Shell Tips!

Math Arithmetic: How To Do Calculation In Bash?

Last updated: 2022-07-16

On This Page

Math and **Arithmetic** are often used in **Bash** scripting, especially when writing crontab reports, monitoring plugins, setting up scripts with dynamic configurations, or other automation tasks like showing a Raspberry PI CPU temperature. There is always some **arithmetic calculation** to be made.

This post covers how to do basic mathematical operations (elementary arithmetic like multiplication and addition) in Bash with ***integers*** or ***floating-point*** numbers.

fraction or decimal and is anywhere from zero to positive or negative infinity. For example, 42, 36, and -12 are integers, while 3.14 and √2 are not. The set of integers includes all negative integers (whole numbers that has value less than zero), zero, and all positive integers (whole numbers that has value larger than zero). An integer and its opposite are the same distance from zero.

A floating-point number is a computing programming term to represent a real number with a fractional part. For example 3.14, √2, -10.5, 4^{e-2} are floating points numbers. A floating-point is specified by a base (binary or decimal), a precision, and an exponent range. It is usually in binary made of 32 (simple precision) or 64 bits (double precision), thought the IEEE 754 standard mention more formats.

Format	Total bits	Significand bits	Exponent bits	Smallest number
Single precision	32	23 + 1 sign	8	≈ 1.2 · 10-38
Double precision	64	52 + 1 sign	11	≈ 2.2 · 10-308

You may see a floating-point being represented in the form of ***significand x base*** ^{exponent}. For example: $3.14 = 314 \times 10^{-2}$

What are the Bash Arithmetic Operators?

id++, id–	variable post-increment, post-decrement
++id, –id	variable pre-increment, pre-decrement
-, +	unary minus, plus
!, ~	logical and bitwise negation
**	exponentiation
*, /, %	multiplication, division, remainder (modulo)
+, -	addition, subtraction
«, »	left and right bitwise shifts
<=, >=, <, >	comparison
==, !=	equality, inequality
&	bitwise AND
^	bitwise XOR
|	bitwise OR
&&	logical AND
||	logical OR
expression ? expression : expression	conditional operator
=, *=, /=, %=, +=, -=, «=, »=, &=, ^=, |=	assignment

Doing Math in Bash with Integer

Natively, Bash can only do **integer arithmetic**, if you need to do **arithmetic operations** that requires **floating-point arithmetic**, see the next section.

Shell Tips!

systems depending on the implementation.

```
# Subtraction
[me@linux ~]$ expr 1 - 1
0
# Addition
[me@linux ~]$ expr 1 + 1
2
# Assign result to a variable
[me@linux ~]$ myvar=$(expr 1 + 1)
[me@linux ~]$ echo $myvar
2
# Addition with a variable
[me@linux ~]$ expr $myvar + 1
3
# Division
[me@linux ~]$ expr $myvar / 3
0
# Multiplication
[me@linux ~]$ expr $myvar \* 3
6
```

> ⚠ When doing a *multiply by* make sure to escape the asterisk (*), or any other *bash wildcards*, to prevent pattern expansion in Bash. You can escape a special character using the backslash (\ ), example: `expr $myvar \* 3` . Not escaping the `*` would lead to an `expr: syntax error` .

Using the *let* or *declare* shell builtin commands

```
[me@linux ~]$ let myvar+=1 ; echo $myvar
7
[me@linux ~]$ let myvar+1 ; echo $myvar
8
[me@linux ~]$ let myvar2=myvar+1 ; echo $myvar2
9
```

With both methods, you can combine multiple expressions on a single line as `let` and `declare` evaluate each argument as a separate arithmetic expression.

```
[me@linux ~]$ let x=4 y=5 z=x*y u=z/2
[me@linux ~]$ echo $x $y $z $u
4 5 20 10

[me@linux ~]$ declare -i x=4 y=5 z=x*y u=z/2
[me@linux ~]$ echo $x $y $z $u
4 5 20 10
```

> ⚠ Integer Declaration using the `declare -i` notation can lead to confusing or hard-to-read shell scripts. A variable set as an integer may accept non-integer values, but it won't store and output the expected new string and may error out. Using `declare -i` force a variable to be an arithmetic context only. It is like prefixing the variable assignment with the `let` command every time. Instead, it is generally clearer to use the Bash Arithmetic Expansion as detailed in the next section.

Using the *Bash Arithmetic Expansion*

The recommended way to evaluate arithmetic expressions with integers in Bash is to use the **Arithmetic Expansion** capability of the shell. The builtin shell expansion allows you to use the parentheses `((...))` to do math calculations.

The format for the Bash arithmetic expansion is `$(( arithmetic expression ))`. The shell expansion will return the result of the latest expression given.

The `$((...))` notation is what is called the *Arithmetic Expansion* while the `((...))` notation is called a *compound command* used to evaluate an *arithmetic expression* in Bash.

The *Arithmetic Expansion* notation should be the preferred way unless doing an arithmetic evaluation in a Bash if statement, in a Bash for loop, or similar statements.

> 👉 The square brackets `$[...]` can also do Arithmetic Expansion in Bash, though this notation has been deprecated and should be avoided. Instead, prefer the use of `$((...))` instead.

```
[me@linux ~]$ myvar=3 && echo $myvar
3
[me@linux ~]$ echo $((myvar+2))
5
[me@linux ~]$ myvar=$((myvar+3))
```

arithmetic operators. All the operators listed in the table above are fully available when using **Arithmetic Expansion**.

```
[me@linux ~]$ myvar=3 && echo $myvar
3
[me@linux ~]$ echo $((myvar++))
3
[me@linux ~]$ echo $myvar
4
[me@linux ~]$ echo $((++myvar))
5
[me@linux ~]$ echo $myvar
5
```

In the previous section, we show an example with `let` containing multiple expressions on a single line. This is also possible with *Arithmetic Expansion*. The only difference is that multiple expressions must be separated by a comma (`,`).

```
[me@linux ~]$ echo $((x=4,y=5,z=x*y,u=z/2))
10
[me@linux ~]$ echo $x $y $z $u
4 5 20 10

[me@linux ~]$ ((x=4,y=5,z=x*y,u=z/2)) ; echo $x $y
$z $u
4 5 20 10
```

Doing Floating-point Arithmetic in Bash

Using the *printf* builtin command

~~$((2/3)) would return 0. To get the floating number using~~ `printf` you would use a formula like below where `<precision>` would be the floating-point precision to display and `<multiplier>` the power of ten multipliers. Similarly, a multiplier of 3 would mean `10**3`, which is `1000`.

```
printf %.<precision>f "$((10**<multiplier> * 2/3))e-<multiplier>
```

Note that the floating point precision in `%.<precision>f` shouldn't be higher than the multiplier itself as it will just fill with zeros.

```
[me@linux ~]$ printf %.3f "$((10**3 * 2/3))e-3"
0.666
[me@linux ~]$ printf %.1f "$((10**3 * 2/3))e-3"
0.7
[me@linux ~]$ printf %.5f "$((10**3 * 2/3))e-3"
0.66600
```

Using the *awk* command line

Another way to do floating-point arithmetic is to use *GNU awk*. You can use all the arithmetic operators listed in the table earlier in this post and the `printf` function to adjust the precision of the printed results.

```
[me@linux ~]$ awk "BEGIN {print 100/3}"
33.3333
[me@linux ~]$ awk "BEGIN {x=100/3; y=6; z=x*y; print z}"
200
```

```
awk: cmd. line:1:                    ^ syntax
error
```

```
[me@linux ~]$ awk "BEGIN {print -8.4 - -8}"
-0.4
```

Using the *bc* command line

Since you can't do floating-point arithmetic natively in Bash, you will have to use a command-line tool. The most common one is "*bc - An arbitrary precision calculator language*".

To start the interactive mode, you simply need to type `bc` in your command prompt. You can use the `-q` (quiet) option to remove the initial `bc` banner.

```
[me@linux ~]$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free
Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
3*5.2+7/8
15.6
15.6+299.33*2.3/7.4
108.6
```

Of course, you can also use bc in a non-interactive mode by using the `STDIN` to send your formula to `bc` then get the output on `STDOUT`, or by using the here-doc notation.

```
# Example of piped arithmetic expression to the bc
```

> There are four special variables, *scale*, *ibase*, *obase*, and *last*. *scale* defines how some operations use digits after the decimal point. The default value of *scale* is 0. *ibase* and *obase* define the conversion base for input and output numbers. The default for both input and output is base 10. *last* (an extension) is a variable that has the value of the last printed number.

The "*scale*" variable is essential for the precision of your results, especially when using integers only.

> 👉 ***Note:*** you can also use `bc -l` to use *mathlib* and see the result at max scale.

```
[me@linux ~]$ echo "2/3" | bc
0
[me@linux ~]$ echo "scale=2; 2/3" | bc
.66
[me@linux ~]$ echo "(2/3)+(7/8)" | bc
0
[me@linux ~]$ echo "scale=2;(2/3)+(7/8)" | bc
1.53
[me@linux ~]$ echo "scale=4;(2/3)+(7/8)" | bc
1.5416
[me@linux ~]$ echo "scale=6;(2/3)+(7/8)" | bc
1.541666
[me@linux ~]$ echo "(2/3)+(7/8)" | bc -l
1.54166666666666666666
```

How to calculate a percentage in Bash?

You can calculate a floating-point precision percentage in Bash using the `printf` method and Arithmetic Expansion, or you can calculate a rounded integer percentage using Arithmetic Expansion with the `((...))` notation.

The round-up approach leverages the shell behavior to round toward zero (0). We first calculate twice the percentage then subtract the regular percentage from it. This gives us the formula: `roundup = (int) (2 * x) - (int) x` where `x` is the percentage calculation.

```
[me@linux ~]$ fileProcessed=17; fileTotal=96;

# Floating-point precision using builtin printf
method and Arithmetic Expansion
[me@linux ~]$ printf %.2f%% "$((10**3 * 100 *
$fileProcessed/$fileTotal))e-3"
17.71%

# Rounding Up Using Arithmetic Expansion and
Integers only
[me@linux ~]$ echo $((200 *
$fileProcessed/$fileTotal))
35
[me@linux ~]$ echo $((100 *
$fileProcessed/$fileTotal ))
17
[me@linux ~]$ echo $((200 *
$fileProcessed/$fileTotal -  100 *
$fileProcessed/$fileTotal ))%
```

Shell Tips!

a factorial is defined by the formula `n! = (n-1)! * n`.

```
[me@linux ~]$ function f() {
  local x=$1
  if ((x<=1)); then
     echo 1
  else
     n=$(f $((x-1)))
     echo $((n*x))
  fi
}
[me@linux ~]$ f 1
1
[me@linux ~]$ f 2
2
[me@linux ~]$ f 3
6
[me@linux ~]$ f 6
720
[me@linux ~]$ f 8
40320
```

> ⚠ This solution is limited to the maximum value of a numeric shell variable in your environment. You can try to resolve the following arithmetic expression `echo $((2**64))`. If the result is zero, your platform and shell environment won't support beyond the `20 factorial`. Trying to resolve a larger number factorial with this method would lead to inaccurate results. Therefore, you should prefer using `bc` for reliable results without such constraints. For an

arithmetic expression, and a bash variable inference. Every time the `calculator` function is called, it will update a variable name by a given value or by default 1 with a given arithmetic operator. Example: `counter <var_name> <operator> <value>`. You can also implement an interactive version of a simple calculator by using bc.

```
[me@linux ~]$ A=0; B=0;
[me@linux ~]$ calculator() ((${1}=${!1}${3:-
+}${2:-1}))
[me@linux ~]$ echo "Counter A=$A and B=$B"
Counter A=0 and B=0
[me@linux ~]$ calculator A; calculator B
[me@linux ~]$ echo "Counter A=$A and B=$B"
Counter A=1 and B=1
[me@linux ~]$ calculator A 5; calculator B 2
[me@linux ~]$ echo "Counter A=$A and B=$B"
Counter A=6 and B=3
[me@linux ~]$ calculator A 5 /; calculator B 2 "*"
Counter A=1 and B=6
```

Obviously, this example is just to demonstrate some of the concepts to do math while using other bash constructs. For a simple variable assignment, you should prefer to use the assignments bash arithmetic operators.

How to do math on a date using *Arithmetic Expansion* and *printf*?

Below is a simple example of doing a date manipulation with a math subtraction in shell script by using the new `$EPOCHSECONDS` variable from GNU Bash version 5 and

```
[me@linux ~]$ printf 'Previous day of the month
was the %(%d)T\n' $((EPOCHSECONDS-86400))
Previous day of the month was the 09
```

> 👉 Read more on how to manipulate and format dates in ***bash*** in the post How To Format Date and Time in Linux, macOS, and Bash?.

How to use different arithmetic bases in an Arithmetic Expansion?

With the Bash Arithmetic Expansion, you can perform calculations between different arithmetic bases. For example, add a base 10 integer to a base 2 integer. To do so, you can prefix each number with the base identifier and the hashtag character `#`, using the form `base#number`. The base must be a decimal between 2 and 64 representing the arithmetic base. The default base value used in bash arithmetic expansion is the `base 10`.

Note that, you can also prefix numbers with a leading zero `0` to represent octal numbers (`base 8`). A leading `0x` or `0X` would be interpreted as an hexadecimal. Also, the dollar sign `$` is required in front of a variable when specifying a base identifier

```
[me@linux ~]$ echo $(( 10#2 + 2#1 ))
3
[me@linux ~]$ echo $(( 2 + 2#1 ))
3
[me@linux ~]$ echo $(( 10#2 + 16#aa ))
```

> ⚠ Using a base identifier only works with unsigned integers. There is a trick to workaround the issue and move the sign in front of the *base identifier*, see example below. In general, when doing complex math calculation, you should prefer another solution like using the Linux bc command line.

```
[me@linux ~]$ x=-08 ; echo $(( 10#$x ))
bash: -08: value too great for base (error token
is "08")
[me@linux ~]$ echo $(( ${x%%[!+-]*}10#${x#[-+]}
));
-8
```

How to solve the bash error *value too great for base (error token is...*?

As mentioned above, Bash Arithmetic Expression will automatically consider numbers with leading zero as octal numbers (base 8). When a variable is expended to represent a number with leading zeros and composed with numbers equal or above 8, it will lead to a bash arithmetic error like `bash: 08: value too great for base (error token is "08")`.

```
[me@linux ~]$ x=01 ; echo $((x+1))
2
[me@linux ~]$ x=0105 ; echo $((x+1))
70
[me@linux ~]$ x=08 ; echo $((x+1))
```

Though, a simpler solution may be to ensure the variable is using a base identifier for the base 10 instead of the default representation (see question above). You must use the dollar sign `$` in front of a variable when using the base identifier notation.

```
# Example using extended glob
[me@linux ~]$ shopt -s extglob
[me@linux ~]$ x=08 ; x=${x##+(0)}; echo $((x+1))
9

# Example using a base 10 identifier
[me@linux ~]$ x=08 ; echo $((10#$x+1))
9
```

If you are getting this error when using *signed numbers*, then refer to the previous question as the *base identifier* only works with *unsigned numbers*.

How to solve the *syntax error: invalid arithmetic operator*?

When using *Arithmetic Expansion*, you may encounter the following error:

```
syntax error: invalid arithmetic operator (error
token is ...)
```

This error is generally due to improperly formatted variables or integers used in the arithmetic expression. The most common mistake would be to try to use a floating-point number, which would fail as such.

expansion.

```
[me@linux ~]$ a=$' 3\r'
[me@linux ~]$ echo "$((a+7))"
")syntax error: invalid arithmetic operator (error
token is "
```

You will notice that in such cases, the error message even gets mangled due to the carriage return character `\r` in the variable. To prevent such issues, you will want to ensure the variables you use are properly formatted by stripping out control characters, mainly those with ASCII values from 1 (octal 001) to 31 (octal 037). You can do that by using the shell parameter expansion.

```
[me@linux ~]$ echo "$((${a//[
$'\001'-$'\037']}+7))"
10
```

> 👉 Before Bash version 5, you may need to ensure the right collating order of the ASCII values by using `shopt -s globasciiranges`. Though, since Bash version 5, you don't need to worry about the `globasciiranges` option since it is now set by default. Read more about Bash 5 with the post What's New in GNU Bash 5?

How to solve the bash error *integer expression expected*?

arithmetic operator.

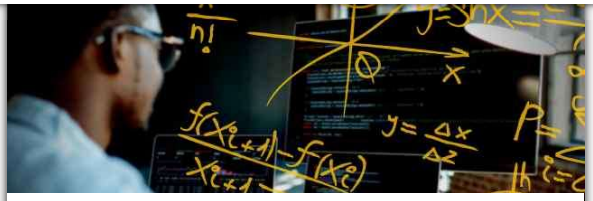
Note that it is best practice to use the bash arithmetic compound expression with actual arithmetic operators instead of the legacy `-lt`, `-gt`, `-le`, and `-ge`.

```
# ERROR
[me@linux ~]$ [ 1 -lt 4.0 ] && echo "1 is smaller than 4.0"
bash: [: 4.0: integer expression expected

# CORRECT
[me@linux ~]$ [ 1 -lt 4 ] && echo "1 is smaller than 4"
1 is smaller than 4

# BEST
[me@linux ~]$ ((1<4)) && echo "1 is smaller than 4"
1 is smaller than 4
```


Shell Tips!

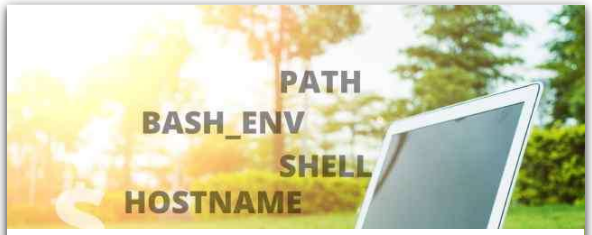


How To Do Advanced Math Calculation Using bc?

How to Parse a CSV File in Bash?

Linux sysctl configuration and tuning script

5 Simple Steps On How To Debug a Bash Shell Script



A Complete Guide to the Bash Environment Variables

How To Make A Custom Bash Shell Prompt

DISCLAIMERS & CONTACTS