic Operations

### Bash can be used to perform calculations. Let’s look at a few examples to see how it’s done.

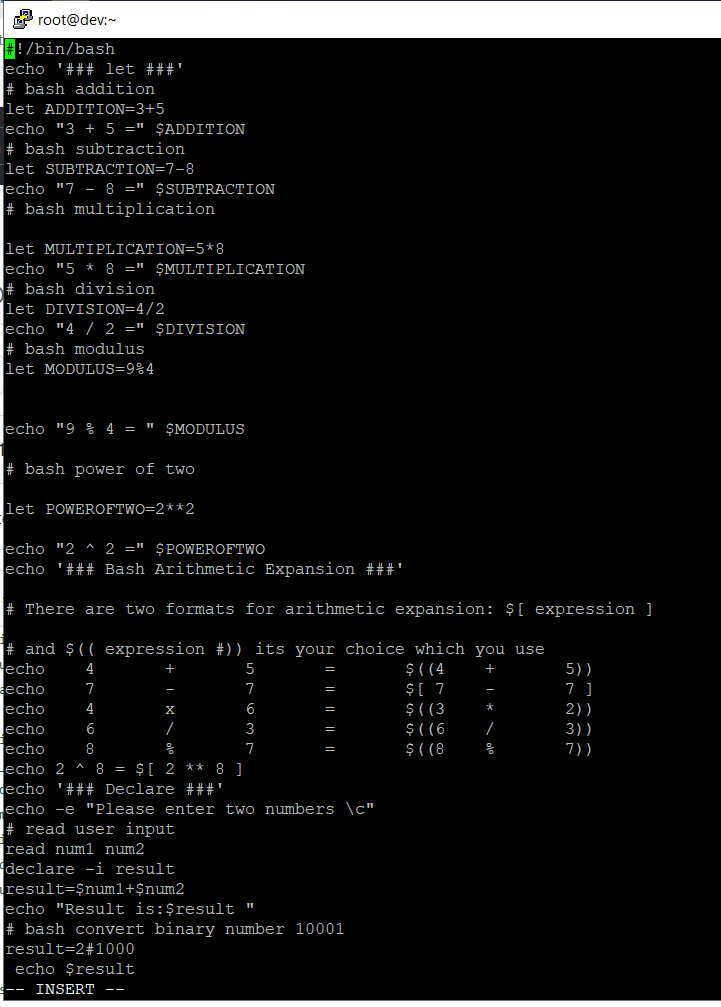
Bash Addition Calculator Example

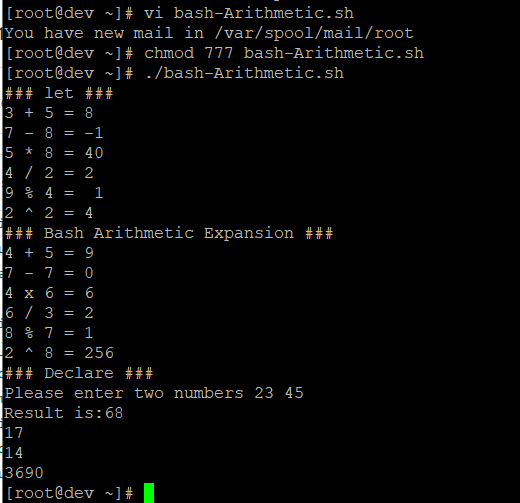




Bash Arithmetics

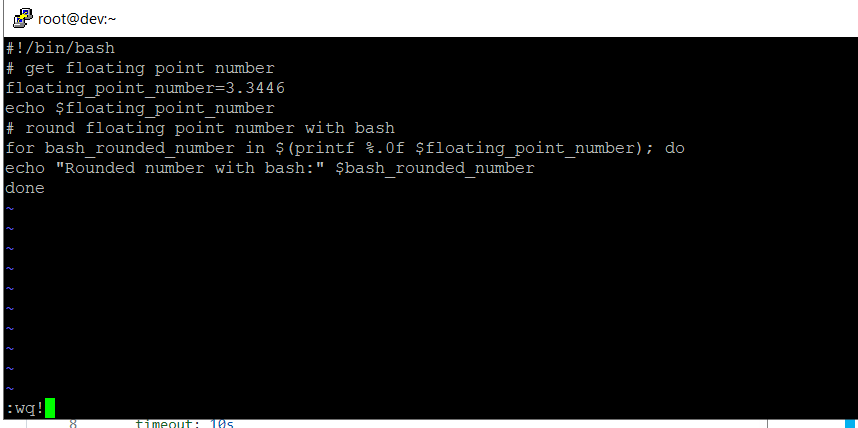
### Let’s see how to do some basic Bash aritmetics such as addition, subtraction, multiplication, division, etc.

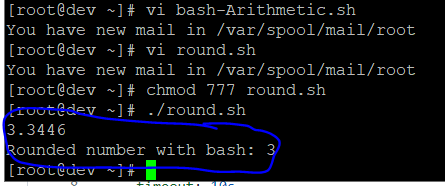




## Round floating point number

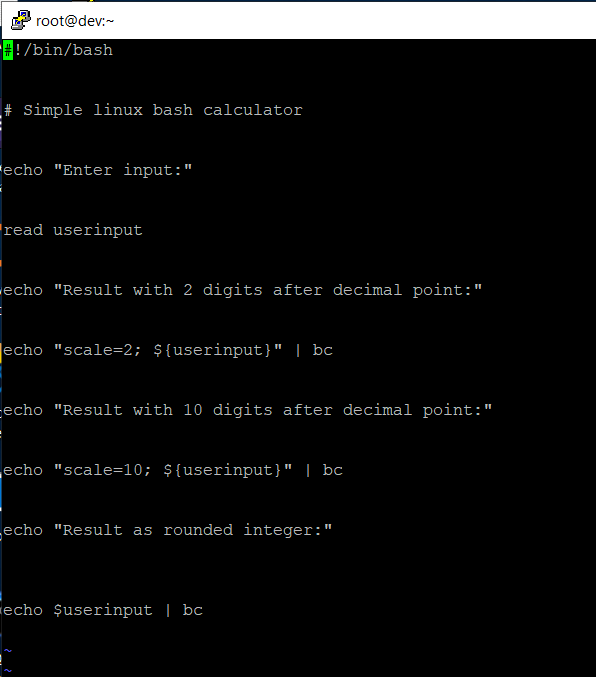
### Here is how to use rounding in Bash calculations.

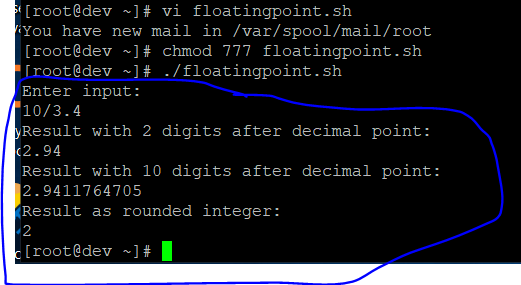




## Bash floating point calculations

### Using the **bc** bash calculator to perform floating point calculations.



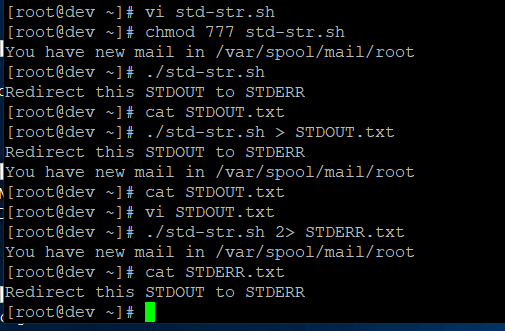


# Redirections

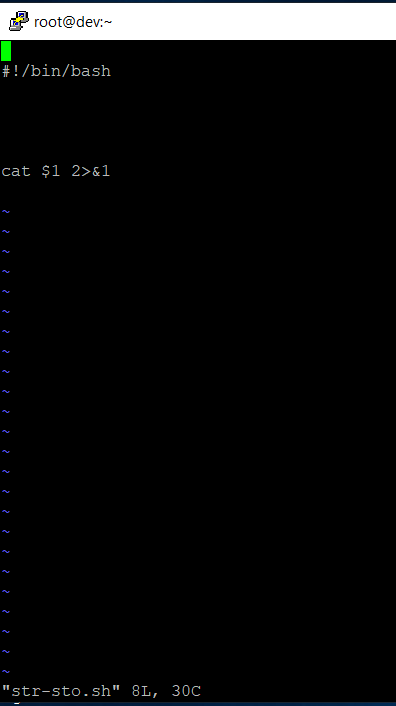
### In the following examples, we will show how to redirect standard error and standard output.

STDOUT from bash script to STDERR



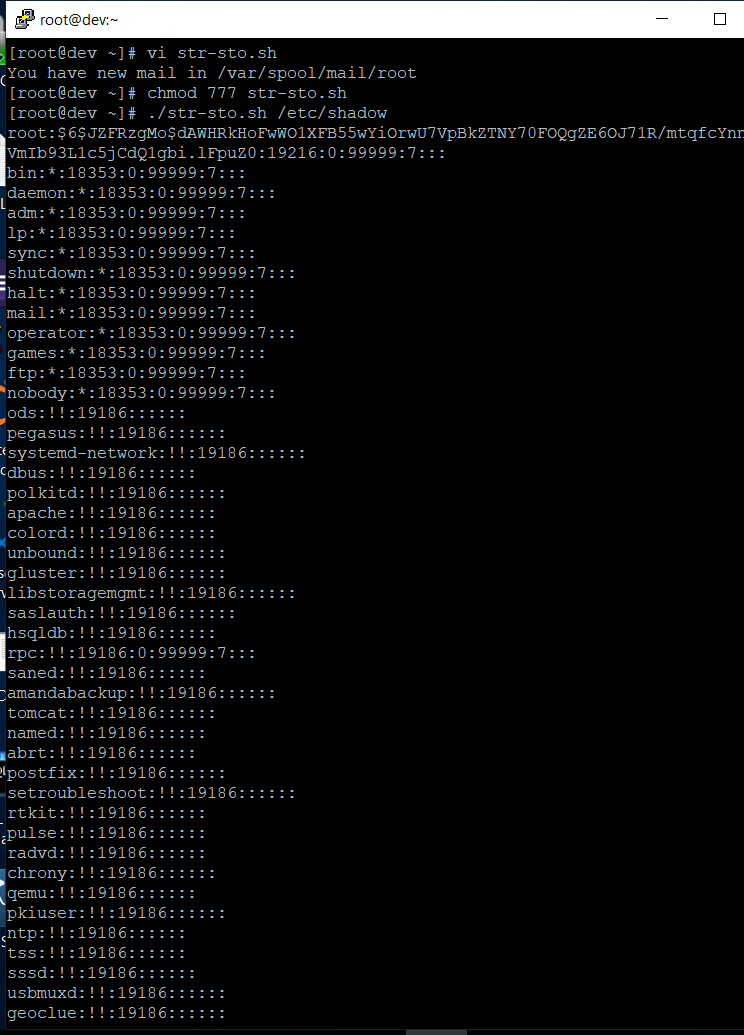


STDERR from bash script to STDOUT



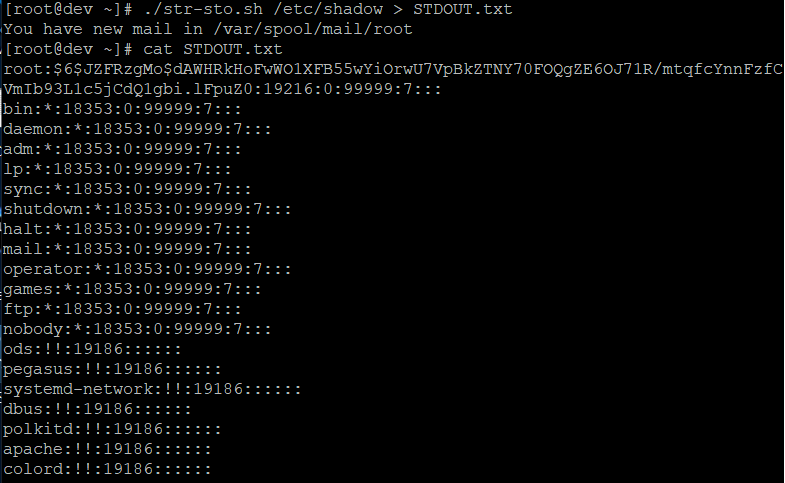
### To prove that STDERR is redirected to STDOUT we can redirect script’s output to file:

$ ./ste-sto.sh /etc/shadow



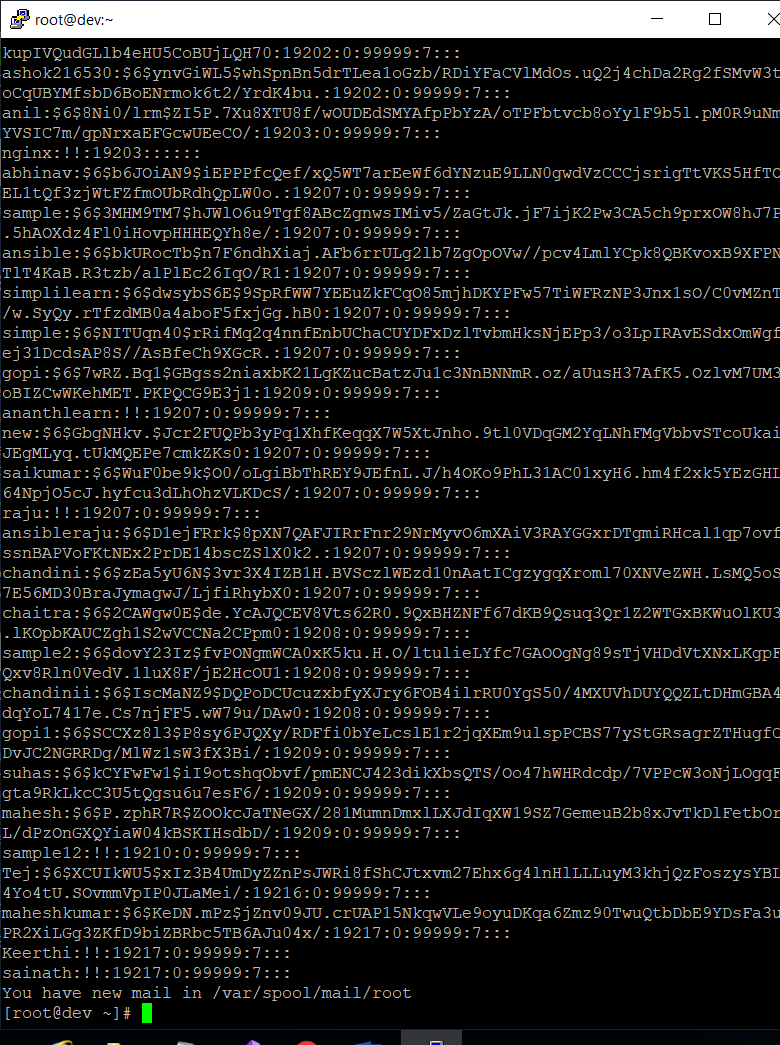
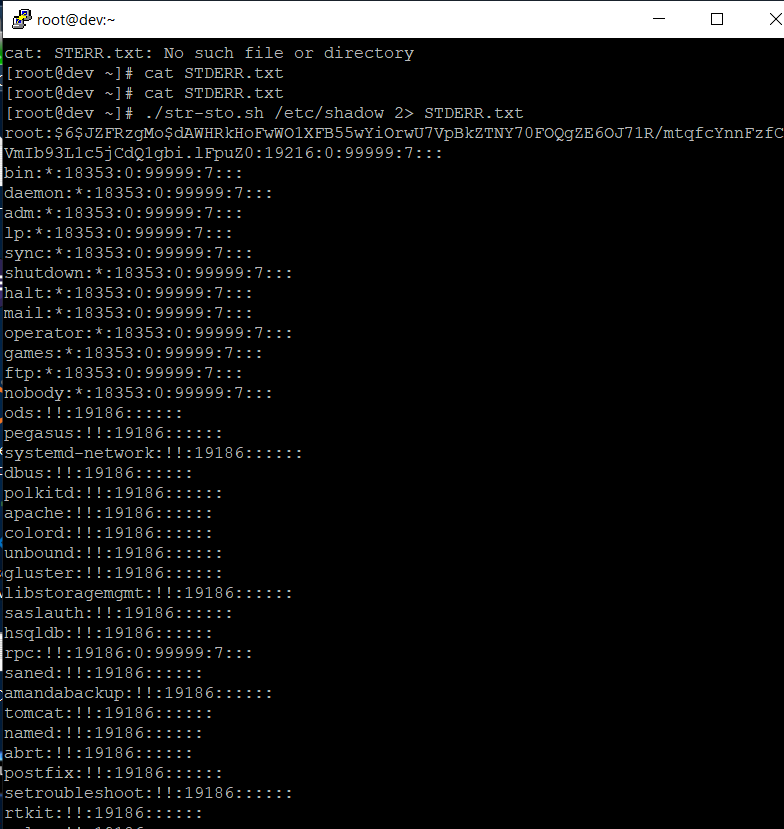
**$ ./str-sto.sh /etc/shadow > STDOUT.txt**

**$ cat STDOUT.txt**



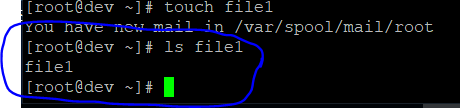
**$ ./redirecting.sh /etc/shadow 2> STDERR.txt**

**$ cat STDERR.txt**



## stdout to screen

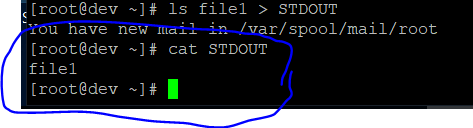
The simple way to redirect a standard output (stdout) is to simply use any command, because by default stdout is automatically redirected to screen. First create a file **file1**:



### As you can see from the example above execution of **ls** command produces STDOUT which by default is redirected to screen.

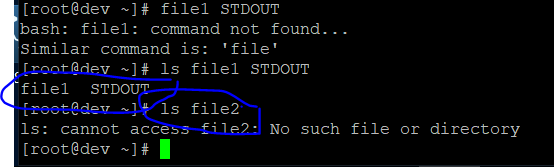
stdout to file

To override the default behavior of STDOUT we can use **>** to redirect this output to file:

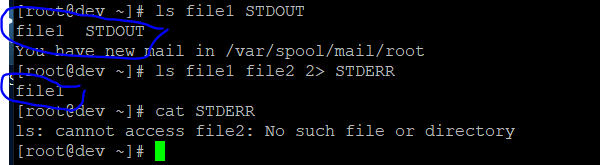


## **stderr to file**

### **By default STDERR is displayed on the screen:**

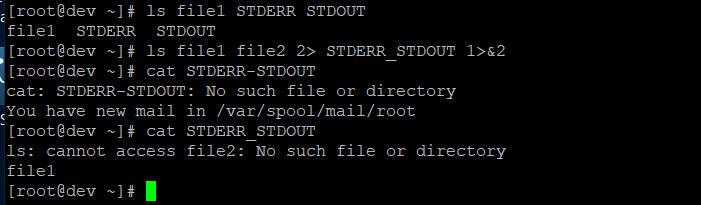


### In the following example we will redirect the standard error (stderr) to a file and stdout to a screen as default. Please note that STDOUT is displayed on the screen, however STDERR is redirected to a file called STDERR:



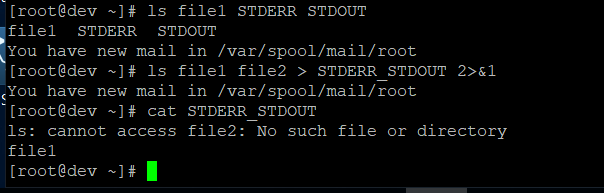
## stdout to stderr

### It is also possible to redirect STDOUT and STDERR to the same file. In the next example we will redirect STDOUT to the same descriptor as STDERR. Both STDOUT and STDERR will be redirected to file “STDERR\_STDOUT”.



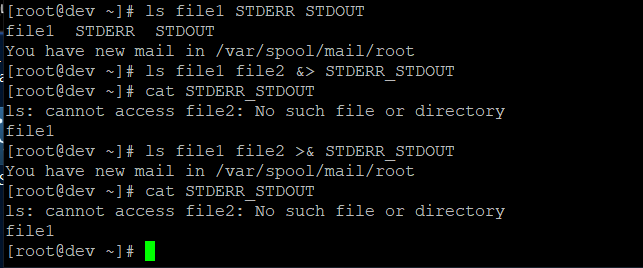
stderr to stdout

The above example can be reversed by redirecting STDERR to the same descriptor as SDTOUT:



## stderr and stdout to file

### Previous two examples redirected both STDOUT and STDERR to a file. Another way to achieve the same effect is illustrated below:



THE END