Open in app Get started

Published in System Failure

This is your **last** free member-only story this month. Sign up for Medium and get an extra one

Uday Hiwarale Follow
Sep 7, 2019 · 40 min read ★ · Listen

Save

SHELL PROGRAMMING

Bash Scripting: Everything you need to know about Bash-shell programming

In this article, we are going to cover almost every single topic there is in Bash programming. This article mainly focuses on programming spec and not how different UNIX command works.

Last update: *Tuesday, 10 September 2019, 3:12 AM | UTC*

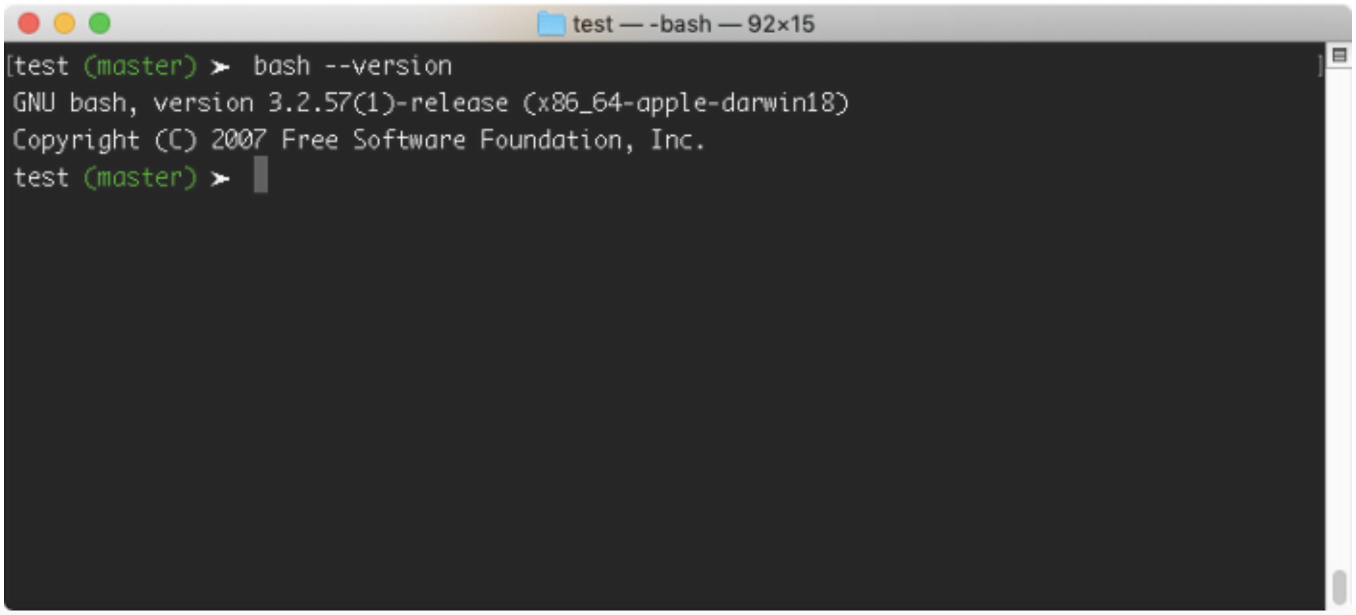
(**source**: pixabay.com)

services to do something. For example, `ls` command lists the files and folders in a directory. Bash is the improved version of **Sh** (*Bourne Shell*). A **shell scripting** is writing a program for the shell to execute and a **shell script** is a file or program that shell will execute.

If you are a programmer, then you might have use commands like `mv` to move or rename a file, `touch` to create a file or `nano` to edit a file. We use these commands in a **terminal** which is the **interface to the shell interpreter**.

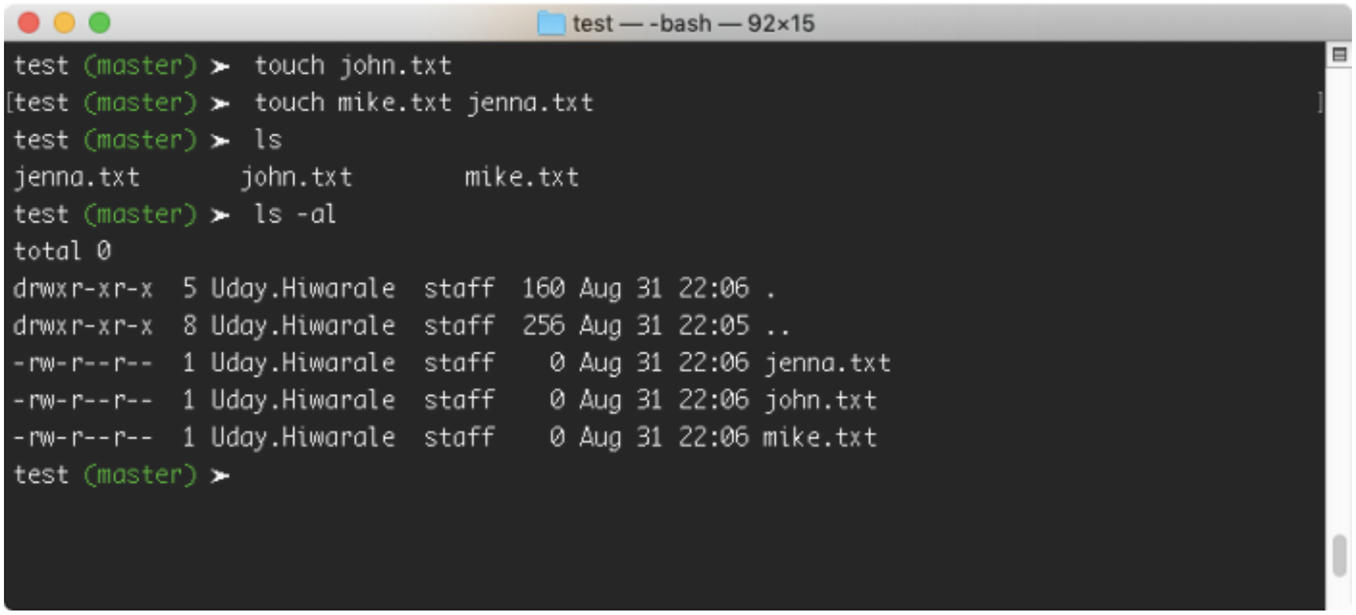
A shell script is a fully-fledged programming language in itself. It can define variables, functions and we can do conditional execution of shell commands as well. Having a terminal at your disposal can save precious time and sometimes GUI of your OS might not provide the necessary tool to perform actions such as executing a binary file with options. And working inside a terminal makes you look like a **geek**, if that's your thing.

Before we begin with the Bash programming, let's go through some common commands we should be using every day.



```
test (master) ➤ bash --version
GNU bash, version 3.2.57(1)-release (x86_64-apple-darwin18)
Copyright (C) 2007 Free Software Foundation, Inc.
test (master) ➤
```

it should print the version of Bash interpreter.



```
test (master) ➤ touch john.txt
[test (master) ➤ touch mike.txt jenna.txt
test (master) ➤ ls
jenna.txt       john.txt        mike.txt
test (master) ➤ ls -al
total 0
drwxr-xr-x  5 Uday.Hiwarale  staff  160 Aug 31 22:06 .
drwxr-xr-x  8 Uday.Hiwarale  staff  256 Aug 31 22:05 ..
-rw-r--r--  1 Uday.Hiwarale  staff    0 Aug 31 22:06 jenna.txt
-rw-r--r--  1 Uday.Hiwarale  staff    0 Aug 31 22:06 john.txt
-rw-r--r--  1 Uday.Hiwarale  staff    0 Aug 31 22:06 mike.txt
test (master) ➤
```

By typing Bash code directly in your terminal, it will get executed on Bash interpreter. In the above example, we have created `john.txt` file using `touch` command and later `mike.txt` and `jenna.txt` at once.

`touch` is a binary executable file located inside `/bin` directory of your system (*use command `which touch` to see its location*). When shell receives `touch john.txt`, it executes `touch` file with `john.txt` as the command line option and `touch` creates `john.txt` in the current directory of the terminal.

`ls` is also a command to list files in a directory (*current directory if no directory path argument is provided*). We can use `ls` because it is a binary executable file located inside `/bin`. We have used `-al` flags (*or `-a` and `-l`*) separately, to list all files and directory with more information. This will also show `.` (*current directory path*) and `..` (*parent directory path*).

When you enter a command like `ls` or `touch`, the shell interpreter first checks for an alias present in `.bash_profile` file in your home directory. If alias is not present, then it looks for

> *rename a command, for example, we can set up* `list` *command as an alias for* `ls`*. We will learn about all this later.*

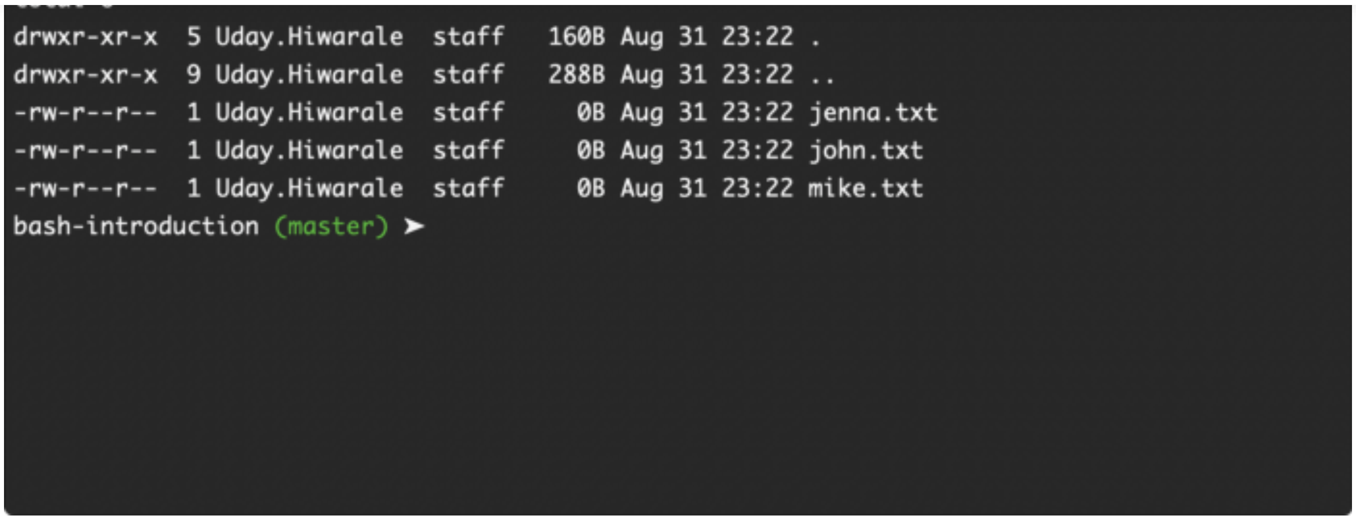
There are other important commands like `mv`, `nano`, `cp`, `top`, etc. which are very important. But these are beyond the scope of this tutorial. In this article, we are going to learn about Bash programs and not how different command works. But I am planning to write a tutorial on different UNIX commands.

To execute multiple Bash commands and execute them at once, we need to save these commands in a file and execute that file with `bash`.

```
# make-files.txt
touch test/john.txt
touch test/mike.txt
touch test/jenna.txt

# list files from `./test` directory
ls -alh ./test
```

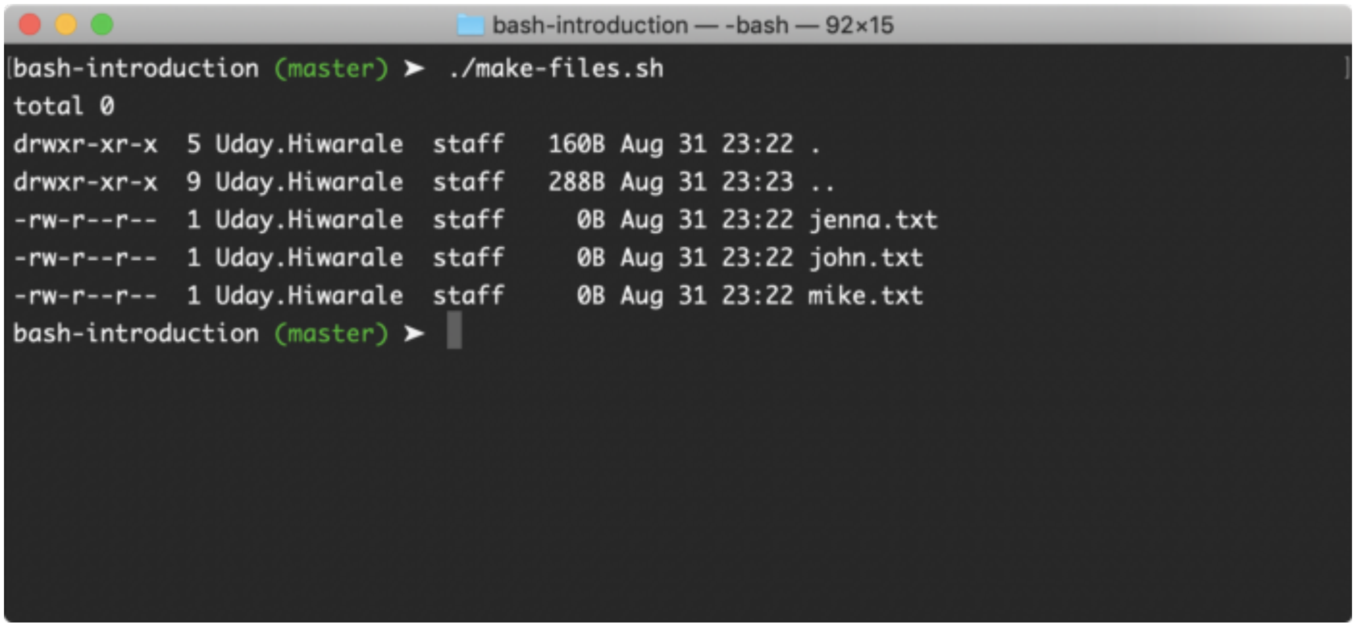
From the above example, we have created `make-files.txt` file which contains multiple Bash commands in order. Bash interpreter will execute these commands one at a time and ignore commands that start with `#` as they are comments. This will produce the below result.



```
drwxr-xr-x  5 Uday.Hiwarale  staff   160B Aug 31 23:22 .
drwxr-xr-x  9 Uday.Hiwarale  staff   288B Aug 31 23:22 ..
-rw-r--r--  1 Uday.Hiwarale  staff     0B Aug 31 23:22 jenna.txt
-rw-r--r--  1 Uday.Hiwarale  staff     0B Aug 31 23:22 john.txt
-rw-r--r--  1 Uday.Hiwarale  staff     0B Aug 31 23:22 mike.txt
bash-introduction (master) ➤
```

Normally, a Bash script file has `.sh` extension to make it clear that it is a shell script file. However, we can directly execute it like a binary but we need to put a **shebang** or **hashbang** line at the top of the file **to declare the interpreter**.

```
#! /usr/bin/env bash
# comment: use command ./make-files.sh
ls -alh ./test
```



```
bash-introduction (master) ➤ ./make-files.sh
total 0
drwxr-xr-x  5 Uday.Hiwarale  staff   160B Aug 31 23:22 .
drwxr-xr-x  9 Uday.Hiwarale  staff   288B Aug 31 23:23 ..
-rw-r--r--  1 Uday.Hiwarale  staff     0B Aug 31 23:22 jenna.txt
-rw-r--r--  1 Uday.Hiwarale  staff     0B Aug 31 23:22 john.txt
-rw-r--r--  1 Uday.Hiwarale  staff     0B Aug 31 23:22 mike.txt
bash-introduction (master) ➤
```

It is important to note that when we run a script file in the terminal, **terminal issues a new session for the script to run**. Hence any variables present in the current session of the terminal (*current scope*) won't be accessible inside the script we are running. We will also learn later how we can avoid that.

Declaring variables and introduction to `echo` and `read` commands

We can declare variables in a Bash script. Unlike other programming languages, we don't need to declare with `var` keyword or data type like `int` .

To declare a variable and assign with a value, use `VARIABLE_NAME=VALUE` expression (*with no spaces in between*).

```
CITY=New_York
NAME="John Doe"
AGE=26
EMAIL='john@doe.com'
GRADE=8.43
EMPLOYED=true
```

In the above example, we have declared a few variables with different data types. But **Bash does not have a type system**, it can only save **string** values. Hence internally, Bash saves them as a string.

> *However, based on operations, Bash might convert them to a suitable data type on the fly. We will also learn about this in **Arithmetic Operations** topic.*

In the first statement, we have set `CITY` with the value **New York** with an **underscore** as the separator. Else, Bash would treat `CITY=New` **`York`** statement as a command, where `York` is the executable binary (*or a command*) and it will set `CITY` environment variable with

```
echo Hello World

⇒ Hello World
```

In the above example, `Hello World` is the output in the terminal. It doesn't matter if run this program using a script file or directly in the terminal. `Hello` and `World` are the two string **arguments** for the `echo` command. `echo` **adds a space automatically** between two adjacent arguments (*in the output*) by default.

> *From now on,* ⇒ *will represent the terminal output and* ⇐ *as a user input.* `~$` *symbol will be used to represent command input line on the terminal.*

To print a variable declared in the script, we need to put `$` as the prefix for the `VARIABLE_NAME` like `$VARIABLE_NAME`.

```
echo $NAME $AGE $EMAIL $GRADE $EMPLOYED

⇒ John Doe 26 john@doe.com 8.43 true
```

We can **read** user input in the terminal using `read` command. When the user input the text and hit enter, that entire text will be saved in a variable.

```
echo -n "Enter a name:"
read NAME
echo "Your name is:" $NAME

⇒ Enter a name: ⇐ John Doe
⇒ Your name is: John Doe
```

From the above program, we used `echo` command with `-n` flag, this will prevent adding a

> 💡 `-n` *is not supported in* `sh`. *You can use* `\c` *to prevent* `echo` *command adding a new line, for example,* `echo “Enter a name:\c”`.

`read` command blocks the script execution until the user presses enter in the terminal. When **Enter a name** message appears, we enter `John Doe` and press enter. The output will be `Your name is: John Doe`.

If a command outputs something to the **STDOUT**, we can save it in a variable. `$(command)` expression will execute the command and returns the output of the `command`. We can save this output in a variable. We can also wrap a command in backticks(`` ` ``) which Bash will execute it and return the output.

```
PWD=$(pwd)
BASH_VERSION=`bash --version`
echo $PWD
echo $BASH_VERSION

⇒ /Users/Uday/bash-introduction
⇒ GNU bash, version 3.2.57(1)-release (x86_64-apple-darwin18)
Copyright (C) 2007 Free Software Found
ation, Inc.
```

> 💡 *The* ***newline*** *in Bash is a* ***command separator*** *but you can also use* `;`. *If you want to* ***write multiple commands on a single line****, then use* `;` *as the command separator. For example,* `echo $PWD; echo $BASH_VERSION`.

String interpolation

We can concatenate two variable by just placing them side by side.

```
FIRST_NAME='John'
LAST_NAME='Doe'
echo $FIRST_NAME$LAST_NAME
```

Bash also supports `+=` operator to concatenate two strings.

```
NUMBER=1
NUMBER+=2          # 1+2 => 12
NUMBER+=$NUMBER    # 12+12 => 1212
echo $NUMBER

⇒ 1212
```

In the above example, we first saved `1` in `NUMBER` variable and concatenate it with `2`. As we know everything saved in a variable will be a string, hence we should get `12`. Then we concatenate `NUMBER` with itself and got `1212`.

We can also **substitute a variable in a string defined by double-quotes** by directly putting it in with `$` prefix as we would use outside or with `${VARIABLE_NAME}` expression (*no spaces inside it*). We can also use this expression outside the string if we want.

```
FIRST_NAME='JOHN'
LAST_NAME='DOE'

# result: "JOHN DOE"
FULL_NAME="$FIRST_NAME ${LAST_NAME}"

# ${VAR} expression outside a string
echo Hello ${FULL_NAME}!

⇒ Hello, JOHN DOE!
```

Bash supports both single-quotes (‘) and double-quotes (“) to define a string. But if we need to put single or double quote as a character in a string, we need to escape it with a backslash character.

```
⇒ I am 'John' and I am "AWESOME".
⇒ "Sorry" for that
```

Even though Bash support single-quotes (‘) and double-quotes (“) to define a string, **only double quotes are capable of string interpolation**. If we tried string interpolation in single-quotes like `'${VAR_NAME}'` , it will treat all the characters like normal characters and return it as is.

```
FIRST_NAME='JOHN'
echo "Hello ${FIRST_NAME}!;"
echo 'Hello ${FIRST_NAME}!;'

⇒ Hello JOHN!;
⇒ Hello ${FIRST_NAME}!;
```

⚠️ CAUTION ⚠️

When we reference a string variable and pass it as an argument to a command like `echo` , Bash **breaks the string into different words** (*which were separated by a* ***space***) and pass them as individual arguments. This can be useful in certain scenarios like `for-in` loop.

Here is a small example of how Bash does it. I have a small command `args` that prints the number of arguments passed to it. I have made it using Bash **function** and `$#` positional parameter (*which will be covered in this article*).

```
~$ NAME="John Doe Jr"
~$ args $NAME

⇒ 3
```

As you can see from the above result, the number of arguments is three. Which means by using **string interpolation**. We just need to wrap a variable in double-quotes.

```
MESSAGE="I am a rock *"
echo $MESSAGE
echo "$MESSAGE"

⇒ I am a rock a variable.sh strings.sh numbers.sh
⇒ I am a rock *
```

In bash, `*` is a special character that represents a wildcard. **When passed as an argument** to a command, `*` contains all the files in the current directory as a list. From the above example, with string interpolation, we are passing one string to the `echo` command which contains `*` as well in it and it's not a separate argument anymore.

💡 **Bonus**: You can use `exit` command to stop the program.

```
echo "I am 'John' and I am '\"AWESOME\"."
exit # exit program
echo '"Sorry" for that'

⇒ I am 'John' and I am '"AWESOME".
```

Arithmetic Operations

We can perform arithmetic operations in Bash even though Bash does not support `number` data type. Let's see different mechanisms through which we can perform arithmetic operations.

1. Using `let` command

We can perform simple arithmetics calculations with `let` command.

```
⇒ 2
```

In the above example, we declared the variable `RESULT` with `let` which will evaluate the value `1+1` (*which is a string*). If we write like `RESULT=1 + 1`, then `+` will be a command as discussed before. Hence, a safe way is to write the expression in **quotes** which allows adding spaces.

```
let RESULT="1 + 1"
echo $RESULT

⇒ 2
```

Since `let` would evaluate the expression in a string. So if you want to use **interpolation** to generate a string, it's possible. Now we know that, based on the operation, Bash will convert string to appropriate data type.

We can use any arithmetic operation like `+`, `-`, `*` or `/`. We can also calculate the modulus using `%` operator. We can also **increment** or **decrement** a variable using `VAR++` , `++VAR`, `VAR--` or `--VAR` expression.

```
NUMBER=1
let RESULT="++NUMBER"
echo $RESULT
echo $NUMBER

⇒ 2
⇒ 2
```

We can also declare a variable inside the expression using `let` or perform an arithmetic operation. Using this, you can also add **spaces** in the expression.

```
echo $RESULT

⇒ 26
```

We can also use `+=` , `-=` , `*=` , `/=` and `%=` operators.

```
NUM=5 #5
let NUM-=2       # 5 - 2 => 3
let NUM*=2       # 3 * 2 => 6
let NUM/=3       # 6 / 3 => 2
let NUM+=1       # 2 + 1 => 3
let NUM%=2       # 3 % 2 => 1

echo $NUM

⇒ 1
```

2. Using `expr` command

We can also use **`expr`** command to execute an expression which takes multiple arguments (*it concatenates them and executes it*). It does not have a problem like `let` to declare a variable but it will **print the result by default**, hence we need to use `$(expression)` syntax.

```
expr 1 + 1   # prints to the STDOUT

RESULT=$(expr 3 \* 3)
echo $RESULT

⇒ 2
⇒ 9
```

In the above example, `expr` will execute `1+1` expression and print it as we supplied three arguments with it. In the second line, we evaluated the `expr 3 \* 3` expression using and

```
expr "1 + 1"
expr "1 + 1" + 1
expr "11" + 1

⇒ 1 + 1
⇒ expr: not a decimal number: '1+1'
⇒ 12
```

The above example is very intuitive but simple. At the line no. 1, we are passing only one argument to `expr` the command, hence it will just output it. At the line no. 2, we are passing three arguments, but first argument **can not be interpreted as a decimal number**.

3. Using double parentheses

Using `$((expression))` syntax, we can also perform arithmetic operations. We can put spaces between the `expression` which is valid.

```
RESULT=$((1 + 1))
echo $RESULT

⇒ 2
```

This is **my favorite** syntax for arithmetic expressions as it provides flexibility to print out the result or execute an expression without printing it. But you can't use an arithmetic expression in a string, like `RESULT=$(("1 + 1"))`.

With this syntax, we can also use `+=`, `-=` , `*=`, `/=` and `%=` operators.

```
NUM=5 # 5
((NUM-=2)) # 5 - 2 => 3
(( NUM*=2 )) # 3 * 2 => 6
```

```
⇒ 1
```

As we can see in the above example, if we do not want to store or print the result, we can drop `$` and use `((expression))` syntax to perform and arithmetic operation.

We can manipulate variables directly with this syntax.

```
NUM=5
(( NUM += NUM )) # 5 + 5 => 10
(( NUM++ )) # 6++ => 11
(( NUM = $NUM - 3)) # 11 - 3 => 8

echo $NUM

⇒ 8
```

if/else conditions in Bash

Before we dive into **if/else** syntax, let's recap an **if/else** statements we studied in any of your favourite programming languages. When we write `if` keyword, it is followed by an **expression** that evaluates either `true` or `false` **or** a value which is either `true` or `false` (*some languages also supports truthy or falsy values like* `0` *or* `1`).

If an **expression** is used, it is a job of runtime or an interpreter to evaluate the expression and return `true` or `false`. In bash, `test` command is used to evaluate an expression. This command accepts a series of arguments which forms an expression when combined.

When an expression is evaluated, it terminates the command with success `0` exit status code or error `1` exit status code. Based on these values, we can check if an expression is `true` or `false`.

```
~$ test 5 -gt 9
⇒
```

In the above example, all the arguments should be in the order given else `test` would not understand the operation. In this case, we are not sure if the expression `5 -gt 9` is `true` or `false` as nothing was printed to the terminal.

In Bash, `$?` expression **prints the status** of the **last command executed**.

```
~$ echo $?
⇒ 1
```

Our last command was `test 5 -gt 9` and it exited with status `1` which means the expression `5 -gt 9` is `false`.

`test` command also has an alias command `[`. Don't be surprised, it is actually a command. You can check that by `which [` and it should return `/bin/[`. This works exactly like `test` command except it needs the last `]` argument. `]` is not a command, it is just a character which is required by `[` command to make the expression easier to understand.

```
~$ [ 5 -lt 9 ]
~$ echo $?
⇒ 0

~$ [ 5 -gt 9 ]
~$ echo $?
⇒ 1
```

If you have multiple conditional expressions then you can use `&&` and `||` for **AND** and **OR** operations respectively. We can also use parentheses to combine conditional expressions.

```
([ 5 -gt 3 ] && [ 6 -lt 5 ]) || ([ 3 -gt 1 ] && [ 6 -gt 5 ])
echo "Result 2: $?"

⇒ Result 1: 1
⇒ Result 2: 0
```

> 💡 `(command)` *syntax runs the* ***command*** *in a subshell which is another shell process started by the main shell. If possible, it should be* ***avoided*** *as it will hamper the performance of our script. Read more about subshell* ***here****.*

Now that we know how bash evaluates a **conditional expression**, we can jump into **if/else** syntax. The keywords used in if/else syntax is `if` , `then` , `elif` , and `fi` . Don't be overwhelmed by this, this is actually pretty simple.

1. `if/else` block

To write if/else statement, we need to use `if/then/else` blocks. `[` command is preferred to evaluate test conditions and it makes code easier to read.

```
if [ conditional expression ]
then
    # if code here
else
    # else code here
fi
```

In `then` and `else` block, **indentation is not necessary** (*like in python*) but it makes our code easier to read. Any type of `if` block must end with `fi` to represent the end. Above example prints below result to the terminal.

```
⇒ 5 is greater than 3
```

```
MY_VALUE=3;
echo -n "Enter a number: "; read USER_VALUE;

if [ "$USER_VALUE" -gt "$MY_VALUE" ]; then
    echo "You are great."
else
    echo "You are not so great."
fi

⇒ Enter a number: ⇐ 10
⇒ You are great
```

> ⚠️ *If you can notice from the above example, we have wrapped* `$USER_VALUE` *inside* ***double-quotes*** *to prevent Bash from expanding its string value into different words and pass them as arguments, as learned from the previous* ***CAUTION*** *warning.* `[` *command needs only 3 arguments, and to safely execute it, it is good practice to wrap the variables inside single or double quotes.*

2. `if` **block (only)**

Of course, you can write `if` block without `else` block.

```
PROCEED=YES

if [ "$PROCEED" = "YES" ]
then
    echo "Performing task..."
fi

⇒ Performing task...
```

Before we move on to more complex `if/else` flavours, let's understand what are the different comparison operators.

https://gist.github.com/thatisuday/d160535f07cea34e949da5a89015fe8a

```
# string comparison
if [ 'proceed' == "proceed" ]; then echo "Performing task..."; fi
if [ 'Hello' != "hello" ]; then echo "Hello is not hello"; fi
if [ 'A' \< "a" ]; then echo "A is lower than a"; fi
if [ 'b' \> "a" ]; then echo "b is greater than a"; fi
if [ -z '' ]; then echo "String is empty"; fi
if [ -n 'something' ]; then echo "String is not empty"; fi
if [ 1 -le 1 ]; then echo "1 is less than equal to 1"; fi
if [ 3 -ge 2 ]; then echo "3 is greater than equal to 2"; fi

⇒ Performing task...
⇒ Hello is not hello
⇒ A is lower than a
⇒ b is greater than a
⇒ String is empty
⇒ String is not empty
⇒ 1 is equal to 1
⇒ 1 is not equal to 2
⇒ 0 is less than 1
⇒ 1 is less than equal to 1
⇒ 3 is greater than equal to 2
```

We need to escape `<` and `>` charcters as they are special characters. There are other comparison operators used for files as well, **here** is the list.

> 💡 *We can negate a condition (negative of the condition output) by putting* `!` *as the first character in* `[]` *like* `if [ ! condition ]`*, or between* `if` *and* `[]` *like* `if ! [condition]`*. But try to avoid this as you can achieve this with operators.*

3. `if-elif-else` **block**

Unlike other programming languages `else if` block is written with `elif` keyword. The pattern for `elif` block is the same as `if` block.

```
VALUE=-10

if [ "$VALUE" -lt 0 ]; then
    echo "VALUE is less than 0"
elif [ "$VALUE" -eq 0 ]; then
    echo "VALUE is 0"
else
    echo "VALUE is greater than 0"
fi

⇒ VALUE is less than 0
```

```
VALUE=10

if [ "$VALUE" -lt 0 ]; then
    echo "VALUE is less than 0"
else
    echo "VALUE is greater than 0"

    if [ "$VALUE" -le 10 ]; then
        echo "VALUE is less than or equal to 10"
    else
        echo "VALUE is greater than 10"
    fi # end of nested if block
fi # end of parent if block

⇒ VALUE is greater than 0
⇒ VALUE is less than or equal to 10
```

Generated Numbers in Bash

1. Random number generation

To generate a 16-bit random number, Bash provides a `RANDOM` global variable that prints a random number between `0` to `32767` every time we access it.

```
echo $RANDOM $RANDOM $RANDOM $RANDOM

⇒ 13044 17964 11391 25892
```

To get a random number between `0` to `n`, we need to use the modulus calculation. As `number % n` will always return a value between `0` and `n-1`, we can get a random number between `0` to `n` by finding the modulus of `n+1` with a random number.

```
⇒ 7
```

2. Sequence (list) generation

To generate a sequence of numbers between `0` to `n`, we can use `{0..n}` syntax. This will generate a sequence from `0` to `n` number with the **step** of `1`. We can also use `seq` command which provides custom step increment.

```
echo {0..10} # step: 1
echo $(seq 0 10) # step: 1
echo $(seq 0 2 10) # step: 2
echo $(seq 10 -2 4) # step: -2

⇒ 0 1 2 3 4 5 6 7 8 9 10
⇒ 0 1 2 3 4 5 6 7 8 9 10
⇒ 0 2 4 6 8 10
⇒ 10 8 6 4
```

> 💡 *Generated sequences are very useful in loops like in* `for:in` *iteration.*

Loops in Bash: `for`, `while` and `until` loops

Bash supports `for` loops, `while` loops and `until` loops. The interesting fact about `for` loop in Bash is, it supports Python-like syntax to iterate on a list of items as well as JavaScript type `from:to` syntax. Let's jump into it.

1. The `for` loops

The basic syntax for a `for` loop in Bash is as following.

```
    # perform action per iteration
done
```

As per the above example, `for` , `do` and `done` are the required keywords. The `iterator` can be different based on which flavour of `for` loop we are using.

Let's first look into `for:in` flavour which resembles Python's for loop syntax.

```
for VAR_NAME in Hello World * Nice to meet you.
```

In the above syntax, `for` is followed by `VAR_NAME` which contains items listed after `in` keyword. This variable is accessible in `do` block.

```
for WORD in "Hello" "World" "*" "Nice" "to" "meet" "you."; do
    echo "The word is: $WORD"
done

⇒ The world is: Hello
⇒ The world is: World
⇒ The world is: *
⇒ The world is: Nice
⇒ The world is: to
⇒ The world is: meet
⇒ The world is: you.
```

In the above example, we have made some modifications to normal existing syntax. First, we wrapped each item with **double quotes** to escape special character like `*` . We can also use `\*` but this is a preferred way.

Also, you can notice that we wrote `do` keyword on the same line of `for` keyword using `;` as the statement separator.

```
FRUITS="Mango Apple Banana"

for FRUIT in $FRUITS; do
    echo "The fruit is: $FRUIT"
done

⇒ The fruit is: Mango
⇒ The fruit is: Apple
⇒ The fruit is: Banana
```

If this is not the desired effect you want, use string interpolation.

```
FRUITS="Mango Apple Banana"

for FRUIT in "$FRUITS"; do
    echo "The fruit is: $FRUIT"
done

⇒ The fruit is: Mango Apple Banana
```

Another interesting fact is, we can drop `in` keyword and ignore to specify the list of items. This is totally legal. In that case, Bash will use arguments passed to the commands which ran the script (*AKA positional parameters*).

```
# file: variable.sh
for FRUIT; do
    echo "The fruit is: $FRUIT"
done

-------------------------------

~$ bash variable.sh Mango Banana Apple

⇒ The fruit is: Mango
⇒ The fruit is: Banana
⇒ The fruit is: Apple
```

```
for FILE in `ls *.json`
do
    echo "Doing something with: $FILE"
done

⇒ Doing something with: users.json
⇒ Doing something with: photos.json
⇒ Doing something with: comments.json
```

> 💡 *Instead of executing* `ls` *command, we can also get the list of* `.json` *files in the current directory using* `*.sh` *pattern, which is the preferred way.*

Now, let's see another and more traditional flavour of the `for` loop we are aware of. We have used this type of `for` loops in `C++` or `Java` or `JavaScript` languages. The syntax in Bash is similar like `for(initialization; comparision; increment/decrement)` but without a variable declaration.

```
for (( i=n; i<N; n++ ))
do
    # perform action per iteration
done
```

Let's consider a simple example.

```
for (( i = 0; i < 5; i++ )); do
    echo "The number is: $i"
done

⇒ The number is: 0
⇒ The number is: 1
⇒ The number is: 2
⇒ The number is: 3
⇒ The number is: 4
```

```
for (( n = 0, i = 1; n < 5; n++, i += i )); do
    echo -n "$i, "
done
echo "" # for a newline

⇒ 1, 2, 4, 8, 16,
```

You can also run `for` loop infinitely with `((;;))` syntax but then you have to break out of the loop manually, like for using **`exit`** command.

```
COUNTER=0

for ((;;))
do
    if [ $COUNTER -eq 5 ]; then
        exit
    else
        echo "Current number is: $(( COUNTER++ ))"
    fi
done

⇒ Current number is: 0
⇒ Current number is: 1
⇒ Current number is: 2
⇒ Current number is: 3
⇒ Current number is: 4
```

2. The `while` loop

The `while` loop in Bash is used to perform the same action **as long as** a condition is `true`. We use `if` style conditional evaluator (*using `[` command*) to evaluate if a conditional expression is `true` or `false`.

```
    # perform action
done
```

`while`, `do` and `done` are mandatory keywords but we can put `while` and `do` keywords on the same line using `;` statement separator. Let's consider a simple example of printing numbers while `n` is less than `5`.

```
NUMBER=1

while [ $NUMBER -lt 5 ]; do
    echo "Number is: $((NUMBER++))"
done

⇒ Number is: 1
⇒ Number is: 2
⇒ Number is: 3
⇒ Number is: 4
```

> 💡 *In Unix-like operating systems,* `true` *and* `false` *are built-in commands. If you check* `which true` *and* `which false` *commands, they will point to their respective binary files.* `true` *command exits with status* `0` *and* `false` *command exits with status* `1`.

We can run a `while` loop infinitely by using `while :` syntax instead of `while [ conditional expression ]` syntax. But as an infinitely running loop goes, we need to break out of the loop manually. You can also use `true` command to run a `while` loop infinitely.

```
NUMBER=1

while :                             # same as: while true
do
    echo "Number is: $((NUMBER++))"

    if [ $NUMBER == 4 ]; then
        exit
```

```
⇒ Number is: 3
```

3. The `until` loop

The `until` loop is similar to while loop, but here `conditional expression` must evaluate to `true` to terminate the loop.

```
NUMBER=1

until [ $NUMBER == 4 ]
do
    echo "Number is: $((NUMBER++))"
done

⇒ Number is: 1
⇒ Number is: 2
⇒ Number is: 3
```

> 💡 *You can also run* `until` *loop infinitely using* `false` *command instead of a conditional expression evaluated by* `[` *command.*

Pattern matching in Bash

If you have used Bash before to move some files of certain extension like `.txt` from one folder to another folder, then you are already familiar with pattern matching. Let's consider we have the following files in a folder.

```
⇒
king.db         photographs.txt    readme.md        users.db
king.txt        photos.db          ring.db          users.txt
photo.db        photos.txt         ring.txt
photo.txt       ping.db            usernames.db
photographs.db  ping.txt           usernames.txt
```

And we want to see only `.db` files from the current directory, we would use the pattern `*.db` to pick files with any name which matched by `*` and `.db` extension (*ending with* `.db` *letters*).

```
~$ ls *.db

⇒
king.db         photographs.db     ping.db        usernames.db
photo.db        photos.db          ring.db        users.db
```

Such patterns are called as `glob` patterns (*short from **global***) and they are supported in all UNIX-like OS. Let's see different flavours of glob patterns supported in Bash.

Open in app Get started

https://gist.github.com/thatisuday/097d38314b9aafd148406220cb0f8ea1

From the above table, it looks like **Glob pattern is not as powerful as Regular Expressions** but it gets the job done. But some shell scripting language might provide some additional features to handle more sophisticated pattern matching.

Bash provides some additional patterns which need to be enabled. `shopt` command (*short for **shell options***) is used to enable shell options. The option we need to look at is `extglob` which means **extended globbing**. To see if this option is enabled, use command `shopt`

Seems like `extglob` is disabled for our shell session. To enable it, we have to use `-s` flag which means **set** or enable.

```
~$ shopt -s extglob
~$ shopt extglob
⇒ extglob         on
```

> 💡 `shopt` *command only enables an option for the current shell session. Hence if you open another terminal, there the shell option will be disabled by default. To disable a shell option, use* `-u` *flag which means* ***unset*** *or disable.*

So what does **extended globbing** in Bash brings to the table? Let's say if we wanted to list files that either have `.db` or `.txt` extensions. What command would we use or **pattern** we would choose?

```
~$ ls *.txt *.db

⇒
king.db  photographs.db  ping.db  usernames.db ...more
```

Now how about files that do not have `.txt` or `.db` extensions? With glob pattern only, it would be much harder or even impossible do match such a request. This is where extended globbing helps us.

```
~$ ls !(*.txt|*.db)
⇒ readme.md
```

In the above example, wrote two patterns `*.txt` and `*.db` side by side separated by a pipe

Let’s create some files with different (*and weird enough*) filenames in a directory so that we can test extended patterns in Bash.

```
~$ ls

⇒
photo.db  photophoto.db   photos.db    photoss.db   photosss.db
photo.txt photophoto.txt  photos.txt   photoss.txt  photosss.txt
```

Here is a list of extended patterns we can use to match files.

https://gist.github.com/thatisuday/e6c934167d370e70b1a060dd21e658e3

Apart from `ls` command, we can also use these patterns in `if-else` statements to test a string for a custom pattern match. But instead of `[`, we need to use `[[` which is a **shell keyword** that Bash support and not a command like `[`. Adding `[[` to your bash script makes it incompatible with other shells like `sh`. But `[[` supports everything that `[` does.

`[[` is used to compare a string with Glob and extended Glob patterns. One more advantage

```
RESPONSE="Error: something went wrong."

if [[ $RESPONSE == +(Error)* ]]; then
    echo "Performing: failure handling"
else
    echo "Performing: success handling"
fi

⇒ Performing: failure handling
```

Since when you run a bash script file, it runs in the different session, hence we need to enable `extglob` option from within the script. Using `+(Error:)*` , we are testing if a string starts with at least one occurrence of `Error:` pattern.

Switch/Case type control flow in Bash

Bash support `switch/case` type control flow statements but not exactly with `switch` block and definitely not like `switch` syntax you might have known before. In Bash, we have a `case` block that contains different cases.

```
case STRING in
  globPattern1)
    statements
  ;;

globPattern2)
    statements
  ;;
esac
```

`case` , `in` and `esac` keywords are absolutely necessary. `)` marks the end of the pattern while `;;` marks the end of the statements for a **pattern-block**. If `STRING` contains a pattern

```
# enable `extglob` shell option
shopt -s extglob

# read what user's favourite planet
echo -n "Enter your favourite planet: "
read USER_PLANET

case "$USER_PLANET" in

  # match exactly `Earth`
  Earth)
    echo "Earth is where we live.";;

  # match exactly `Mars`
  Mars ) echo "Mars is where will go someday.";;

  # match anything
  * )
    echo "I am getting more information about this planet.";;
esac

⇒ Enter your favourite planet: ⇐ Mars
⇒ Mars is where will go someday.
```

From the above example, we can see that a `default case` can be constructed by using `*` pattern which matches everything. Since in Bash, only one pattern-block will be matched and executed, `*` will only be evaluated if none of the patterns above it is matched. We can also add space between pattern and `)` character as well as use `;;` to end the statements on the same line.

> 💡 *In a* `case` *block, if a pattern matches and respective statement(s) gets executed, exit status code the* `case` *block will be the exit status code of the last statement. If none of the patterns matches, the exit status code will be* `0`*. You can verify this by executing* `false` *command and echoing* `$?` *after* `case` *block.*

```
# fruits.sh
echo "\$0: $0"
echo "\$1: $1"
echo "\$2: $2"
echo "\$3: $3"

------------------------------

~$ bash fruits.sh Apples Mangoes
⇒ $0: fruits.sh
⇒ $1: Apples
⇒ $2: Mangoes
⇒ $3:
```

From the above program, we can see that, when we run `fruits.sh` using `bash` (*or using* `./fruits` *when the shebang line is added to the script*), the first positional parameter `$0` points to the path of the script relative to the **PWD** of the terminal. Later positional parameter `$N` points to the Nth argument passed to the command. If an argument is missing, `$N` will be empty.

Positional parameters are read-only and can not be modified. To modify the value of a positional parameter or calculate its length, we need to store it inside another custom variable. For example, to **calculate a length of argument** (*or any string*), we use `${#VARNAME}` syntax.

```
ACTION=$1

if [ "${#ACTION}" -eq 0 ]; then
    echo "Please provide an action. Length: ${#ACTION}"
fi

-------------------------------

~$ bash perform.sh
⇒ Please provide an action. Length: 0
```

If our script file receives arbitrary arguments or we just want all arguments as a single string, we can use `$*` syntax.

```
# fruits.sh
echo "Args: $*"

-------------------------------

~$ bash fruits.sh Apples Mangoes Bananas
⇒ Args: Apples Mangoes Bananas
```

There is also the `$@` syntax which is similar to `$*` but behaves differently when used in **double-quotes**. When used in quotes, `$*` expands to a single word while `$@` expands to separate words. This is very well explained in **this** answer as well as in the below example.

```
# fruits.sh

# loop around arguments string
for ARG in "$*"; do
    echo "ARGS ITEM: $ARG"
done

# loop around arguments vector
for ARG in "$@"; do
    echo "ARGV ITEM: $ARG"
done
```

```
⇒ ARGV ITEM: Apples
⇒ ARGV ITEM: Mangoes
⇒ ARGV ITEM: Bananas
```

There are other positional parameters like `$#` which stores the count of the number of arguments a script file has received. **Here** is a complete list of positional parameters.

Functions in Bash

We can also define functions inside a Bash script file (*function is also called as a procedure*). A function in Bash script behaves like a regular command because they are invoked like a command with arguments.

```
function name() {
    # statements
}
```

`function` keyword is optional as it is not supported in other shells like `sh`. We can do pretty much anything inside the function body. It is invoked by simply using the name of the function just like a regular command.

```
# greet.sh
function hi_me(){
    echo "Hello `whoami`"
}

hi_me # calling function `hi_me` with no arguments

-------------------------------

~$ bash greet.sh
⇒ Hello Uday.Hiwarale
```

but its body has a **separate scope for the positional parameters**. Hence positional parameters inside function body do not interfere with the outer scope.

```
# fruits.sh
function show_fruits(){
    echo "Local Args: \$1:$1 | \$2:$2"
}

show_fruits Apple Oranges

echo "Global Args: \$1:$1 | \$2:$2"

-------------------------------

~$ bash fruits.sh Mangos Grapes
⇒ Local Args: $1:Apple | $2:Oranges
⇒ Global Args: $1:Mangos | $2:Grapes
```

However, the **variables defined outside the scope are accessible inside the function**. Also, if **we declare a variable inside the function, it will be written to the global scop**e. To avoid that, we use `local` command. This way, we can prevent variables defined inside a function body to be written in the global scope.

```
function say_hello() {
    FIRST_NAME=$1         # global scope
    local LAST_NAME=$2    # local scope

    echo "Hello, $FIRST_NAME $LAST_NAME"
}

echo "Before: FIRST_NAME: $FIRST_NAME, LAST_NAME: $LAST_NAME"
say_hello Ross Geller
echo "After: FIRST_NAME: $FIRST_NAME, LAST_NAME: $LAST_NAME"

⇒ Before: FIRST_NAME: John, LAST_NAME: Doe
⇒ Hello, Ross Geller
⇒ After: FIRST_NAME: Ross, LAST_NAME: Doe
```

A function can also **return** a value. The return value is more like an exit status code we have seen with some commands before. Hence we need to access the returned value of a function using `$?` syntax.

```
get_random(){
    END=$1 # fist argument represents end digit

    # if $END is not empty
    # return number between `0` and `$END`
    if [ -n "$END" ]; then
        echo 'get_random() / context: if-block'
        return $(( $RANDOM % ( $END + 1 ) ))
    fi

    # if `END` is not provided, return $RANDOM
    echo 'get_random() / context: body'
    return $RANDOM
}

# call function
get_random
echo "get_random:" $?

# call function with 10 as argument
get_random 10
```

```
⇒ get_random() / context: if-block
⇒ get_random 10: 9
```

There is a huge problem with the above example. In any case, the value returned by the function would not go above `255`. This is because the return value of a function is meant to return the status code of the function execution, `0` meaning the function executed successfully and `1-255` meaning there was some problem with the function execution.

Hence, we should try to avoid function with return values and use return values to provide custom status codes. Since we can capture **`STDOUT`** of a command `$()` or ` `` ` syntax, we can use the same principles with function and just `echo` the return value so that invoker can capture it easily.

```
get_random(){
    END=$1

    if [ -n "$END" ]; then
        echo $(( $RANDOM % ( $END + 1 ) ))
    else
        echo $RANDOM
    fi
}

echo "get_random: $( get_random )"      # using $()
echo "get_random 10: `get_random 10`"   # using ``
```

> 💡 *A function in Bash can also be written in inline form using* `function name(){ //statement; }` *syntax (where* `;` *is mandatory). If a function does not return a value, the exit status of the function is the exit status of the last statement executed inside the function body.*

Arrays in Bash

```
ARRAY=(Apple "Orange" Mango)
```

And to access or set a value at a given index, we use `ARRAY[index]` syntax. Like in most of the programming languages, the `index` starts at **0**.

```
ARRAY=(Apple "Orange" Mango)

echo ${ARRAY[0]} # $ARRAY[0] is not valid
echo "At index 1: ${ARRAY[1]}" # string interpolation

ARRAY[2]=Banana # update value at index 2
ARRAY[3]=Papaya # add new value at index 3

echo ${ARRAY[2]} ${ARRAY[3]}

⇒ Apple
⇒ At index 1: Orange
⇒ Banana Papaya
```

Like we used `${#VAR_NAME}` to calculate the length of a variable `VAR_NAME`, we can use the same syntax to calculate length of an array item at a given index.

```
echo ${#ARRAY[0]}
⇒ 5
```

As we know `*` represents a wildcard, when we use it as an `index`, we can get all elements in the array. Likewise, using `*` as an `index` in the syntax `${#ARRAY[index]}`, we can calculate the length of an array.

```
⇒ Apple Orange Banana Papaya
⇒ Length of ARRAY: 4
```

Bash supports array iteration using `for:in` loop. And it also gives **`${!ARRAY[*]}`** syntax that returns a list of indexes which can also be iterated.

```
GREETINGS=("Good Morning" "Good Afternoon")

echo "GREETINGS: ${GREETINGS[*]}"

for GREET in ${GREETINGS[*]}; do
    echo "GREET: $GREET"
done

⇒ GREETINGS: Good Morning Good Afternoon
⇒ GREET: Good
⇒ GREET: Morning
⇒ GREET: Good
⇒ GREET: Afternoon
```

From the above example, we can't quite make sense of it. What is happening behind the scene when you write `${GREETINGS[*]}` is, Bash is returning the array items in a single string. And to fix that, we use string interpolation.

```
GREETINGS=("Good Morning" "Good Afternoon")

for GREET in "${GREETINGS[*]}"; do
    echo "GREET: $GREET"
done

⇒ GREET: Good Morning Good Afternoon
```

Hmm! This was expected. Because now our `for` loop is looping on a single string. This reminds us of the problem we had with `$*` positional parameter. `$*` expands to a single

Same goes with an array. `"${ARRAY[*]}"` syntax expands to a single word and that's why we have the above problem. We just need to use `@` as an index instead of `*` to make it expand into different words where a word is an actual item of the array.

```
GREETINGS=("Good Morning" "Good Afternoon")

for GREET in "${GREETINGS[@]}"; do
    echo "GREET: $GREET"
done

⇒ GREET: Good Morning
⇒ GREET: Good Afternoon
```

> 💡 `@` *is fully substitutional for* `*`*, for example, we can also use it to get indexes as a series of words with* `"${!ARRAY[@]}"` *syntax.*

Importing an external script

If you have previously worked with Bash terminal, then you might have come across `.bash_profile` which is a file that resides under your `~` (*home*) directory. This is a Bash script file which is automatically loaded in your terminal session when you open a terminal.

Since this lives inside a user directory, this will be only loaded when that particular user is successfully logged in (*in the terminal*). In contrast, `/etc/profile` file is imported system-wide irrespective of the user. So how does Bash imports these scripts and what's their use?

Like any other programming language, Bash also supports breaking your code into different files. If a bash script contains functions that can be reused or a code that must execute before the code you have written in a Bash script, we have the ability to import such script files.

files are called as **startup scripts** and they are loaded in a **startup order**.

When we source a Bash script, we **basically execute the content of that script file** in the current session. To source a Bash script, we use `source` (*shell built-in*) command, which you might have used like `source ~/.bash_profile` to load the changes made to `.bash_profile` file. This particular statement executes the content of `.bash_profile` in the current terminal session hence we are able to see the changes (*since `.bash_profile` is re-executed*).

Let's create a `functions.sh` Bash script which contains some function.

```
# functions.sh
function my_fake_ip() {
    echo '192.168.1.221'
}
```

Then will source this file inside `exec.sh`. This way, whenever `exec.sh` gets executed, it has the code of `functions.sh` and we can make use of it.

To source an external Bash script, we use `source` command and provide a **relative** or an **absolute** path of Bash script as an argument.

```
# exec.sh
source ./functions.sh

echo "My IP is: $(my_fake_ip)"

⇒ My IP is: 192.168.1.221
```

`source` command is an alias for `.` built-in command. Let's source `functions.sh` script inside our current terminal session and execute `my_fake_ip` function there.

```
⇒ 192.168.1.221
```

Using this feature, we can load other custom Bash scripts inside `.bash_profile` which will be eventually executed in our terminal session at the startup. For example, to get a real local IP address, we can write `myip` function inside `common.sh` and import it inside `.bash_profile`. This way, from our terminal, we can use `myip` function as a command.

```
~$ myip

⇒ 192.168.1.7
```

Other important topics

The Bash scripting language has much more features and things to worry about. Here are some things to watch out for and learn.

1. Aliases in Bash

We can create an alias of a command using `alias` command. For example, to list files in greater details, we use `ls -alh` command. We can shorten it by defining `ll` alias like below.

```
~$ alias ll="ls -alh"
~$ ll

⇒
drwxr-xr-x  11 Uday.Hiwarale  staff   352B  7 Sep 20:10 .
drwxr-xr-x   4 Uday.Hiwarale  staff   128B  4 Sep 06:38 ..
-rw-r--r--@  1 Uday.Hiwarale  staff   6.0K  6 Sep 12:45 photo.txt
```

https://gist.github.com/thatisuday/1365d0e0be1d086799697db4bd18a62c

3. Executing multiple commands at once

Bash gives us the ability to run multiple commands at once. You might have used `&&` to combine two command to run at once. Let's see other variants.

```
# || : Run commands until one is successful
false || echo "2: Got printed"
true || echo "2: This won't get printed"

# ; : Run all commands (just a statement separator)
false ; echo "2A: Got printed"
true ; echo "2B: Got printed"

⇒ 1: Got printed
⇒ 2: Got printed
⇒ 2A: Got printed
⇒ 2B: Got printed
```

We can pass `STDOUT` of command to `STDIN` of another command using `|` operator. For example, `STDOUT` of `ls -alh *` command can be passed to `less` command when we execute `ls -alh * | less` statement.

> 💡 *for more information about* `less` *command, execute* `man less` *command in your terminal.* `man` *is also a command that provides* ***manual*** *of a command distributed in UNIX-like operating systems.*

```
~$ ls -alh * | less

⇒
Applications:
total 16
drwx------@  5 Uday  staff   160B Aug 31 18:15 .
drwxr-xr-x+ 31 Uday  staff   992B Sep  9 17:16 ..
-rw-r--r--@  1 Uday  staff   6.0K Aug 31 18:16 .DS_Store
-rw-r--r--@  1 Uday  staff     0B Aug 28 16:23 .localized
drwx------@ 11 Uday  staff   352B Sep  4 23:03 Chrome Apps.localized

Creative Cloud Files:
total 47936
drwxrwxr-x@ 14 Uday  staff   448B Sep  8 00:46 .
drwxr-xr-x+ 31 Uday  staff   992B Sep  9 17:16 ..
-rw-r--r--@  1 Uday  staff     0B Sep  8 00:43 Icon
-rw-r--r--   1 Uday  staff   707K Jul 11 00:44 post-1.png
-rw-r--r--   1 Uday  staff   1.6M Jul 11 00:44 post-2.ai
```

The main purpose of passing the output of `ls -alh *` to `less` is to prevent `ls` command to dump too much information to the terminal that terminal can not handle. Using the above command, we can feed the output of `ls` to `less` and `less` will let us read that output one or few lines at a time.

Basically, `STDIN`, `STDERR` and `STDOUT` are streams, which means they are like pipes through which data flows. They are always open and any program can pass data through it. Anybody like terminal shell listening to these streams will get the data whenever somebody sends through it.

`|` is called the **pipe** operator, because it connects two streams and passes the data from one stream to another. For example, it can connect `STDOUT` of program to `STDIN` of another program. So `|` in `ls -alh * | less` basically connecting `STDOUT` of `ls` to `STDIN` of less.

If a command starts a continuously running program like infinite `while` loop and that program pipes data to another command, the connection between `STDOUT` and `STDIN` remain open. This can be dangerous in multiple ways. For example, if the program receiving a continuous stream of data from another program can not consume fast enough, the **pipe buffer** will eventually be full.

Let's create a simple infinitely running `while` loop that prints a random number, which means sends to `STDOUT` stream. But instead printing it to the terminal directly, we will redirect or **pipe** the output to `less` command which will read it one line at a time.

```
(while true; do sleep 1; echo $RANDOM; done) | less
```

The above program will never exit as while loop is continuously running. However, in this case, since the pipe operator connects the **output stream** of the left-side program to the **input stream** of `less` and it remains connected forever, whenever the left-side program

If we want to write the **output stream** of a command to a file, we can use `>` operator **AKA redirection operator**. This will overwrite a file content if the file already exists, else a new file will be created.

```
echo "Hello" > random.txt
cat random.txt

⇒ Hello

echo "World" > random.txt
cat random.txt

⇒ World
```

If you want to append the output of a command to a file, use `>>` instead.

```
echo "Hello " >> random.txt # adds a newline as well
cat random.txt

⇒ Hello

echo "World" >> random.txt # adds a newline as well
cat random.txt

⇒
Hello
World
```

If you are writing the output of a command that starts an infinitely running program, the pipe between file's `STDIN` and program's `STDOUT` will remain open. Any data sent in `STDIN` will be appended continuously in the file.

We can also use `<` operator which is same as `>` but it redirects the content of a file to a command. A typical use case syntax will look like `command < file.txt` which means send data of `file.txt` into the `command`.

```
⇒
Hello
World
```

> *So far, we have dealt with `STDOUT`, but if we need `STDERR` stream as well, there is `&>` and `&>>` syntax for it. It is explained very well in this **answer**.*

4. Environment variables

A shell variable is a variable defined inside a Bash script. When you source an external Bash script, variables defined inside the external Bash script will be available inside the current Bash script. These are **short-lived** variables.

Environmental variables are also Bash variable but defined with `export` command. Let's take a small example of defining such a variable.

```
# env-test.sh
export MY_IP='192.168.1.7'
echo "My IP is: $MY_IP"

-------------------------------

~$ bash env-test.sh
⇒ My IP is: 192.168.1.7
```

When we use `export` comand, Bash registers this variable and saves in the namespace of the current terminal session and sub-sessions (*sub-processes*) created by it. To see the list of all available environment variables, we use `printenv` command.

```
~$ printenv

⇒ TERM_PROGRAM=vscode
⇒ TERM=xterm-256color
...
⇒ HOME=/Users/Uday.Hiwarale
```

`bash` command runs a script in a separate session, the environment variable will be created there and killed at the end.

```
# env-test.sh
export MY_IP='192.168.1.7'
printenv

-------------------------------

~$ bash env-test.sh
⇒ MY_IP=192.168.1.7
⇒ TERM_PROGRAM=vscode
⇒ TERM=xterm-256color
...
⇒ HOME=/Users/Uday.Hiwarale
```

As you can see from the above results, `MY_IP` environmental variable exists inside the session started by `bash env-test.sh` command.

The main purpose of the environment is to set global values for sub-processes (*sub-sessions*) created by the main Bash script. Let's see an example.

```
# child.sh
echo "LOCAL_VAR inside child.sh: $LOCAL_VAR"
echo "MY_IP inside child.sh : $MY_IP"

# main.sh
LOCAL_VAR="MAIN"
export MY_IP='192.168.1.7'
bash ./child.sh  # starts a sub-session

-------------------------------

~$ bash main.sh
⇒ LOCAL_VAR inside child.sh:
⇒ MY_IP inside child.sh : 192.168.1.7
```

> *identical copy of the main shell process.* ***This*** *article explains the differenrce between sub-shell and sub-process.*

We can also pass an environmental variable to a process directly from the command which started it using below syntax. This way, our environment variables can be portable and we can avoid writing unnecessary boilerplate.

```
# main.sh
MY_IP='192.168.1.7' bash ./child.sh

------------------------------

~$ bash main.sh
⇒ MY_IP inside child.sh : 192.168.1.7
```

If you need to set an environmental variable for all the process started by the current terminal session, users can directly execute `export` command. But once you open a new terminal, it won't have that environmental variable. To make environmental variables accessible across all terminal sessions, export them from `.bash_profile` or any other startup script.

> 💡 *It is recommended to write* ***environment variables in all uppercase*** *and* ***shell variables in all lowercase****, but my personal choice is to use all uppercase for the both kinds. If you want to check if an environmental variable exists in the terminal session, you can just echo it using* `echo $SOME_ENV_VAR` *command.*

5. Code Block (statements block)

If we need to execute some code as a block, then we can put our code in `{}` curly braces. In contrast with `()` block which executes the code inside it in a sub-shell, **code-block** executes the code in the same shell, hence in the same shell process. Let's see a quick example.

```
    echo "code-block: $MAIN_VAR"
}
```

```
⇒ code-block: main
```

If we want to run some code as a block on a single line, we need to terminate the statements wi `;` character (*unlike a sub-shell*).

```
MAIN_VAR="main"

{ sleep 1; echo "code-block: $MAIN_VAR"; }
( sleep 1; echo "sub-shell: $MAIN_VAR" )

⇒ code-block: main
⇒ sub-shell: main
```

In the example, code-block and sub-shell both have access to `MAIN_VAR` because code-block run in the same environment of the main shell while sub-shell might run in the different process bu it has an identical copy of main process which also contains the variables from the main proces

The difference comes where we try to set or update a variable in the main process from a sub-shell. Here is a demonstration of that.

```
VAR_CODE_BLOCK="INIT"
VAR_SUB_SHELL="INIT"

{ VAR_CODE_BLOCK="MODIFIED"; echo "code-block: $VAR_CODE_BLOCK"; }
( VAR_SUB_SHELL="MODIFIED"; echo "sub-shell: $VAR_SUB_SHELL" )

echo "main/code-block: $VAR_CODE_BLOCK"
echo "main/sub-shell: $VAR_SUB_SHELL"

⇒ code-block: MODIFIED
⇒ sub-shell: MODIFIED
⇒ main/code-block: MODIFIED
⇒ main/sub-shell: INIT
```

A good use case of code block would be to **pipe** `(|)` or **redirect** `(>)` the output of some statements as a whole.

```
{ echo -n "Hello"; sleep 1; echo " World!"; } > hello.txt

-------------------------------

~$ bash main.sh && cat hello.txt
⇒ Hello World!
```

6. Special characters

In many examples, we have seen that it is not safe to pass a variable **as is** to a command or use inside an expression, because Bash expands it in series of arguments (*or series of words*).

There is no harm as long as our string contains plain words. The problem comes when a word means something different to Bash, like `*`. Such characters are called as special characters anc they should be either escaped using `\` like `\*` or enclosed inside quotes like `"*"`.

https://gist.github.com/thatisuday/10b6c065fd9c98d7fff74127fb356364

> 💡 ***Here*** *is a complete list of special characters in Bash you should watch out for.*

Open in app Get started

twitter/thatisuday
In case if you have a question.
github/thatisuday
This is where you will find some of the open source projects I am working on. Hop on and give me a hand to build awesome tools.
medium/@thatisuday
instagram/@thatisuday
Well, I don't do much here but I love movies. So you might see some stupid posts related to some stupid movies that you might end up liking.

(**thatisuday.com** / **GitHub** / **Twitter**/ **StackOverflow** / **Instagram**)

Please clap if you found this article useful.