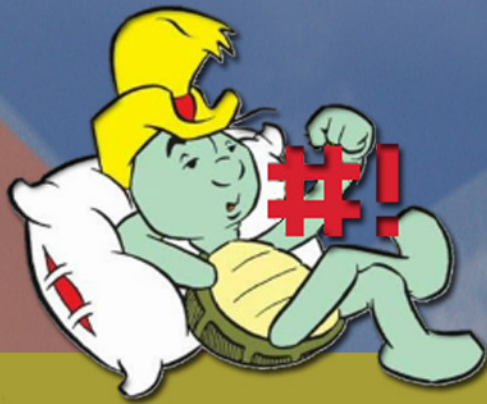


# Shellman



## **#!BASH** **SCRIPTING**



*Remisa Yousefvand*

# Shellman Bash Scripting

Remisa Yousefvand

This book is for sale at <http://leanpub.com/shellman>

This version was published on 2019-01-07



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2019 Remisa Yousefvand

*To my mom and daddy. Thank you for your unconditional support and for being the source of passion and inspiration in my life.*

# Contents

<b>Preface</b> . . . . .	<b>i</b>
<b>Prerequisites</b> . . . . .	<b>ii</b>
<b>Shellman Structure</b> . . . . .	<b>iii</b>
<b>Shell Scripting Basics</b> . . . . .	<b>1</b>
Comments . . . . .	1
shebang . . . . .	1
Run a Bash Script . . . . .	1
Run a Command from Shell Script . . . . .	1
Multiline Command . . . . .	2
Variables . . . . .	2
Variable Types . . . . .	4
Commands . . . . .	4
Argument parsing . . . . .	6
Organizing your Bash Script . . . . .	8
Double Quote vs Single Quote vs Backtick . . . . .	9
<b>Namespaces</b> . . . . .	<b>11</b>
loop . . . . .	12
logic . . . . .	16
string . . . . .	25
math . . . . .	29
date . . . . .	33
time . . . . .	35
array . . . . .	36
directory . . . . .	40
function . . . . .	42
command . . . . .	44
archive . . . . .	45
crypto . . . . .	46
http . . . . .	47
ftp . . . . .	50
file . . . . .	51

## CONTENTS

color . . . . .	53
format . . . . .	54
system . . . . .	55
git . . . . .	58
miscellaneous . . . . .	64
lib . . . . .	68
<b>Advanced . . . . .</b>	<b>74</b>
piping . . . . .	74
shift command . . . . .	74
<b>Solutions . . . . .</b>	<b>75</b>
Argument Parsing . . . . .	75
Nested Directories . . . . .	76
Colorful Text . . . . .	76
Factorial . . . . .	76

# Preface

I was thinking about how to simplify usage of Shellman by sharing some tips that could be helpful for both using Shellman and improving shell scripting skills so I liked to share them in Readme of Shellman but as you know there are some limitations thus I thought maybe a lean and simple guide is not a bad idea for the purpose.

The hard part of *shell scripting* is not *shell scripting* itself, it is knowing the correct *command* and *switches*, so if you can do it in *terminal*, you can do it easily via shell script too. *Shell scripting* is useful for common tasks automation.

This book is a guide for beginners who want to start shell scripting with **Shellman** effectively. If you are of pragmatic type people then go ahead and read **Basics** section and desired **namespaces**.

Also the business model of **Shellman** is published on [medium](https://medium.com/@remisa.yousefvand/shellman-reborn-f2cc948ce3fc)<sup>1</sup>.

This book is a *work in progress*, please consider downloading new versions once a while.

Remisa Yousefvand

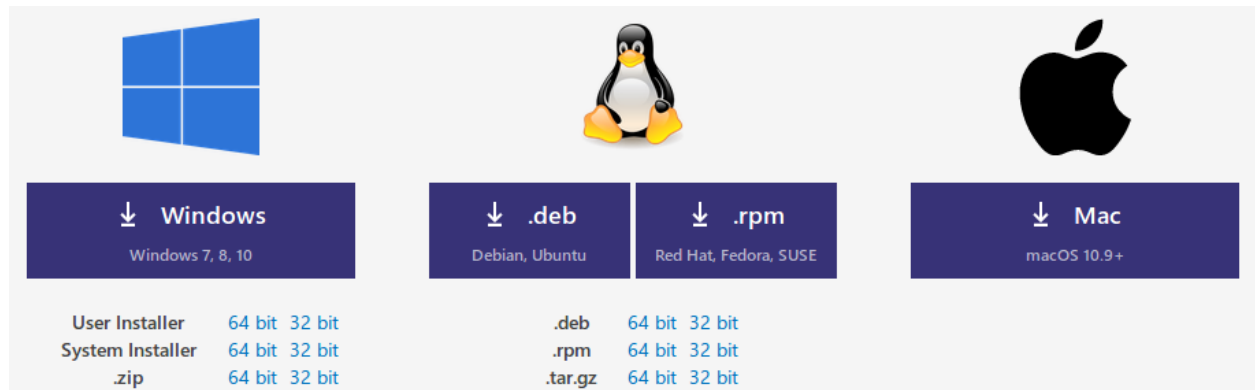
January 2019

---

<sup>1</sup><https://medium.com/@remisa.yousefvand/shellman-reborn-f2cc948ce3fc>

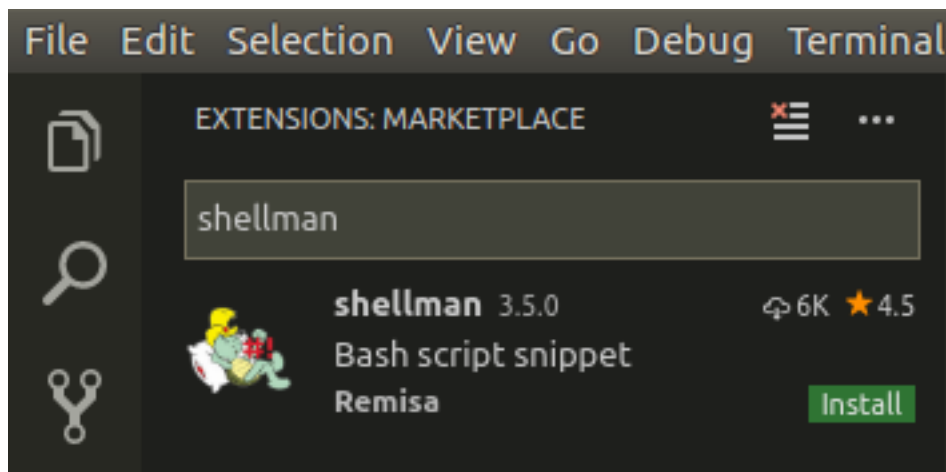
# Prerequisites

- [vscode](#)<sup>2</sup> IDE



vscode download

- [Shellman](#)<sup>3</sup> snippet



shellman install

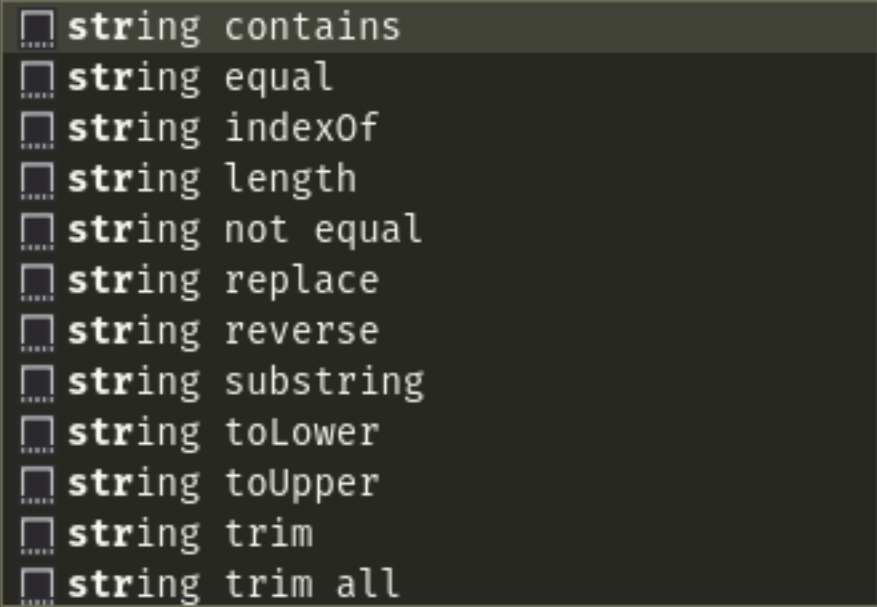
<sup>2</sup><https://code.visualstudio.com>

<sup>3</sup><https://marketplace.visualstudio.com/items?itemName=Remisa.shellman>

# Shellman Structure

*Shellman* divides its content into semantical categories named **namespace**. The concept is already familiar to programmers, but in simple words it means *keeping related materials together under a generic name*. So if you need to do something with `String` like changing it to upper case then it makes sense to look at `string` namespace.

```
1  #!/usr/bin/env bash
2
3  | str
```



The screenshot shows a terminal window with a dark background. The first two lines are a shebang and an empty line. The third line shows a prompt followed by the text 'str'. A completion menu is displayed, listing various string-related functions with a small icon to the left of each item. The items are: 'string contains', 'string equal', 'string indexOf', 'string length', 'string not equal', 'string replace', 'string reverse', 'string substring', 'string toLower', 'string toUpper', 'string trim', and 'string trim all'.

- string contains
- string equal
- string indexOf
- string length
- string not equal
- string replace
- string reverse
- string substring
- string toLower
- string toUpper
- string trim
- string trim all

String Namespace

*Shellman* is structured into [namespaces](#), so it is useful to know supported namespaces and their members. There is no order in learning [namespaces](#) and you can learn them on need, but before that, you need to know a few things about *shell scripting*. I will try my best to keep [Basics](#) section short and simple so you can move fast to desired [namespaces](#).



# Shell Scripting Basics

## Comments

In shell scripts anything after # in a line is considered a comment. The exception is [shebang](#) which you see as the first line of scripts.

```
1 # This is a comment
```

## shebang

This is the first line of any bash script. You may see different versions of it:

- `#!/usr/bin/sh`
- `#!/usr/bin/bash`
- `#!/usr/bin/env bash`
- ...

This line tells the *operating system* which script engine should be used to run the script. Usually you don't need to change the default value **Shellman** provides:

```
1 #!/usr/bin/env bash
```

## Run a Bash Script

Bash script files by convention has `.sh` *file extension*<sup>4</sup>. To run a bash script (`test.sh` for example) from terminal you have two options:

- Run it with bash command (pass file path to bash):
  1. `bash test.sh`
- Give it execute permission and run it directly (prefix file name with a `./` without space):
  1. `chmod +x test.sh`
  2. `./test.sh`

## Run a Command from Shell Script

To run a command from your script just write it as you do in terminal:

---

<sup>4</sup>In *Linux* unlike *Windows*, file extensions has no special meaning to *operating system* but still you can use them to remember which file type you are dealing with. **vscode** uses file extensions to recognize file types (`.sh` for *Shellscript*)

```
1  #!/usr/bin/env bash
2
3  rm some_file
```

If the command need **root**<sup>5</sup> privileges (in *Windows* it is known as *Admin*), prefix the command with **sudo**:

```
1  #!/usr/bin/env bash
2
3  sudo rm some_file
```

If you need the result of the executed command refer to [command substitution](#).

## Multiline Command

A single command can be written in multiple lines if each line ends in a backslash.

```
1  #!/usr/bin/env bash
2
3  curl --request GET -sL \
4      --user-agent 'Shellman' \
5      --cookie 'key=value' \
6      --url 'http://example.com'
```

You can write multiple commands in a single line and separate them by semicolon (;).

```
1  #!/usr/bin/env bash
2
3  var1=2; var2=3; var3="hello"
```

## Variables

There is a simple difference between when you define a variable and when use its value. In latter case you need to prefix a \$ to the variable name.

Define a variable named `firstName` and set its value to `Remisa`:

---

<sup>5</sup>In *Linux/Unix* systems, **root** is the most privileged user (same as *Administrator* in *Windows*).

```
1 firstName=Remisa
```



## Variable Assignment Rule

Spaces are not allowed over equal sign = in variable assignment.

Now if we want to read our variable value and print in on screen with echo command we can write:

```
1 firstName=Remisa
2 echo $firstName
```



## Variable Access Rule

To access a variable value prefix it with \$

Here we face a serious problem which without well understanding it, *shell scripting* becomes *hell scripting*.

As you may guessed in assignment rule, *space* has a special meaning in *shell scripting* and we should take care of where a *space* may appear. For example our variable value may contains *space*:

```
1 fullName=Remisa Yousefvand
```

Now when we want to use `fullName` value we put a \$ before it and use `$fullName` instead. But it contains *space* and we need to take care of that. To do so, simply surround wherever whitespace may appear in "":

```
1 fullName="Remisa Yousefvand"
2 echo "$fullName"
```



## Handling whitespace in variables

Always surround variables in "" when accessing their values if they may contain white space(s).

To concat multiple variables put them in "" in desired order:

```
1 a="Hello"
2 b="world"
3 c="!"
4 echo "$a $b$c"
5 # Hello world!
```

The whitespace between `$a` and `$b` is the whitespace between `Hello` and `world` in the output.

If we want to assign a variable if and only if it has no value currently, then we can use `assign if empty` snippet:

```
1 #!/usr/bin/env bash
2
3 : "${variable:=default}"
```

In above example `variable` is set only if it is *empty*.

## Variable Types

The only **type** you have in shell is **String**. Even when working with numbers they are strings you pass to commands which take care of converting those strings to numbers, do calculations, and return **String** back to you.

## Commands

### Command substitution

It is common practice to store the output of commands inside variables for further processing in script. The process is known as *command substitution* and can be done in two syntaxes:

1. `output=`command``
2. `output=$(command)`

In some references method two is recommended specially for nested command substitutions but for the sake of brevity and consistency, we will use method one (backtick) in this book.

To store results of `ls` command in a variable named `output`:

```

1 output=`ls` # store ls results in a variable named output
2
3 echo "$output" # print output value (ls result)

```

There is a more advance technique for using a command output as another command input, namely **piping** (`|`), you can read about it in [advanced](#) section.

## Command success/failure check

It happens when you are interested to know if a previous command succeeded or failed. In Linux every program returns a number to *operating system* on exit<sup>6</sup>. By convention if the return value is *zero*, in means no error happened and other values indicates command **failure**.



### Command success/failure

Programs return 0 in case of **success** and non zero if **failure** happens.

To check that, you can check *last command return value* by reading `$?` value. There is a snippet at func [namespace](#) for retrieving last command return value as func `ret val`:

```

1 echo "$?"

```

**Shellman** supports checking **failure** of last command via cmd [namespace](#) as cmd `failure check` snippet:

```

1 # following command will fail due to lack of permission
2 touch /not_enough_permission_to_create_file

```

`touch` command creates an empty file. Here we are trying to create the empty file `not_enough_permission_to_create_file` at the root of your file system. Without **sudo** this command will fail due to lack of enough permissions.

```

1 touch /not_enough_permission_to_create_file
2
3 # check last command (touch) success/failure
4 if [[ $? != 0 ]]; then
5     echo "command failed"
6 fi

```

To check **success**, use cmd `success check` snippet from cmd [namespace](#):

---

<sup>6</sup>This number is between 0 and 255 (one byte). If you have ever programmed in C/C++, you may noticed a return 0 as a default behavior, that is the code your program is returning to OS, here 0 as success.

```
1 echo "Hello World!"
2
3 # check last command (echo) success/failure
4 if [[ $? == 0 ]]; then
5     echo command succeed
6 fi
```

## Argument parsing

By convention most Linux commands/programs supports a long and short version for the same flag/switch. Short version is usually the first letter of the long version. Some examples:

short	long
-v	-verbose
-s	-silent
-f	-force
-o	-output

You may want to support different *switches/flags* by your script and act differently based on them. Suppose your script name is `backup.sh`. With supporting flags someone can run it as:

```
1 ./backup.sh -v
```

So your script works different with `-v`. For example you print verbose information. We need to know if user has run our script with or without `-v` flag. **Shellman** makes it easy for you, keep reading.

If your script supports *switches*, it means user is passing some information to your script via that switch. For example where to save the backup in our example:

```
1 ./backup.sh -o ~/my_backups
```

In above code we are telling the script to save the output in `~/my_backups`<sup>7</sup> directory.



### Flag vs Switch

**Flag** is used for boolean values and its presence means **True** while **Switch** accepts an argument.

**Shellman** has a `parse args` snippet. It looks like this:

---

<sup>7</sup> `~` is a shorthand for current user, *home directory*, which usually is `/home/username`.

```

1  POSITIONAL=( )
2  while [[ $# > 0 ]]; do
3      case "$1" in
4          -f|--flag)
5              echo flag: $1
6              shift # shift once since flags have no values
7              ;;
8          -s|--switch)
9              echo switch $1 with value: $2
10             shift 2 # shift twice to bypass switch and its value
11             ;;
12             *) # unknown flag/switch
13             POSITIONAL+=("$1")
14             shift
15             ;;
16         esac # end of case. "case" word in reverse!
17     done
18
19     set -- "${POSITIONAL[@]}" # restore positional params

```

This snippet will take care of **Flags** and **Switches** of your script. For implementing your own flag(s) replace `-f|--flag` with desired flag, i.e. `-v|--verbose` and on the next lines (before `shift`) do whatever you need. It is recommended to define a variable and set it here to keep track of the flag:

```

1  -v|--verbose)
2      verbose=true

```

Repeat above procedure for more flags.

To implement a **switch** like `-o/--output`:

```

1  -o|--output)
2      output_path=$2

```

In above example we are saving the switch value in `output_path` for using later.

Repeat above procedure for more switches.



## Argument Parsing Exercise

Write a shell script to greet. Script receives the name via `--name` or `-n` switch to print good night name and if `-m` flag is set, it should print good morning name. name is what value passed to script via `--name` flag. If `--name` or `-n` is not passed default value would be everyone. Example outputs:

```
1 ./greet.sh
2 # good night everyone
3
4 ./greet.sh -m
5 # good morning everyone
6
7 ./greet.sh --name Remisa
8 # good night Remisa
9
10 ./greet.sh -n Remisa
11 # good night Remisa
12
13 ./greet.sh -m --name Remisa
14 # good morning Remisa
15
16 ./greet.sh -m -n Remisa
17 # good morning Remisa
```

For the answer refer to [Solutions](#) section, [argument parsing](#).

As you have noticed, first argument can be accessed via \$1, second argument via \$2...

Same is true inside the body of a function to access passed arguments to the function.

## Organizing your Bash Script

Using **Shellman** snippets you can well organize your bash script, so it is easy to read by other users. Recommended structure of `script.sh` from top to bottom is:

1. shebang (bash snippet)
2. summary
3. functions region
4. command parsing

In *summary* you provide some information about `script`.





```
1  var1="Hello World!"
2  echo "$var1" # Hello World!
3
4  var2='$var1'
5  echo "$var2" # $var1
6
7  var3=' "&$*'
8  echo "$var3" # "&$*
```

Use *backtick* for command substitution

```
1  directoryList=`ls | xargs echo`
2  echo "$directoryList"
```

# Namespaces

Namespaces are semantic categories to hold related items together. *Folders* play the same role in keeping related *files* together on a *file system*.

# loop

Contains `while`, `until`, `for`.

## while

`while` condition.

For arithmetic comparison use `(( ))`.

```
1  #!/usr/bin/env bash
2
3  a=3
4  while (( a > 0 )); do
5      echo "$a"
6      ((a--))
7  done
8  # 3
9  # 2
10 # 1
```

For string comparison use `[ ]`.

```
1  #!/usr/bin/env bash
2
3  str="s"
4  while [ "$str" != "end" ]; do
5      echo "start"
6      str="end"
7  done
8  # start
```

## until

`until` condition (opposite of `while`).

For arithmetic comparison use `(( ))`.

```
1  #!/usr/bin/env bash
2
3  a=3
4  until (( a <= 0 )); do
5      echo "$a"
6      ((a--))
7  done
8  # 3
9  # 2
10 # 1
```

For string comparison use [ ].

```
1  #!/usr/bin/env bash
2
3  str="s"
4  until [ "$str" == "end" ]; do
5      echo "start"
6      str="end"
7  done
8  # start
```

## for i

for loop.

```
1  #!/usr/bin/env bash
2
3  for((i=0;i<5;i++)); do
4      echo "$i"
5  done
6  # 0
7  # 1
8  # 2
9  # 3
10 # 4
```

## for i j

Nested for loop.

```
1  #!/usr/bin/env bash
2
3  for((i=0;i<3;i++)); do
4      for((j=0;j<2;j++)); do
5          echo "$i, $j"
6      done
7  done
8  # 0, 0
9  # 0, 1
10 # 1, 0
11 # 1, 1
12 # 2, 0
13 # 2, 1
```

## for in

Iterate over ranges. Range can be numerical or alphabetical and can be defined as {start..end}.

Numerical range:

```
1  #!/usr/bin/env bash
2
3  for item in {1..5}; do
4      echo "$item"
5  done
6  # 1
7  # 2
8  # 3
9  # 4
10 # 5
```

alphabetical range:

```
1  #!/usr/bin/env bash
2
3  for item in {A..D}; do
4      echo "$item"
5  done
6  # A
7  # B
8  # C
9  # D
```

## for in column

Sometimes output is arranged in multiple columns while we are interested in one or few of them. For example output of `docker images` command:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sonatype/nexus3	3.13.0	777b20c20405	3 months ago	505MB
sonatype/nexus3	latest	777b20c20405	3 months ago	505MB
busybox	glibc	c041448940c8	4 months ago	4.42MB
busybox	latest	c041448940c8	4 months ago	4.42MB

What if we are just interested in column one?

```
1  #!/usr/bin/env bash
2
3  for col in `docker images | awk '{ print $1}'`; do
4      echo "$col"
5  done
```

Output of above script is:

```
1  REPOSITORY
2  sonatype/nexus3
3  sonatype/nexus3
4  busybox
5  busybox
```

If you need column two you can *pipe* (`|`) output of `docker images` to `awk '{ print $2}':`

```
1  #!/usr/bin/env bash
2
3  for col in `docker images | awk '{ print $2}'; do
4      echo "$col"
5  done
```

Output would be:

```
1  TAG
2  3.13.0
3  latest
4  glibc
5  latest
```

## logic

You can find logical related commands here under `if` namespace.

### if

`if`, else condition.

For arithmetic comparison use `(( ))`.

```
1  #!/usr/bin/env bash
2
3  var1=32
4  var2=33
5
6  if (( $var1 == $var2 )); then
7      echo "equal"
8  elif (( $var1 >= $var2 )); then
9      echo "bigger"
10 else
11     echo "smaller"
12 fi
13 # smaller
```

For string comparison use `[ ]`.

`elif` part can be repeated as much as necessary.



```
1  #!/usr/bin/env bash
2
3  str1="bye"
4  str2="hello"
5  str3="bye"
6
7  if [ "$str1" = "$str2" ]; then
8      echo "1 = 2"
9  elif [ "$str1" = "$str3" ]; then
10     echo "1 = 3"
11 elif [ "$str2" = "$str3" ]; then
12     echo "2 = 3"
13 else
14     echo "no equal pair"
15 fi
16 # 1 = 3
```

Simpler forms of if:

```
1  #!/usr/bin/env bash
2
3  str1="bye"
4  str2="hello"
5
6  if [ "$str1" = "$str2" ]; then
7      echo "equal"
8  fi
```

or if/else:

```
1  #!/usr/bin/env bash
2
3  str1="bye"
4  str2="hello"
5
6  if [ "$str1" = "$str2" ]; then
7      echo "equal"
8  else
9      echo "NOT equal"
10 fi
11 # NOT equal
```

## iff

If condition is true then run command (short circuit).

For arithmetic comparison use `(( ))`.

```
1  #!/usr/bin/env bash
2
3  var=5
4  (( var > 3 )) && echo "greater than 3"
5  # greater than 3
```

For string comparison use `[ ]`.

```
1  #!/usr/bin/env bash
2
3  var="hi"
4  [ "$var" = "hi" ] && echo "hi"
5  # hi
```

## iff not

If condition is false then run command (short circuit).

For arithmetic comparison use `(( ))`.

```
1  #!/usr/bin/env bash
2
3  var=5
4  (( var > 8 )) || echo "less than 8"
5  # less than 8
```

For string comparison use `[ ]`.

```
1  #!/usr/bin/env bash
2
3  var="hi"
4  [ "$var" = "bye" ] || echo "hi"
5  # hi
```

## if directory exists

Check if given *path* is a *directory*.

```
1  #!/usr/bin/env bash
2
3  if [ -d ~/backup ]; then
4      echo "backup exists"
5  fi
```

## if cmd exists

Read `cmd`

## if exists

If *path* is a *file* or *directory*

```
1  #!/usr/bin/env bash
2
3  path=~/.bashrc
4  if [ -e "$path" ]; then
5      echo exists
6  fi
```

## if file =

Check if two files are equal.

```
1  #!/usr/bin/env bash
2
3  file1=~/.some_file
4  file2=~/.another_file
5
6  if [ "$file1" -ef "$file2" ]; then
7      echo files are equal
8  fi
```

## if file executable

Check if file is executable.

```
1  #!/usr/bin/env bash
2
3  if [ -x /bin/ls ]; then
4      echo file is executable
5  fi
```

## if file link

If given *path* is a *symbolic link*.

```
1  #!/usr/bin/env bash
2
3  if [ -h /vmlinuz ]; then
4      echo symbolic link
5  fi
```

## if file newer

Check if first file is newer than the second.

```
1  #!/usr/bin/env bash
2
3  file1=~/.bashrc
4  file2=~/.profile
5
6  if [ "$file1" -nt "$file2" ]; then
7      echo file1 is newer than file2
8  fi
```

## if file not empty

Check if file is not empty.

```
1  #!/usr/bin/env bash
2
3  if [ -s ~/.profile ]; then
4      echo file not empty
5  fi
```

## if file older

Check if first file is older than the second.

```
1  #!/usr/bin/env bash
2
3  file1=~/.bashrc
4  file2=~/.profile
5
6  if [ "$file1" -ot "$file2" ]; then
7      echo file1 is older than file2
8  fi
```

## if file readable

Check if file is readable.

```
1  #!/usr/bin/env bash
2
3  if [ -r ~/.profile ]; then
4      echo file is readable
5  fi
```

## if file writable

Check if file is writable.

```
1  #!/usr/bin/env bash
2
3  if [ -w ~/.profile ]; then
4      echo file is writable
5  fi
```

## if int =

Check if two integers are equal.

```
1  #!/usr/bin/env bash
2
3  int1=67
4  int2=67
5
6  if (( int1 == int2 )); then
7      echo equal
8  fi
```

## if int !=

Check if two integers are not equal.

```
1  #!/usr/bin/env bash
2
3  int1=12
4  int2=13
5
6  if (( int1 != int2 )); then
7      echo not equal
8  fi
```

## if int <

Check if the first integer is smaller than the second.

```
1  #!/usr/bin/env bash
2
3  int1=12
4  int2=13
5
6  if (( int1 < int2 )); then
7      echo lesser
8  fi
```

## if int <=

Check if the first integer is smaller or equal to the second one.

```
1  #!/usr/bin/env bash
2
3  int1=12
4  int2=13
5
6  if (( int1 <= int2 )); then
7      echo less or equal
8  fi
```

## if int >

Check if the first integer is greater than the second.

```
1  #!/usr/bin/env bash
2
3  int1=15
4  int2=13
5
6  if (( int1 > int2 )); then
7      echo greater
8  fi
```

## if int >=

Check if the first integer is greater or equal to the second one.

```
1  #!/usr/bin/env bash
2
3  int1=12
4  int2=13
5
6  if (( int1 >= int2 )); then
7      echo greater or equal
8  fi
```

## if string empty

Check if string is empty.

```
1  #!/usr/bin/env bash
2
3  str=""
4  if [ -z "$str" ]; then
5      echo "Empty string"
6  fi
7  # Empty string
```

## if string not empty

Check if string is not empty.

```
1  #!/usr/bin/env bash
2
3  str="a"
4  if [ -n "$str" ]; then
5      echo "String is not empty"
6  fi
7  # String is not empty
```

## string equal | if string =

Check if strings are equal.

```
1  #!/usr/bin/env bash
2
3  str1="hello"
4  str2="hello"
5  if [ "$str1" = "$str2" ]; then
6      echo "equal"
7  fi
8  # equal
```

## string not equal | if string !=

Check if strings are not equal.



```
1  #!/usr/bin/env bash
2
3  str1="hi"
4  str2="hello"
5  if [ "$str1" != "$str2" ]; then
6      echo "not equal"
7  fi
8  # not equal
```

## if string contains

Check if string contains given substring.

```
1  #!/usr/bin/env bash
2
3  str="hello world!"
4
5  if [[ "$str" = *world* ]]; then
6      echo contains world
7  fi
8  # contains world
```

## string

Contains String related operations.

### contains

Checks if a String contains another String (substring).

```
1  #!/usr/bin/env bash
2
3  var="hello world!"
4
5  if [[ "$var" = *world* ]]; then
6      echo "substring found"
7  else
8      echo "substring NOT found"
9  fi
```

## equal

Checks if two Strings are the same.

```
1  #!/usr/bin/env bash
2
3  string1='This is a string!'
4  string2='This is a string!'
5
6  if [ "$string1" = "$string2" ]; then
7      echo 'Strings are equal'
8  fi
```

## indexOf

Returns index of substring inside a string.

```
1  #!/usr/bin/env bash
2
3  myString="Hello World!"
4  temp=${myString%%"or"*} && indexOf=`echo ${myString%%"or"*} | echo ${#temp}`
5  echo $indexOf # 7
```

## length

Returns *length* of given string.

```
1  #!/usr/bin/env bash
2
3  var="abcdefg"
4  length=${#var}
5  echo "$length"
```

## not equal

Checks if two strings are not equal.

```
1  #!/usr/bin/env bash
2
3  str1="shellman"
4  str2="shellmen"
5  if [ "$str1" != "$str2" ]; then
6      echo "Strings are NOT equal"
7  fi
```

## replace

Replace a substring with given string in another string.

```
1  #!/usr/bin/env bash
2
3  str1="Hello World!"
4  replaced=`echo -e "${str1}" | sed -e 's/World/Everyone/g'`
5  echo "$replaced" # Hello Everyone!
```

## reverse

Reverse given string.

```
1  #!/usr/bin/env bash
2
3  str1="abcd"
4  reversed=`echo -e "${str1}" | rev`
5  echo "$reversed" # dcba
```

## substring

Returns a substring from given string starting at *index* and with the length of *length*.

```
1  #!/usr/bin/env bash
2
3  str1="abcdefg"
4  substring=`echo -e "${str1:2:3}"`
5  echo "$substring" # cde
```

In above example we want a substring starting at *index* 2 to the *length* of 3. In abcdefg index 2 is c (index starts at zero) and length of 3 will end up cde.

## toLower

Returns lowercase of given string.

```
1  #!/usr/bin/env bash
2
3  str1="AbCdE"
4  toLower=`echo -e "${str1}" | tr '[:upper:]' '[:lower:]'`
5  echo "$toLower" # abcde
```

## toUpper

Returns uppercase of given string.

```
1  #!/usr/bin/env bash
2
3  str1="AbCdE"
4  toLower=`echo -e "${str1}" | tr '[:upper:]' '[:lower:]'`
5  echo "$toLower" # abcde
```

## trim

Removes leading and trailing whitespace(s).

```
1  #!/usr/bin/env bash
2
3  str1="  result  "
4  result=`echo -e "${str1}" | sed -e 's/^[[:space:]]*//' | sed -e 's/[[:space:]]*$//'`
5  echo "Variable $result contains no leading and trailing space as you see"
6  # Variable result contains no leading and trailing space as you see
```

## trim all

Removes all whitespace(s) from given string (leading, inside, trailing).

```
1  #!/usr/bin/env bash
2
3  str1="  ab c de  "
4  result=`echo -e "${str1}" | tr -d '[:space:]'`
5  echo "All whitespaces are removed from $result as you see"
6  # All whitespaces are removed from abcde as you see
```

## trim left

Removes all whitespace(s) from left of given string (leading).

```
1  #!/usr/bin/env bash
2
3  str1="  whitespace on left"
4  result=`echo -e "${str1}" | sed -e 's/^[[:space:]]*//'`
5  echo "There is no $result as you see"
6  # There is no whitespace on left as you see
```

## trim right

Removes all whitespace(s) from right of given string (trailing).

```
1  #!/usr/bin/env bash
2
3  str1="whitespace on right  "
4  result=`echo -e "${str1}" | sed -e 's/[[:space:]]*$//'`
5  echo "There is no $result as you see"
6  # There is no whitespace on right as you see
```

## math

Contains Math related operations. Math functions are available under `fn math ... namespace`.

## reminder %

Given two numbers, returns reminder of dividing the first number to the second number.

```
1  #!/usr/bin/env bash
2
3  var1=17
4  var2=5
5  reminder=$((var1 % var2))
6  echo "$reminder" # 2
```

## multiply \*

Given two numbers, returns product of them.

```
1  #!/usr/bin/env bash
2
3  var1=3
4  var2=4
5  result=$((var1 * var2))
6  echo "$result" # 12
```



## Factorial

Write a function which gets a number N and prints N!.

For the answer refer to [Solutions](#) section, [factorial](#).

## add +

Given two numbers, returns sum of them.

```
1  #!/usr/bin/env bash
2
3  var1=2
4  var2=3
5  result=$((var1 + var2))
6  echo "$result" # 5
```

## increase ++

Given a number, adds one to it.

```
1  #!/usr/bin/env bash
2
3  var=7
4  echo $((++var)) # 8
```

## subtract -

Given two numbers, returns subtract of the second from the first.

```
1  #!/usr/bin/env bash
2
3  var1=7
4  var2=5
5  result=$((var1 - var2))
6  echo "$result" # 2
```

## decrease -

Given a number, subtracts one from it.

```
1  #!/usr/bin/env bash
2
3  var=8
4  echo $((--var)) # 7
```

## divide /

Given two numbers, returns first divided by the second.

```
1  #!/usr/bin/env bash
2
3  var1=12
4  var2=4
5  result=$((var1 / var2))
6  echo "$result" # 3
```

## scale 0.00

Math operations with x decimal point precision.

Multiply example:

```
1  #!/usr/bin/env bash
2
3  var1="2.13"
4  var2=""2
5  result=`echo "scale=2;($var1 * $var2)" | bc`
6  echo "$result" # 4.26
```

Division example:

```
1  #!/usr/bin/env bash
2
3  var1=7
4  var2=2
5  result=`echo "scale=2;($var1 / $var2)" | bc`
6  echo "$result" # 3.50
```

## exponentiation ^

Exponentiate *base* to the *power*.

```
1  #!/usr/bin/env bash
2
3  echo $((2 ** 4)) # 16
4  echo $((3 ** 3)) # 27
```

## square root

Returns square root of given number up to given *precision*.

Calculate square root of 2 up to 7 decimal points.

```
1  #!/usr/bin/env bash
2
3  var=2
4  result=`echo "scale=7;sqrt($var)" | bc`
5  echo "$result" # 1.4142135
```

## random

Generate random number between *min* and *max*



```
1  #!/usr/bin/env bash
2
3  echo $((5000 + RANDOM % $((65535-5000)))) # 27502
```

## constants

Some useful math constants.

## date

Contains Date related operations.

### now short

Short version of current system *date*.

```
1  #!/usr/bin/env bash
2
3  dateShort=`date -I`
4  echo "$dateShort" # 2019-01-06
```

### now UTC

Returns current system time in *Coordinated Universal Time* format.

```
1  #!/usr/bin/env bash
2
3  dateUTC=`date -u`
4  echo "$dateUTC" # Sunday, January 06, 2019
```

### now year

Current year.

```
1  #!/usr/bin/env bash
2
3  year=`date +%Y`
4  echo "$year" # 2019
```

## now monthNumber

Current month number.

```
1  #!/usr/bin/env bash
2
3  monthNumber=`date +%m`
4  echo "$monthNumber" # 01
```

## now monthName

Current month name.

```
1  #!/usr/bin/env bash
2
3  monthName=`date +%B` # %B for full month name, %b for abbreviated month name
4  echo "$monthName" # January
```

## now dayOfMonth

Current day of month.

```
1  #!/usr/bin/env bash
2
3  dayOfMonth=`date +%d`
4  echo "$dayOfMonth" # 06
```

## now dayOfWeek

Current weekday name.

```
1  #!/usr/bin/env bash
2
3  dayOfWeek=`date +%A` # %A for full weekday name, %a for abbreviated weekday name
4  echo "$dayOfWeek" # Sunday
```

## now dayOfYear

Current day of year (1-366).

```
1  #!/usr/bin/env bash
2
3  dayOfYear=`date +%j`
4  echo "$dayOfYear" # 006
```

## time

Contains Time related operations.

### now local

Current local time.

```
1  #!/usr/bin/env bash
2
3  timeNowLocal=`date +%R` # %R for 24 hrs
4  echo "$timeNowLocal" # 13:23
5
6  timeNowLocal=`date +%r` # %r for 12 hrs
7  echo "$timeNowLocal" # 01:23:45
```

### now UTC

Current UTC time.

```
1  #!/usr/bin/env bash
2
3  timeNowUTC=`date -u +%R`
4  echo "$timeNowUTC" # 12:56
```

## seconds epoch

Seconds from 01-01-1970 00:00.

```
1  #!/usr/bin/env bash
2
3  timeNowSecondsEpoch=`date +%s`
4  echo "$timeNowSecondsEpoch" # 1545223678
```

## array

Contains Array related operations.

## declare

Declare a literal array.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4
5  for item in ${myArray[@]}; do
6      echo "$item"
7  done
8
9  # Alice
10 # Bob
11 # Eve
```

## add | push

Add a new item to the array.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  myArray+=("Shellman")
5
6  for item in ${myArray[@]}; do
7      echo "$item"
8  done
9
10 # Alice
11 # Bob
12 # Eve
13 # Shellman
```

## all

All items of array.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  echo ${myArray[@]} # Alice Bob Eve
```

## at index

Returns item Nth from array (N = index).

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve" "Shellman")
4  echo ${myArray[2]} # Eve
```

## concat

Returns an array made of concatenation of two given arrays.

```
1  #!/usr/bin/env bash
2
3  array1=("Alice" "Bob" "Eve")
4  array2=("1" "2" "3")
5  newArray=("${array1[@]}" "${array2[@]}")
6  echo "${newArray[@]}" # Alice Bob Eve 1 2 3
```

## delete

Delete entire array.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  unset myArray
5  echo "${myArray[@]}"
6  #
```

## delete at

Delete Nth item in array (N = index)

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  unset myArray[1]
5  echo "${myArray[@]}" # Alice Eve
```

## filter

Filter elements of an array based on given pattern.

```
1  #!/usr/bin/env bash
2
3  myArray=('Alice' '22' 'Bob' '16' 'Eve')
4  filtered=(`for i in ${myArray[@]} ; do echo $i; done | grep [0-9]`)
5  echo "${filtered[@]}" # 22 16
```

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  echo ${myArray[@]/e/} # Alice Eve
```

## iterate | for each

Iterate over array items.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4
5  for item in ${myArray[@]}; do
6     echo "$item"
7 done
8
9  # Alice
10 # Bob
11 # Eve
```

## length

Returns length of array.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  echo ${#myArray[@]} # 3
```

## range

Return items from *index* up to the *count*.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve" "Shellman" "Remisa")
4  echo ${myArray[@]:1:3} # Bob Eve Shellman
```

In above example we are interested in 3 items of array starting at index 1 (arrays are zero base indexed)

## replace

Find and replace items in array based on regex.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  echo ${myArray[@]//e/9} # Alic9 Bob Ev9
```

## set elements

Set element given value as Nth element.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve")
4  myArray[1]="Shellman"
5  echo ${myArray[@]} # Alice Shellman Eve
```

## slice

Returns a subarray starting at *index* and containing *count* items of the original array.

```
1  #!/usr/bin/env bash
2
3  myArray=("Alice" "Bob" "Eve" "Shellman")
4  echo ${myArray[@]:1:2} # Bob Eve
```

## directory

Contains String related operations.

### create

Creates a directory.



```
1 mkdir "test dir"
```

Creates test directory at the current path.

## create nested

Create directories as required.

```
1 #!/usr/bin/env bash
2
3 mkdir -p "parent dir"/"child dir"
```



## Nested Directories

Write a shell script to create a test directory containing 26 directories named from a to z each containing 100 directories from 1 to 100 with a single command.

Directory structure should look like:

```
1  — test
2  |   └─ a
3  |   |   └─ 1
4  |   |   └─ 2
5  |   |   .
6  |   |   .
7  |   |   └─ 100
8  |   └─ b
9  |   |   └─ 1
10 |   |   └─ 2
11 |   |   .
12 |   |   .
13 |   |   └─ 100
14 |   └─ c
15 |   .
16 |   .
```

For the answer refer to [Solutions](#) section, [nested directories](#).

## find

Find files or directories based on *criteria* in the given path up to N level depth.

```
1  #!/usr/bin/env bash
2
3  result=`find . -maxdepth 3 -type f -name "*.txt"`
4  echo "$result"
```

Above example finds all files (-type f) up to 3 level depth (-maxdepth 3) in the current directory (.).

## function

Contains function related operations available through **func** namespace. A function can return a number between 0 to 255 which can be retrieved through \$? (available as func ret val snippet).

## func

Define a function to be called later. Function definition must precede its usage.

```
1  #!/usr/bin/env bash
2
3  function myFunction () {
4      echo "$1"
5      echo "$2"
6  }
7
8  myFunction "some argument" "another argument"
9  # some argument
10 # another argument
```

## args

Access to function arguments.

```
1  #!/usr/bin/env bash
2
3  function myFunction () {
4      echo "$@"
5  }
6
7  myFunction "some argument" "another argument"
8  # some argument another argument
```

## args count

Number of function arguments.

```
1  #!/usr/bin/env bash
2
3  function myFunction () {
4      echo $#
5  }
6
7  myFunction "some argument" "another argument"
8  # 2
```

## ret val

Check the value last function call has returned (0-255). By convention, zero is returned if no error occurs, otherwise a non-zero value is returned.

```
1  #!/usr/bin/env bash
2
3  function test () {
4      echo "$1"
5      return 25
6  }
7
8  test "return value"
9  echo "$?"
10 # return value
11 # 25
```

## command

Contains command execution related operations available through **cmd** namespace.

### cmd

To run a command and use the returned value is named [command substitution](#).

```
1  #!/usr/bin/env bash
2
3  response=`curl -s http://example.com`
4  echo "$response"
```

In above example using `curl` we retrieve the content of `http://example.com` and store it in `response` variable (`-s` flag tells `curl` to work in silent mode).

### success check

Check if last command has succeeded.

```
1  #!/usr/bin/env bash
2
3  ls # this command will succeed
4
5  if [[ $? == 0 ]]; then
6      echo "command succeeded"
7  else
8      echo "command failed"
9  fi
10 # command succeeded
```

### failure check

Check if last command has failed.

```
1  #!/usr/bin/env bash
2
3  touch /file.txt # this command will fail without sudo
4
5  if [[ $? == 0 ]]; then
6      echo "command succeeded"
7  else
8      echo "command failed"
9  fi
10 # command failed
```

## nice

Run a command with modified scheduling priority. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process) and default value is 10.

```
1  #!/usr/bin/env bash
2
3  sudo nice -n 19 cp ~/file ~/tmp
```

In above example we are copying a file from *home* to *tmp* folder, and schedule minimum CPU time to cp.

## if cmd exists

Check if a desired command exists (program is installed).

```
1  #!/usr/bin/env bash
2
3  if [ `command -v docker` ]; then
4      echo "docker is installed"
5  else
6      echo "docker is NOT installed"
7  fi
```

In above example we are checking if docker program is available on the system.

## archive

Contains archive related operations like compressing and decompressing files/directories.

## compress tar.gz

Compress file(s)/director(ies) into a compressed archive file (.tar.gz)

```
1  #!/usr/bin/env bash
2
3  tar -czvf ~/archive.tar.gz ~/some-directory
```

In above example we are compressing and archiving a directory (some-directory) from our *home* into archive.tar.gz file in our *home* directory. This is useful for example if we are interested to backup some-directory.

## decompress tar.gz

Decompress an archive file (.tar.gz) into a path.

```
1  #!/usr/bin/env bash
2
3  tar -C ~/ -xzf ~/archive.tar.gz
```

In above example we are decompressing archive.tar.gz file from our *home* directory into our *home* directory.

## crypto

Contains Cryptography related operations like encryption, decryption and hashing.

### base64 encode

Encode variable content into *base64*.



#### Base64

This encoding is used to transform *binary* data into *string* usually to save in a file or transfer over network.

```
1  #!/usr/bin/env bash
2
3  base64Encoded=`echo -n "$variableToEncode" | base64`
```

## base64 decode

Decode String from *base64* into Binary.

```
1  #!/usr/bin/env bash
2
3  base64Decoded=`echo -n "$variableToDecode" | base64 -d`
```

## hash

Hash variable content with desired algorithm.

```
1  #!/usr/bin/env bash
2
3  hash=`echo -n "$variableToHash" | md5sum | cut -f1 -d ' '`
4  echo "$hash"
```

Supported algorithms:

- md5
- sha
- sha1
- sha224
- sha256
- sha384
- sha512

## http

Contains HTTP related operations.

### GET

Send a *GET* request to specified *URL*.

```
1  #!/usr/bin/env bash
2
3  curl --request GET -sL \
4      --user-agent 'Shellman' \
5      --url 'http://example.com'
```

Above example sends a *HTTP GET* request to <http://example.com> with desire *User Agent*<sup>8</sup>.

## DELETE

Send a *DELETE* request to specified *URL*.

```
1  #!/usr/bin/env bash
2
3  curl --request DELETE -sL \
4      --user-agent 'Shellman' \
5      --url 'http://example.com'
```

## POST

Send a *POST* request to specified *URL*.

```
1  #!/usr/bin/env bash
2
3  curl --request POST -sL \
4      --user-agent 'Shellman' \
5      --url 'http://example.com' \
6      --data 'key1=value1' \
7      --data 'key2=value2'
```

## POST file

Send file with *http POST*.

---

<sup>8</sup>[https://en.wikipedia.org/wiki/User\\_agent](https://en.wikipedia.org/wiki/User_agent)



```
1  #!/usr/bin/env bash
2
3  curl --request POST -sL \
4      --user-agent 'Shellman' \
5      --url 'http://example.com' \
6      --form 'key=value' \
7      --form 'file=@~/image.jpg'
```

Above example sends `image.jpg` to <http://example.com> via *POST* method.

## header

Send http request with custom header(s).

```
1  #!/usr/bin/env bash
2
3  curl --request GET -sL \
4      --user-agent 'Shellman' \
5      --header 'key: value' \
6      --url 'http://example.com'
```

## cookie

Send http request with desired cookies.

```
1  #!/usr/bin/env bash
2
3  curl --request GET -sL \
4      --user-agent 'Shellman' \
5      --cookie 'key=value' \
6      --url 'http://example.com'
```

## download

Download from url and save to desired *path*.

```
1  #!/usr/bin/env bash
2
3  curl --request GET -sL \
4      --user-agent 'Shellman' \
5      --output '~/downloaded-file.zip' \
6      --url 'http://example.com/file.zip'
```

## ftp

Contains FTP related operations.

### list

Get the list of files on the ftp server at specific path.

```
1  #!/usr/bin/env bash
2
3  curl ftp://remisa:1234@mydomain/backup/
```

### download

Download specified file from ftp server.

```
1  #!/usr/bin/env bash
2
3  curl ftp://remisa:1234@mydomain/backup/latest.zip
```

### upload

Upload specified file to ftp server at desired path.

```
1  #!/usr/bin/env bash
2
3  curl -T test.zip ftp://remisa:1234@mydomain/backup/
```

### delete file

Delete specified file from ftp server.

```
1  #!/usr/bin/env bash
2
3  curl ftp://remisa:1234@mydomain/backup/test.zip -Q "DELE test.zip"
```

## rename

Rename specified file/directory on ftp server.

```
1  #!/usr/bin/env bash
2
3  curl ftp://remisa:1234@mydomain/backup/ -Q "-RNFR backup/test.zip" -Q "-RNT0 backup/\
4  renamed.zip"
```

## file

Contains File related operations.

### file delete | file remove

Delete given file.

```
1  #!/usr/bin/env bash
2
3  rm -f ~/test.txt
```

In above example `test.txt` will be deleted from *home*.

### file find

Find files or directories based on criteria in given path.

```
1  #!/usr/bin/env bash
2
3  result=`find ~ -maxdepth 3 -type f -name "*.txt"`
4  echo "$result"
```

In above example all files (`-type f`) with `txt` extension in *home* (`~`) path up to 3 level of depth will be found. To search for directories use `-type d`.

### file search

Find files which contain the search criteria.

```
1  #!/usr/bin/env bash
2
3  result=`find ~ -maxdepth 1 -type f -exec grep "ls" {} +`
4  echo "$result"
```

In above example we will search all files in *home* (~) directory up to 1 depth level, and find the ones which contain text `ls`.

## file read

Read contents of a file line by line.

```
1  #!/usr/bin/env bash
2
3  cat ~/test.txt | while read line; do
4      echo "$line"
5  done
```

In above example we read contents of `test.txt` which is in user *home* directory, and print it line by line.

## file write

Write to a file.

```
1  #!/usr/bin/env bash
2
3  lines=`docker images`
4  echo "sample header" > ~/test.txt
5  for line in ${lines}; do
6      echo "$line" >> ~/test.txt
7  done
```

In above example we store result of `docker images` command in `lines` variable then send sample header text to `test.txt` file in *home* (~) directory. Inside for loop we send each line of `lines` to `test.txt`.

Operator `>` redirects output to a file and overwrite its content while operator `>>` will append to the contents of the file.

## file write multiline

Write multiple lines into file.

```
1  #!/usr/bin/env bash
2
3  cat >~/test.txt <<EOL
4  Header
5
6  first line
7  second line
8  EOL
```

## file write multiline sudo

Write multiple lines into a file which needs root permission.

```
1  #!/usr/bin/env bash
2
3  cat << EOL | sudo tee /test.txt
4  Header
5
6  first line
7  second line
8  EOL
```

## remove files older than

Remove files older than x days.

```
1  #!/usr/bin/env bash
2
3  find ~/backup -mtime +14 | xargs rm -f
```

Above example removes files from ~/backup directory which are older than two weeks.

## color

Write text in color. *color namespace* contains commands to write in different foreground colors. To write in color we use *tput setaf* command followed by *color code*. Here is color code table:

Color	Code
Black	0
Red	1
Green	2
Yellow	3
Blue	4
Magenta	5
Cyan	6
White	7

To set *foreground color* to red we use `tput setaf 1` command and after some output we use `tput sgr0` command to set everything to default. So for writing *hello world* in red:

```
1  #!/usr/bin/env bash
2
3  echo `tput setaf 1`hello world`tput sgr0`
```



## Colorful Text

Write a shell script that prints *Hello World!* in all 8 colors using a *for loop*.

For the answer refer to [Solutions](#) section, [colorful text](#).

## format

Write text in italic, bold, dim or reverse contrast.

```
normal text
bold text
italic text
dimmed text
reversed text
```

formatted text

## bold

Write in **bold**.

```
1  #!/usr/bin/env bash
2
3  echo `tput bold`bold text`tput sgr0`
```

## italic

Write in *italic*.

```
1  #!/usr/bin/env bash
2
3  echo `tput sitm`italic text`tput sgr0`
```

## dim

Write dim text.

```
1  #!/usr/bin/env bash
2
3  echo `tput dim`dimmed text`tput sgr0`
```

## reverse

Write text in reverse contrast.

```
1  #!/usr/bin/env bash
2
3  echo `tput rev`reversed text`tput sgr0`
```

## system

Contains System related information and operations.

## uptime

System uptime (hh:mm:ss).

```
1  #!/usr/bin/env bash
2
3  sys_uptime=`uptime | cut -d ' ' -f2`
4  echo "$sys_uptime" # 03:26:47
```

## memory info

System memory information in kilobytes (KB).

```
1  #!/usr/bin/env bash
2
3  sysMemoryMemTotal=`cat /proc/meminfo | grep 'MemTotal' | awk '{print $2}' | head -n \
4  1`
5  echo "$sysMemoryMemTotal" # total system memory in KB
```

## distro name

Operating System ID (i.e. Ubuntu).

```
1  #!/usr/bin/env bash
2
3  distroName=`lsb_release -i | awk '{print $3}'`
4  echo "$distroName"
```

## distro version

Operating System release version (i.e. 16.04).

```
1  #!/usr/bin/env bash
2
3  distroName=`lsb_release -r | awk '{print $2}'`
4  echo "$distroName"
```

## distro codename

Operating System codename (i.e. xenial).



```
1  #!/usr/bin/env bash
2
3  distroName=`lsb_release -c | awk '{print $2}'`
4  echo "$distroName"
```

## kernel name

Operating System kernel name (i.e. Linux).

```
1  #!/usr/bin/env bash
2
3  kernelName=`uname -s`
4  echo "$kernelName" # Linux
```

## kernel release

Operating System kernel release (i.e. 4.4.0-140-generic).

```
1  #!/usr/bin/env bash
2
3  kernelRelease=`uname -r`
4  echo "$kernelRelease" # 4.4.0-140-generic
```

## processor type

Operating System processor type (i.e. x86\_64).

```
1  #!/usr/bin/env bash
2
3  processorType=`uname -p`
4  echo "$processorType" # x86_64
```

## processor count

Number of processors (cores).

```
1  #!/usr/bin/env bash
2
3  processorCount=$(lscpu | grep 'CPU(s)' | awk '{print $2}' | head -n 1`
4  echo "$processorCount" # 4
```

## processor architecture

Processor architecture (i.e. x86\_64).

```
1  #!/usr/bin/env bash
2
3  processorArchitecture=$(lscpu | grep 'Architecture' | awk '{print $2}' | head -n 1`
4  echo "$processorArchitecture" # x86_64
```

## processor model

Processor model name (i.e. Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz).

```
1  #!/usr/bin/env bash
2
3  processorModel=$(lscpu | grep 'Model name' | cut -d ' ' -f 3- | sed -e 's/^[[[:space:]]\
4  */'`
5  echo "$processorModel" # Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
```

## git

Contains git commands. You may need to install git on your system. Git is a version control system for tracking changes of projects.

Install git:

- Debian-based linux systems
  - `sudo apt install git`
- Red Hat-based linux systems
  - `sudo yum install git`
- Mac
  - `brew install git`
- Windows
  - Download from <https://gitforwindows.org/>

## clone

Clone a repository to local machine.

```
1  #!/usr/bin/env bash
2
3  git clone https://github.com/user/repository.git
4  cd repository
```

## clone branch

Clone a repository to local machine and switch to a specific branch.

```
1  #!/usr/bin/env bash
2
3  git clone -b develop https://github.com/user/repository.git
4  cd repository
```

## config list

List git configurations.

```
1  #!/usr/bin/env bash
2
3  git config --list
```

## config set

Set a *git* configuration.

```
1  #!/usr/bin/env bash
2
3  git config --global user.name "Remisa"
```

## commit

Commit changes.

```
1  #!/usr/bin/env bash
2
3  git commit -m "fixed typo"
```

## commit search

Search for a commit which contains searchCriteria.

```
1  #!/usr/bin/env bash
2
3  git log --all --grep='typo'
```

## commit undo

Undo last N commits. **soft** preserve local changes. **hard** delete local changes.

```
1  #!/usr/bin/env bash
2
3  git reset --soft HEAD~1 # undo last local change but don't delete them
```

## commit list notPushed

List non pushed commits.

```
1  #!/usr/bin/env bash
2
3  git log origin/master..HEAD
```

## branch create

Create a local branch and switch into it.

```
1  #!/usr/bin/env bash
2
3  git checkout -b develop
```

## branch list

List all branches.

```
1  #!/usr/bin/env bash
2
3  git branch
```

## branch push

Push branch to remote.

```
1  #!/usr/bin/env bash
2
3  git push origin develop
```

## branch rename

Rename current branch.

```
1  #!/usr/bin/env bash
2
3  git branch -m newName
```

## branch delete local

Delete local branch.

```
1  #!/usr/bin/env bash
2
3  git branch --delete localBranch
```

## branch delete remote

Delete remote branch.

```
1  #!/usr/bin/env bash
2
3  git push origin --delete remoteBranch
```

## changes revert

Revert tracked changes.

```
1  #!/usr/bin/env bash
2
3  git checkout .
```

## patch create

Create a patch from changes.

```
1  #!/usr/bin/env bash
2
3  git diff > patch1.patch
```

## patch apply

Apply a patch from file.

```
1  #!/usr/bin/env bash
2
3  git apply < patch1.patch
```

## remote list

List all remotes.

```
1  #!/usr/bin/env bash
2
3  git remote
```

## remote urlAdd

Add remote url.

```
1  #!/usr/bin/env bash
2
3  git remote add origin https://github.com/user/repository.git
```

## remote urlChange

Change remote url.

```
1 #!/usr/bin/env bash
2
3 git remote set-url origin https://github.com/user/repository.git
```

## tag list

List all tags.

```
1 #!/usr/bin/env bash
2
3 git tag
```

## tag commit

Tag a commit.

```
1 #!/usr/bin/env bash
2
3 git tag -a release/1.0.0 -m "1.0.0 release"
```

## tag remote delete

Delete tag from remote.

```
1 #!/usr/bin/env bash
2
3 git push --delete origin tagName && git push origin :tagName
```

## tag remote push

Push tag to remote.

```
1 #!/usr/bin/env bash
2
3 git push origin tagName
```

## miscellaneous

Contains other operations not available in namespaces.

### switch case

This is the `switch / case` you may be familiar in other languages. You can define different actions based on `switch`:

```
1  #!/usr/bin/env bash
2
3  var=2
4
5  case "$var" in
6      1)
7          echo "case 1"
8          ;;
9      2|3)
10         echo "case 2 or 3"
11         ;;
12     *)
13         echo "default action"
14         ;;
15  esac
16  # case 2 or 3
```

In above example we are deciding on the value of `var` which here is 2. If `var` is 2 or 3 the second case will be triggered. If none of cases (1, 2 or 3) are triggered, `*` means default and that will be triggered. change `var` to 5 and output will be `default action`.

### let

`let` is used for mathematic operations.





```
1  #!/usr/bin/env bash
2
3  read -ep "What is your name? " -i Remisa ANSWER
4  echo "$ANSWER" # print user's answer
```

## timeout

Run a command within a time frame.

```
1  #!/usr/bin/env bash
2
3  timeout 5 curl -s http://example.com
4  echo "at most 5 seconds later"
```

## ips

Array of local IPs.

```
1  #!/usr/bin/env bash
2
3  IPS=`hostname -I`
4  echo "$IPS"
```

## ip info

public ip information (ip, city, region, country, location, postal code, organization).

```
1  #!/usr/bin/env bash
2
3  echo `curl -s ipinfo.io/country`
4  # U.K
```

## ip public

Find public ip address via different services.

- [bot.whatismyipaddress.com](http://bot.whatismyipaddress.com)
- [ident.me](http://ident.me)
- [ipecho.net/plain](http://ipecho.net/plain)
- [icanhazip.com](http://icanhazip.com)
- [ifconfig.me](http://ifconfig.me)
- [api.ipify.org](http://api.ipify.org)
- [ipinfo.io/ip](http://ipinfo.io/ip)

```
1  #!/usr/bin/env bash
2
3  PUBLIC_IP=`curl -s api.ipify.org`
4  echo "$PUBLIC_IP"
```

## service manager

Commands related to *services*. A *service* is a program which runs in background.

```
1  #!/usr/bin/env bash
2
3  sudo systemctl restart service
```

## sleep

Halt script for desired period in seconds *s*, minutes *m*, hours *h*, days *d*.

```
1  #!/usr/bin/env bash
2
3  sleep 2m
4  # halts script for 2 minutes
```

## stopwatch

Use *stopwatch* to calculate script running time.

```
1  #!/usr/bin/env bash
2
3  # beginning of script
4  STOPWATCH_START_TIME=$(date +%s)
5
6  # script
7  sleep 30s
8
9
10 # end of script
11 STOPWATCH_END_TIME=$(date +%s)
12
13 # print elapsed time
```

```

14 STOPWATCH_ELAPSED_TOTAL_SECONDS=$((STOPWATCH_END_TIME - STOPWATCH_START_TIME))
15 STOPWATCH_ELAPSED_MINUTES=$((STOPWATCH_ELAPSED_TOTAL_SECONDS / 60))
16 STOPWATCH_ELAPSED_SECONDS=$((STOPWATCH_ELAPSED_TOTAL_SECONDS % 60))
17 echo elapsed $STOPWATCH_ELAPSED_MINUTES minutes and $STOPWATCH_ELAPSED_SECONDS seconds
18 ds

```

## lib

Contains a set of library functions and calling them under `fn` / `fx` namespaces. Functions can be accessed through `fn...` and their usage through `fx...`

## Math

Math related functions.

### math sum

Calculates sum of given integers. Available as `fn math sum` snippet.

Example usage:

```

1  #!/usr/bin/env bash
2
3  function sum () {
4      local result=0
5      for item in $@; do
6          ((result += item))
7      done
8      echo $result
9  }
10
11  var1=2; var2=5; var3=4
12  result=`sum $var1 $var2 $var3`
13  echo $result

```

### math product

Calculates product of given integers. Available as `fn math product` snippet.

Example usage:

```

1  #!/usr/bin/env bash
2
3  function product () {
4      local result=1
5      for item in $@; do
6          ((result *= item))
7      done
8      echo $result
9  }
10
11  var1=2; var2=3; var3=5
12  result=`product $var1 $var2 $var3`
13  echo $result

```

## math average

Calculates average of given integers. Available as `fn math average` snippet.

Example usage:

```

1  #!/usr/bin/env bash
2
3  function average () {
4      local result=0
5      for item in $@; do
6          ((result += item))
7      done
8      echo $((result / $#))
9  }
10
11  var1=2; var2=3; var3=4
12  result=`average $var1 $var2 $var3`
13  echo $result

```

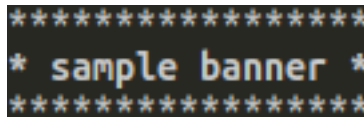
## Misc

Other useful functions.

### banner simple

A function to print simple banners. To define the function use `fn banner simple` at the top of script so later it can be called via `fx banner simple`:

```
1  #!/usr/bin/env bash
2
3  function banner_simple() {
4      # function body...
5  }
6
7  # call function
8  banner_simple "sample banner"
```

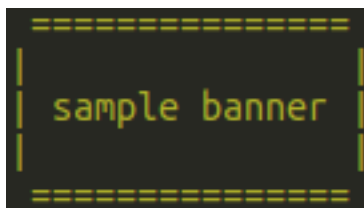


simple banner

## banner color

A function to print colorful banners. To define the function use `fn banner color` at the top of script so later it can be called via `fx banner color`:

```
1  #!/usr/bin/env bash
2
3  function banner_color() {
4      # function body...
5  }
6
7  # call function
8  banner_color yellow "sample banner"
```



color banner

## import

Use functions defined in other bash script files inside your script. To define the function use `fn import` at the top of script so later it can be called via `fx import`. Default folder for library files is `lib` relative to calling script.

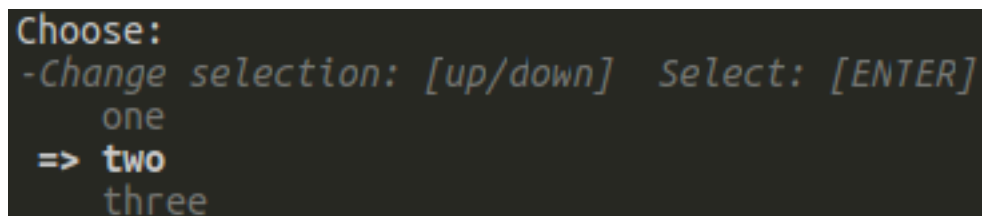
```
1  #!/usr/bin/env bash
2
3  function import() {
4      # function body...
5  }
6
7  import "mylib"
8
9  # Call some function from mylib.sh
```

In above example with `import "mylib"` we are importing functions defined in `lib/mylib.sh`.

## options

A function to print multi choice questions. To define the function use `fn options` at the top of script so later it can be called via `fx options` with question and choices. Default choice is zero based so 0 means first option, 1 means second...

```
1  #!/usr/bin/env bash
2
3  function chooseOption() {
4      # function body...
5  }
6
7  options=("one" "two" "three") # array of options
8  chooseOption "Choose:" 1 "${options[@]}"; choice=$? # call function
9  echo "${options[$choice]}" selected # print selected item by user
```



```
Choose:
-Change selection: [up/down] Select: [ENTER]
  one
=> two
  three
```

options 1

```
Choose:
-Change selection: [up/down]  Select: [ENTER]
  one
  two
=> three
```

options 2

```
Choose:
-Change selection: [up/down]  Select: [ENTER]
  one
  two
=> three

three selected
```

choice

## progress

A dummy progress bar. You can use it as real progress bar with a little change.

```
1  #!/usr/bin/env bash
2
3  function progressBar() {
4      # function body...
5  }
6
7  progressBar .2 "Installing foo..."
```

```
|██████████| 35% [ Installing foo... ]
```

progress

## scan

Scan a host port range (tcp/udp).



```
1  #!/usr/bin/env bash
2
3  function scan () {
4      # function body...
5  }
6
7  # scan tcp ports 5000-10000 of localhost
8  scan tcp localhost 5000 10000
9
10 # tcp 8081 => open
11 # tcp 9000 => open
```

# Advanced

## piping

To do...

## shift command

Argument parsing

# Solutions

# Argument Parsing

Contents of `greet.sh`:

## Argument Parsing

[illegible]

## Nested Directories

Contents of nested-directories.sh:

Nested Directories

---

```
1  #!/usr/bin/env bash
2
3  mkdir -p test/{a..z}/{1..100}
```

---

## Colorful Text

Contents of colorful-text.sh:

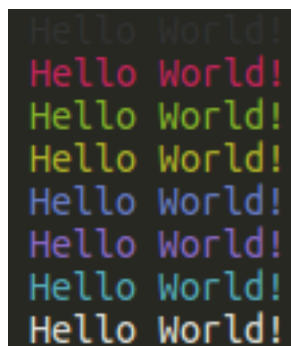
Nested Directories

---

```
1  #!/usr/bin/env bash
2
3  for((i=0;i<=7;i++)); do
4      echo `tput setaf $i`Hello World!`tput sgr0`
5  done
```

---

Output:



```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

colorful text

## Factorial

Contents of factorial.sh:

**Factorial**

---

```
1  #!/usr/bin/env bash
2
3  function fact () {
4      result=1
5      for((i=2;i<=$1;i++)); do
6          result=$((result * i))
7      done
8      echo $result
9  }
10
11 # example: 4! = 4 * 3 * 2 = 24
12 fact 4
```

---