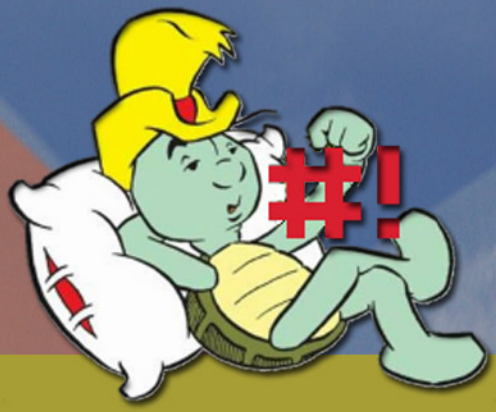
Shellman



#!BASH SCRIPTING

Remisa Yousefvand

Shellman Bash Scripting

Remisa Yousefvand

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

This version was published on 2019-02-10

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.

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

Contents

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 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[1].

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

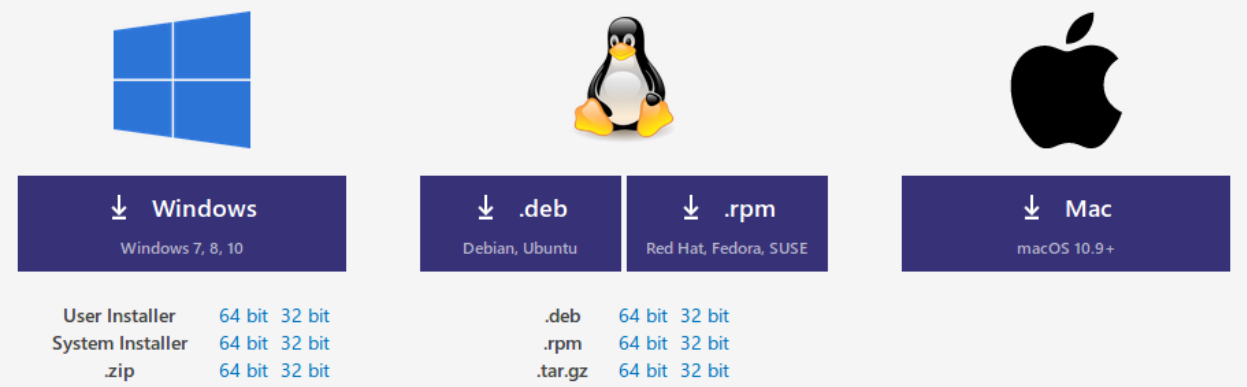
Remisa Yousefvand

January 2019

[1] https://medium.com/@remisa.yousefvand/shellman-reborn-f2cc948ce3fc

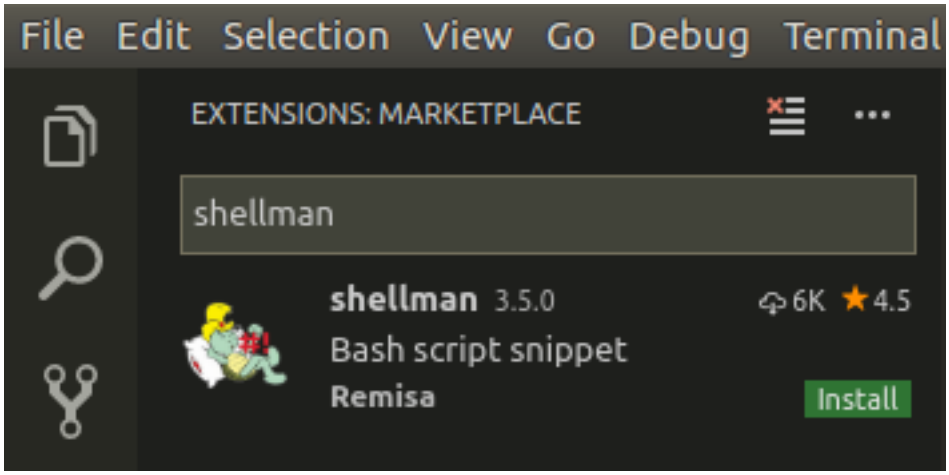
Prerequisites

- vscode[2] IDE



vscode download

- Shellman[3] snippet

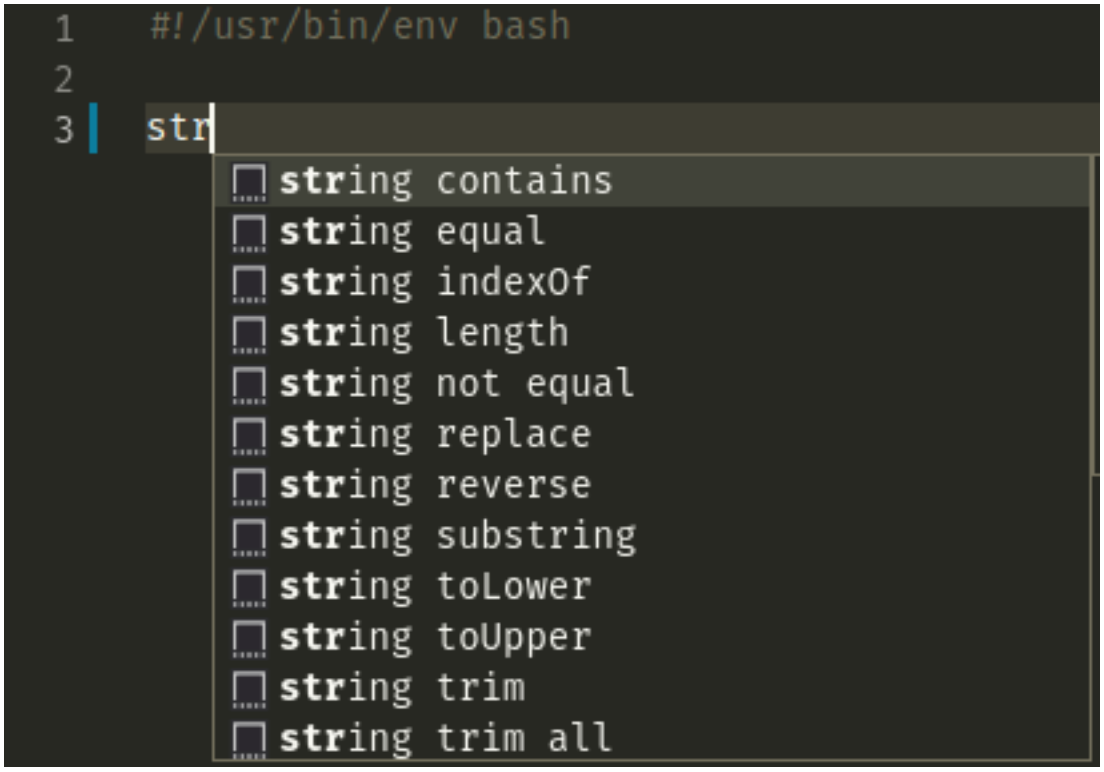


shellman install

[2] https://code.visualstudio.com
[3] 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.



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.

```
# 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:

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

Run a Bash Script

Bash script files by convention has **.sh** *file extension*[4]. 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:

[4] 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*)

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

rm some_file
```

If the command need **root**[5] privileges (in *Windows* it is known as *Admin*), prefix the command with **sudo**:

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

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`.

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

curl --request GET -sL \
  --user-agent 'Shellman' \
  --cookie 'key=value' \
  --url 'http://example.com'
```

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

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

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`:

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

```
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:

```
firstName=Remisa
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*:

```
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 `""`:

```
fullName="Remisa Yousefvand"
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:

```
a="Hello"
b="world"
c="!"
echo "$a $b$c"
# 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:

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

: "${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`:

```
output=`ls` # store ls results in a variable named output

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[6]. 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`:

```
echo "$?"
```

Shellman supports checking **failure** of last command via `cmd` namespace as `cmd failure check` snippet:

```
# following command will fail due to lack of permission
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.

```
touch /not_enough_permission_to_create_file

# check last command (touch) success/failure
if [[ $? != 0 ]]; then
  echo "command failed"
fi
```

To check **success**, use `cmd success check` snippet from `cmd` namespace:

[6] 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.

```
echo "Hello World!"

# check last command (echo) success/failure
if [[ $? == 0 ]]; then
  echo command succeed
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:

```
./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:

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

In above code we are telling the script to save the output in `~/my_backups`[7] 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:

[7] ~ is a shorthand for current user, *home directory*, which usually is `/home/username`.

```
POSITIONAL=()
while [[ $# > 0 ]]; do
  case "$1" in
    -f|--flag)
    echo flag: $1
    shift # shift once since flags have no values
    ;;
    -s|--switch)
    echo switch $1 with value: $2
    shift 2 # shift twice to bypass switch and its value
    ;;
    *) # unknown flag/switch
    POSITIONAL+=("$1")
    shift
    ;;
  esac # end of case. "case" word in reverse!
done

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:

```
-v|--verbose)
verbose=true
```

Repeat above procedure for more flags.

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

```
-o|--output)
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:

```
./greet.sh
# good night everyone

./greet.sh -m
# good morning everyone

./greet.sh --name Remisa
# good night Remisa

./greet.sh -n Remisa
# good night Remisa

./greet.sh -m --name Remisa
# good morning Remisa

./greet.sh -m -n Remisa
# 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`.

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

# Title:         test
# Description:   a test script
# Author:        Remisa <remisa.yousefvand@gmail.com>
# Date:          2019-01-06
# Version:       1.0.0
```

Use `region` snippet to define a `functions` region and put all of your functions there. Remember you need to define functions before you can use them so it is a good idea to put them on the top of the script (after summary). If function `B` calls function `A`, then function `A` definition should precede definition of function `B`.

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

# summary here

# >>>>>>>>>>>>>>>>>>>>>>> functions >>>>>>>>>>>>>>>>>>>>>>>

function greet() {
  # access the argument via $1
  echo "Hello $1"
}

# <<<<<<<<<<<<<<<<<<<<<<< functions <<<<<<<<<<<<<<<<<<<<<<<

greet "Shellman" # call the function and pass an argument
```

Double Quote vs Single Quote vs Backtick

Use *double quotation* where you have a variable that contains *whitespace*. Any variable inside a double quotation will be replaced by its value:

```
var1="Hello World!"
echo "$var1" # Hello World!
```

Double Quote

By default use Double Quote " when defining variable or trying to access a variable value.

Use *single quotation* where you need to define a variable that contains special characters. Anything inside a single quotation will remain exact the same:

```
var1="Hello World!"
echo "$var1" # Hello World!

var2='$var1'
echo "$var2" # $var1

var3='"&$*'
echo "$var3" # "&$*
```

Use *backtick* for command substitution

```
directoryList=`ls | xargs echo`
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 `(( ))`.

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

a=3
while (( a > 0 )); do
  echo "$a"
  ((a--))
done
# 3
# 2
# 1
```

For string comparison use `[ ]`.

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

str="s"
while [ "$str" != "end" ]; do
  echo "start"
  str="end"
done
# start
```

until

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

For arithmetic comparison use `(( ))`.

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

a=3
until (( a <= 0 )); do
  echo "$a"
  ((a--))
done
# 3
# 2
# 1
```

For string comparison use `[ ]`.

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

str="s"
until [ "$str" == "end" ]; do
  echo "start"
  str="end"
done
# start
```

for i

`for` loop.

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

for((i=0;i<5;i++)); do
  echo "$i"
done
# 0
# 1
# 2
# 3
# 4
```

for i j

Nested `for` loop.

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

for((i=0;i<3;i++)); do
  for((j=0;j<2;j++)); do
    echo "$i, $j"
  done
done
# 0, 0
# 0, 1
# 1, 0
# 1, 1
# 2, 0
# 2, 1
```

for in

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

Numerical range:

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

for item in {1..5}; do
  echo "$item"
done
# 1
# 2
# 3
# 4
# 5
```

alphabetical range:

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

for item in {A..D}; do
  echo "$item"
done
# A
# B
# C
# 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?

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

for col in `docker images | awk '{ print $1}'`; do
  echo "$col"
done
```

Output of above script is:

```
REPOSITORY
sonatype/nexus3
sonatype/nexus3
busybox
busybox
```

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

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

for col in `docker images | awk '{ print $2}'`; do
  echo "$col"
done
```

Output would be:

```
TAG
3.13.0
latest
glibc
latest
```

logic

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

if

`if`, `else` condition.

For arithmetic comparison use `(( ))`.

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

var1=32
var2=33

if (( $var1 == $var2 )); then
  echo "equal"
elif (( $var1 >= $var2 )); then
  echo "bigger"
else
  echo "smaller"
fi
# smaller
```

For string comparison use `[ ]`.

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

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

str1="bye"
str2="hello"
str3="bye"

if [ "$str1" = "$str2" ]; then
  echo "1 = 2"
elif [ "$str1" = "$str3" ]; then
  echo "1 = 3"
elif [ "$str2" = "$str3" ]; then
  echo "2 = 3"
else
  echo "no equal pair"
fi
# 1 = 3
```

Simpler forms of `if`:

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

str1="bye"
str2="hello"

if [ "$str1" = "$str2" ]; then
  echo "equal"
fi
```

or `if/else`:

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

str1="bye"
str2="hello"

if [ "$str1" = "$str2" ]; then
  echo "equal"
else
  echo "NOT equal"
fi
# NOT equal
```

iff

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

For arithmetic comparison use `(( ))`.

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

var=5
(( var > 3 )) && echo "greater than 3"
# greater than 3
```

For string comparison use `[ ]`.

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

var="hi"
[ "$var" = "hi" ] && echo "hi"
# hi
```

iff not

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

For arithmetic comparison use `(( ))`.

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

var=5
(( var > 8 )) || echo "less than 8"
# less than 8
```

For string comparison use `[ ]`.

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

var="hi"
[ "$var" = "bye" ] || echo "hi"
# hi
```

if directory exists

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

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

if [ -d ~/backup ]; then
  echo "backup exists"
fi
```

if cmd exists

Read cmd

if exists

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

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

path=~/.bashrc
if [ -e "$path" ]; then
  echo exists
fi
```

if file =

Check if two files are equal.

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

file1=~/some_file
file2=~/another_file

if [ "$file1" -ef "$file2" ]; then
  echo files are equal
fi
```

if file executable

Check if file is executable.

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

if [ -x /bin/ls ]; then
  echo file is executable
fi
```

if file link

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

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

if [ -h /vmlinuz ]; then
  echo symbolic link
fi
```

if file newer

Check if first file is newer than the second.

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

file1=~/.bashrc
file2=~/.profile

if [ "$file1" -nt "$file2" ]; then
  echo file1 is newer than file2
fi
```

if file not empty

Check if file is not empty.

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

if [ -s ~/.profile ]; then
  echo file not empty
fi
```

if file older

Check if first file is older than the second.

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

file1=~/.bashrc
file2=~/.profile

if [ "$file1" -ot "$file2" ]; then
  echo file1 is older than file2
fi
```

if file readable

Check if file is readable.

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

if [ -r ~/.profile ]; then
  echo file is readable
fi
```

if file writable

Check if file is writable.

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

if [ -w ~/.profile ]; then
  echo file is writable
fi
```

if int =

Check if two integers are equal.

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

int1=67
int2=67

if (( int1 == int2 )); then
  echo equal
fi
```

if int !=

Check if two integers are not equal.

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

int1=12
int2=13

if (( int1 != int2 )); then
  echo not equal
fi
```

if int <

Check if the first integer is smaller than the second.

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

int1=12
int2=13

if (( int1 < int2 )); then
  echo lesser
fi
```

if int <=

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

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

int1=12
int2=13

if (( int1 <= int2 )); then
  echo less or equal
fi
```

if int >

Check if the first integer is greater than the second.

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

int1=15
int2=13

if (( int1 > int2 )); then
  echo greater
fi
```

if int >=

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

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

int1=12
int2=13

if (( int1 >= int2 )); then
  echo greater or equal
fi
```

if string empty

Check if string is empty.

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

str=""
if [ -z "$str" ]; then
  echo "Empty string"
fi
# Empty string
```

if string not empty

Check if string is not empty.

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

str="a"
if [ -n "$str" ]; then
  echo "String is not empty"
fi
# String is not empty
```

string equal | if string =

Check if strings are equal.

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

str1="hello"
str2="hello"
if [ "$str1" = "$str2" ]; then
  echo "equal"
fi
# equal
```

string not equal | if string !=

Check if strings are not equal.

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

str1="hi"
str2="hello"
if [ "$str1" != "$str2" ]; then
  echo "not equal"
fi
# not equal
```

if string contains

Check if string contains given substring.

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

str="hello world!"

if [[ "$str" = *world* ]]; then
  echo contains world
fi
# contains world
```

string

Contains `String` related operations.

contains

Checks if a `String` contains another `String` (substring).

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

var="hello world!"

if [[ "$var" = *world* ]]; then
  echo "substring found"
else
  echo "substring NOT found"
fi
```

equal

Checks if two `Strings` are the same.

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

string1='This is a string!'
string2='This is a string!'

if [ "$string1" = "$string2" ]; then
  echo 'Strings are equal'
fi
```

indexOf

Returns index of substring inside a string.

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

myString="Hello World!"
temp=${myString%%"or"*} && indexOf=`echo ${myString%%"or"*} | echo ${#temp}`
echo $indexOf # 7
```

length

Returns *length* of given string.

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

var="abcdefg"
length=${#var}
echo "$length"
```

not equal

Checks if two strings are not equal.

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

str1="shellman"
str2="shellmen"
if [ "$str1" != "$str2" ]; then
  echo "Strings are NOT equal"
fi
```

replace

Replace a substring with given string in another string.

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

str1="Hello World!"
replaced=`echo -e "${str1}" | sed -e 's/World/Everyone/g'`
echo "$replaced" # Hello Everyone!
```

reverse

Reverse given string.

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

str1="abcd"
reversed=`echo -e "${str1}" | rev`
echo "$reversed" # dcba
```

substring

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

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

str1="abcdefg"
substring=`echo -e "${str1:2:3}"`
echo "$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.

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

str1="AbCdE"
toLower=`echo -e "${str1}" | tr '[:upper:]' '[:lower:]'`
echo "$toLower" # abcde
```

toUpper

Returns uppercase of given string.

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

str1="AbCdE"
toLower=`echo -e "${str1}" | tr '[:upper:]' '[:lower:]'`
echo "$toLower" # abcde
```

trim

Removes leading and trailing whitespace(s).

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

str1="   result  "
result=`echo -e "${str1}" |  sed -e 's/^[[:space:]]*//' | sed -e 's/[[:space:]]*$//'`
echo "Variable $result contains no leading and trailing space as you see"
# Variable result contains no leading and trailing space as you see
```

trim all

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

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

str1="   ab c de  "
result=`echo -e "${str1}" | tr -d '[[:space:]]'`
echo "All whitespaces are removed from $result as you see"
# All whitespaces are removed from abcde as you see
```

trim left

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

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

str1="   whitespace on left"
result=`echo -e "${str1}" | sed -e 's/^[[:space:]]*//'`
echo "There is no $result as you see"
# There is no whitespace on left as you see
```

trim right

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

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

str1="whitespace on right   "
result=`echo -e "${str1}" | sed -e 's/[[:space:]]*$//'`
echo "There is no $result as you see"
# 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.

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

var1=17
var2=5
reminder=$((var1 % var2))
echo "$reminder" # 2
```

multiply *

Given two numbers, returns product of them.

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

var1=3
var2=4
result=$((var1 * var2))
echo "$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.

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

var1=2
var2=3
result=$((var1 + var2))
echo "$result" # 5
```

increase ++

Given a number, adds one to it.

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

var=7
echo $((++var)) # 8
```

subtract -

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

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

var1=7
var2=5
result=$((var1 - var2))
echo "$result" # 2
```

decrease –

Given a number, subtracts one from it.

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

var=8
echo $((--var)) # 7
```

divide /

Given two numbers, returns first divided by the second.

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

var1=12
var2=4
result=$((var1 / var2))
echo "$result" # 3
```

scale 0.00

Math operations with `x` decimal point precision.

Multiply example:

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

var1="2.13"
var2=""2
result=`echo "scale=2;($var1 * $var2)" | bc`
echo "$result" # 4.26
```

Division example:

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

var1=7
var2=2
result=`echo "scale=2;($var1 / $var2)" | bc`
echo "$result" # 3.50
```

exponentiation ^

Exponentiate *base* to the *power*.

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

echo $((2 ** 4)) # 16
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.

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

var=2
result=`echo "scale=7;sqrt($var)" | bc`
echo "$result" # 1.4142135
```

random

Generate random number between *min* and *max*

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

echo $((5000 + RANDOM % $((65535-5000)))) # 27502
```

constants

Some useful math constants.

date

Contains `Date` related operations.

now short

Short version of current system *date*.

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

dateShort=`date -I`
echo "$dateShort" # 2019-01-06
```

now UTC

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

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

dateUTC=`date -u`
echo "$dateUTC" # Sunday, January 06, 2019
```

now year

Current year.

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

year=`date +%Y`
echo "$year" # 2019
```

now monthNumber

Current month number.

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

monthNumber=`date +%m`
echo "$monthNumber" # 01
```

now monthName

Current month name.

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

monthName=`date +%B` # %B for full month name, %b for abbreviated month name
echo "$monthName" # January
```

now dayOfMonth

Current day of month.

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

dayOfMonth=`date +%d`
echo "$dayOfMonth" # 06
```

now dayOfWeek

Current weekday name.

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

dayOfWeek=`date +%A` # %A for full weekday name, %a for abbreviated weekday name
echo "$dayOfWeek" # Sunday
```

now dayOfYear

Current day of year (1-366).

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

dayOfYear=`date +%j`
echo "$dayOfYear" # 006
```

time

Contains `Time` related operations.

now local

Current local time.

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

timeNowLocal=`date +%R` # %R for 24 hrs
echo "$timeNowLocal" # 13:23

timeNowLocal=`date +%r` # %r for 12 hrs
echo "$timeNowLocal" # 01:23:45
```

now UTC

Current UTC time.

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

timeNowUTC=`date -u +%R`
echo "$timeNowUTC" # 12:56
```

seconds epoch

Seconds from 01-01-1970 00:00.

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

timeNowSecondsEpoch=`date +%s`
echo "$timeNowSecondsEpoch" # 1545223678
```

array

Contains `Array` related operations.

declare

Declare a literal array.

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

myArray=("Alice" "Bob" "Eve")

for item in ${myArray[@]}; do
  echo "$item"
done

# Alice
# Bob
# Eve
```

add | push

Add a new item to the array.

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

myArray=("Alice" "Bob" "Eve")
myArray+=("Shellman")

for item in ${myArray[@]}; do
  echo "$item"
done

# Alice
# Bob
# Eve
# Shellman
```

all

All items of array.

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

myArray=("Alice" "Bob" "Eve")
echo ${myArray[@]} # Alice Bob Eve
```

at index

Returns item Nth from array (N = index).

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

myArray=("Alice" "Bob" "Eve" "Shellman")
echo ${myArray[2]} # Eve
```

concat

Returns an array made of concatenation of two given arrays.

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

array1=("Alice" "Bob" "Eve")
array2=("1" "2" "3")
newArray=("${array1[@]}" "${array2[@]}")
echo ${newArray[@]} # Alice Bob Eve 1 2 3
```

delete

Delete entire array.

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

myArray=("Alice" "Bob" "Eve")
unset myArray
echo ${myArray[@]}
#
```

delete at

Delete Nth item in array (N = index)

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

myArray=("Alice" "Bob" "Eve")
unset myArray[1]
echo ${myArray[@]} # Alice Eve
```

filter

Filter elements of an array based on given pattern.

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

myArray=('Alice' '22' 'Bob' '16' 'Eve')
filtered=(`for i in ${myArray[@]} ; do echo $i; done | grep [0-9]`)
echo ${filtered[@]} # 22 16
```

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

myArray=("Alice" "Bob" "Eve")
echo ${myArray[@]/e/} # Alice Eve
```

iterate | for each

Iterate over array items.

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

myArray=("Alice" "Bob" "Eve")

for item in ${myArray[@]}; do
  echo "$item"
done

# Alice
# Bob
# Eve
```

length

Returns length of array.

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

myArray=("Alice" "Bob" "Eve")
echo ${#myArray[@]} # 3
```

range

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

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

myArray=("Alice" "Bob" "Eve" "Shellman" "Remisa")
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.

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

myArray=("Alice" "Bob" "Eve")
echo ${myArray[@]//e/9} # Alic9 Bob Ev9
```

set elements

Set element given value as Nth element.

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

myArray=("Alice" "Bob" "Eve")
myArray[1]="Shellman"
echo ${myArray[@]} # Alice Shellman Eve
```

slice

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

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

myArray=("Alice" "Bob" "Eve" "Shellman")
echo ${myArray[@]:1:2} # Bob Eve
```

directory

Contains `String` related operations.

create

Creates a directory.

```
mkdir "test dir"
```

Creates `test` directory at the current path.

create nested

Create directories as required.

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

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:

```
── test
|   ├── a
|   |   ├── 1
|   |   ├── 2
|   .   .   .
|   .   .   .
|   |   └── 100
|   ├── b
|   |   ├── 1
|   |   ├── 2
|   .   .   .
|   .   .   .
|   |   └── 100
|   ├── c
|   .   .
|   .   .
```

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.

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

result=`find . -maxdepth 3 -type f -name "*.txt"`
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.

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

function myFunction () {
  echo "$1"
  echo "$2"
}

myFunction "some argument" "another argument"
# some argument
# another argument
```

args

Access to function arguments.

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

function myFunction () {
  echo "$@"
}

myFunction "some argument" "another argument"
# some argument another argument
```

args count

Number of function arguments.

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

function myFunction () {
  echo $#
}

myFunction "some argument" "another argument"
# 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.

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

function test () {
  echo "$1"
  return 25
}

test "return value"
echo "$?"
# return value
# 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.

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

response=`curl -s http://example.com`
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.

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

ls # this command will succeed

if [[ $? == 0 ]]; then
  echo "command succeeded"
else
  echo "command failed"
fi
# command succeeded
```

failure check

Check if last command has failed.

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

touch /file.txt # this command will fail without sudo

if [[ $? == 0 ]]; then
  echo "command succeeded"
else
  echo "command failed"
fi
# command failed
```

nice

Run a command with modified scheduling priority. Niceness values range from `-20` (highest priority) to `19` (lowest priority) and default value is `0`.

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

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`.

renice

Change a running process priority. Niceness values range from `-20` (highest priority) to `19` (lowest priority) and default value is `0`.

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

sudo renice -n -5 -p `pgrep dockerd`
```

In above example we are changing priority of `dockerd` process (docker daemon on a system where docker is installed) to higher than normal.

if cmd exists

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

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

if [ `command -v docker` ]; then
  echo "docker is installed"
else
  echo "docker is NOT installed"
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`)

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

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.

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

tar -C ~/ -xzvf ~/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.

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

base64Encoded=`echo -n "$variableToEncode" | base64`
```

base64 decode

Decode `String` from *base64* into `Binary`.

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

base64Decoded=`echo -n "$variableToDecode" | base64 -d`
```

hash

Hash variable content with desired algorithm.

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

hash=`echo -n "$variableToHash" | md5sum | cut -f1 -d ' '`
echo "$hash"
```

Supported algorithms:

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

http

Contains `HTTP` related operations.

GET

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

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

curl --request GET -sL \
  --user-agent 'Shellman' \
  --url 'http://example.com'
```

Above example sends a *HTTP GET* request to http://example.com with desire User Agent[8].

DELETE

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

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

curl --request DELETE -sL \
  --user-agent 'Shellman' \
  --url 'http://example.com'
```

POST

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

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

curl --request POST -sL \
  --user-agent 'Shellman' \
  --url 'http://example.com' \
  --data 'key1=value1' \
  --data 'key2=value2'
```

POST file

Send file with *http POST*.

[8] https://en.wikipedia.org/wiki/User_agent

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

curl --request POST -sL \
  --user-agent 'Shellman' \
  --url 'http://example.com' \
  --form 'key=value' \
  --form 'file=@~/image.jpg'
```

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

header

Send http request with custom header(s).

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

curl --request GET -sL \
  --user-agent 'Shellman' \
  --header 'key: value' \
  --url 'http://example.com'
```

cookie

Send http request with desired cookies.

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

curl --request GET -sL \
  --user-agent 'Shellman' \
  --cookie 'key=value' \
  --url 'http://example.com'
```

download

Download from url and save to desired *path*.

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

curl --request GET -sL \
  --user-agent 'Shellman' \
  --output '~/downloaded-file.zip' \
  --url 'http://example.com/file.zip'
```

ftp

Contains `FTP` related operations.

list

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

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

curl ftp://remisa:1234@mydomain/backup/
```

download

Download specified file from ftp server.

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

curl ftp://remisa:1234@mydomain/backup/latest.zip
```

upload

Upload specified file to ftp server at desired path.

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

curl -T test.zip ftp://remisa:1234@mydomain/backup/
```

delete file

Delete specified file from ftp server.

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

curl ftp://remisa:1234@mydomain/backup/test.zip -Q "DELE test.zip"
```

rename

Rename specified file/directory on ftp server.

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

curl ftp://remisa:1234@mydomain/backup/ -Q "-RNFR backup/test.zip" -Q "-RNTO backup/\
renamed.zip"
```

file

Contains `File` related operations.

file delete | file remove

Delete given file.

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

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.

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

result=`find ~ -maxdepth 3 -type f -name "*.txt"`
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.

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

result=`find ~ -maxdepth 1 -type f -exec grep "ls" {} +`
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.

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

cat ~/test.txt | while read line; do
  echo "$line"
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.

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

lines=`docker images`
echo "sample header" > ~/test.txt
for line in ${lines}; do
  echo "$line" >> ~/test.txt
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.

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

cat >~/test.txt <<EOL
Header

first line
second line
EOL
```

file write multiline sudo

Write multiple lines into a file which needs root permission.

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

cat << EOL | sudo tee /test.txt
Header

first line
second line
EOL
```

remove files older than

Remove files older than x days.

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

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:

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

echo `tput setaf 1`hello world`tput sgr0`
```

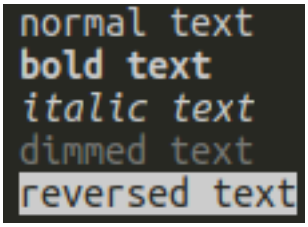
Colorful Text

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.



formated text

bold

Write in **bold**.

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

echo `tput bold`bold text`tput sgr0`
```

italic

Write in *italic*.

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

echo `tput sitm`italic text`tput sgr0`
```

dim

Write dim text.

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

echo `tput dim`dimmed text`tput sgr0`
```

reverse

Write text in reverse contrast.

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

echo `tput rev`reversed text`tput sgr0`
```

process

Contains `Process` related information and operations.

list

List all system processes.

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

ps -A
#   PID TTY          TIME       CMD
#     1   ?        00:00:03 systemd
#     2   ?        00:00:00 kthreadd
#     3   ?        00:00:01 ksoftirqd/0
#     5   ?        00:00:00 kworker/0:0H
#     7   ?        00:01:46 rcu_sched
# ...
```

ID

Get process ID by its name. Many Linux commands need *process id* (PID).

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

firefoxPID=`pgrep firefox`
echo $firefoxPID
```

Kill

Kill a process by its name. `kill` command needs a *PID* (process ID) which we can find by `pgrep` command via command substitution.

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

sudo kill -9 `pgrep firefox`
```

In above example we find *firefox* `PID` and pass it to `kill` command. Here `-9` is a switch of `kill` command (kill signal). You can see a list of all signals by typing `kill -l` in terminal.

system

Contains `System` related information and operations.

uptime

System uptime (hh:mm:ss).

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

sys_uptime=`uptime | cut -d ' ' -f2`
echo "$sys_uptime" # 03:26:47
```

memory info

System memory information in kilobytes (KB).

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

sysMemoryMemTotal=`cat /proc/meminfo | grep 'MemTotal' | awk '{print $2}' | head -n \
1`
echo "$sysMemoryMemTotal" # total system memory in KB
```

distro name

Operating System ID (i.e. Ubuntu).

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

distroName=`lsb_release -i | awk '{print $3}'`
echo "$distroName"
```

distro version

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

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

distroName=`lsb_release -r | awk '{print $2}'`
echo "$distroName"
```

distro codename

Operating System codename (i.e. xenial).

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

distroName=`lsb_release -c | awk '{print $2}'`
echo "$distroName"
```

kernel name

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

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

kernelName=`uname -s`
echo "$kernelName" # Linux
```

kernel release

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

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

kernelRelease=`uname -r`
echo "$kernelRelease" # 4.4.0-140-generic
```

processor type

Operating System processor type (i.e. x86_64).

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

processorType=`uname -p`
echo "$processorType" # x86_64
```

processor count

Number of processors (cores).

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

processorCount=`lscpu | grep 'CPU(s)' |awk '{print $2}' | head -n 1`
echo "$processorCount" # 4
```

processor architecture

Processor architecture (i.e. x86_64).

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

processorArchitecture=`lscpu | grep 'Architecture' |awk '{print $2}' | head -n 1`
echo "$processorArchitecture" # x86_64
```

processor model

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

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

processorModel=`lscpu | grep 'Model name' |cut -d ' ' -f 3- | sed -e 's/^[[:space:]]\
*//'`
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.

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

git clone https://github.com/user/repository.git
cd repository
```

clone branch

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

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

git clone -b develop https://github.com/user/repository.git
cd repository
```

config list

List git configurations.

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

git config --list
```

config set

Set a *git* configuration.

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

git config --global user.name "Remisa"
```

commit

Commit changes.

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

git commit -m "fixed typo"
```

commit search

Search for a commit which contains searchCriteria.

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

git log --all --grep='typo'
```

commit undo

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

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

git reset --soft HEAD~1 # undo last local change but don't delete them
```

commit list notPushed

List non pushed commits.

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

git log origin/master..HEAD
```

branch create

Create a local branch and switch into it.

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

git checkout -b develop
```

branch list

List all branches.

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

git branch
```

branch push

Push branch to remote.

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

git push origin develop
```

branch rename

Rename current branch.

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

git branch -m newName
```

branch delete local

Delete local branch.

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

git branch --delete localBranch
```

branch delete remote

Delete remote branch.

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

git push origin --delete remoteBranch
```

changes revert

Revert tracked changes.

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

git checkout .
```

patch create

Create a patch from changes.

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

git diff > patch1.patch
```

patch apply

Apply a patch from file.

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

git apply < patch1.patch
```

remote list

List all remotes.

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

git remote
```

remote urlAdd

Add remote url.

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

git remote add origin https://github.com/user/repository.git
```

remote urlChange

Change remote url.

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

git remote set-url origin https://github.com/user/repository.git
```

tag list

List all tags.

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

git tag
```

tag commit

Tag a commit.

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

git tag -a release/1.0.0 -m "1.0.0 release"
```

tag remote delete

Delete tag from remote.

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

git push --delete origin tagName && git push origin :tagName
```

tag remote push

Push tag to remote.

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

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`:

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

var=2

case "$var" in
  1)
    echo "case 1"
  ;;
  2|3)
    echo "case 2 or 3"
  ;;
  *)
    echo "default action"
  ;;
esac
# 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.

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

let a=2+3
echo $a # 5
let "a = 2 + 3"
echo $a # 5
let a++ # increase a
echo $a # 6
let "a = 2 * 3"
echo $a # 6
```

region

Create a region to separate different parts of *script*.

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

# >>>>>>>>>>>>>>>>>>>>>>> variables >>>>>>>>>>>>>>>>>>>>>>>
var=1
# <<<<<<<<<<<<<<<<<<<<<<< variables <<<<<<<<<<<<<<<<<<<<<<<
```

expr

It is and old command for doing *arithmetic operations*. Use `$(( ))` instead.

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

result=`expr 2 \* 3`
echo "$result" # 6
```

Equivalent to:

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

result=$((2 * 3))
echo "$result" # 6
```

ask question

Ask a question from user and receive its answer from input. It is possible to provide a default answer to the question.

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

read -ep "What is your name? " -i Remisa ANSWER
echo "$ANSWER" # print user's answer
```

timeout

Run a command within a time frame.

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

timeout 5 curl -s http://example.com
echo "at most 5 seconds later"
```

ips

Array of local IPs.

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

IPS=`hostname -I`
echo "$IPS"
```

ip info

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

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

echo `curl -s ipinfo.io/country`
# U.K
```

ip public

Find public ip address via different services.

- bot.whatismyipaddress.com
- ident.me
- ipecho.net/plain
- icanhazip.com
- ifconfig.me
- api.ipify.org
- ipinfo.io/ip

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

PUBLIC_IP=`curl -s api.ipify.org`
echo "$PUBLIC_IP"
```

service manager

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

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

sudo systemctl restart service
```

sleep

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

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

sleep 2m
# halts script for 2 minutes
```

stopwatch

Use *stopwatch* to calculate script running time.

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

# beginning of script
STOPWATCH_START_TIME=$(date +%s)

# script
sleep 30s


# end of script
STOPWATCH_END_TIME=$(date +%s)

# print elapsed time
```

```
STOPWATCH_ELAPSED_TOTAL_SECONDS=$((STOPWATCH_END_TIME - STOPWATCH_START_TIME))
STOPWATCH_ELAPSED_MINUTES=$((STOPWATCH_ELAPSED_TOTAL_SECONDS / 60))
STOPWATCH_ELAPSED_SECONDS=$((STOPWATCH_ELAPSED_TOTAL_SECONDS % 60))
echo elapsed $STOPWATCH_ELAPSED_MINUTES minutes and $STOPWATCH_ELAPSED_SECONDS secon\
ds
```

lib

Contains a set ol 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:

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

function sum () {
  local result=0
  for item in $@; do
    ((result += item))
  done
  echo $result
}

var1=2; var2=5; var3=4
result=`sum $var1 $var2 $var3`
echo $result
```

math product

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

Example usage:

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

function product () {
  local result=1
  for item in $@; do
    ((result *= item))
  done
  echo $result
}

var1=2; var2=3; var3=5
result=`product $var1 $var2 $var3`
echo $result
```

math average

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

Example usage:

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

function average () {
  local result=0
  for item in $@; do
    ((result += item))
  done
  echo $((result / $#))
}

var1=2; var2=3; var3=4
result=`average $var1 $var2 $var3`
echo $result
```

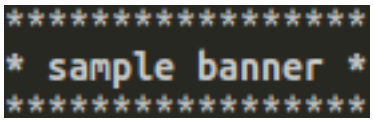
Misc

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`:

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

function banner_simple() {
  # function body...
}

# call function
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`:

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

function banner_color() {
  # function body...
}

# call function
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.

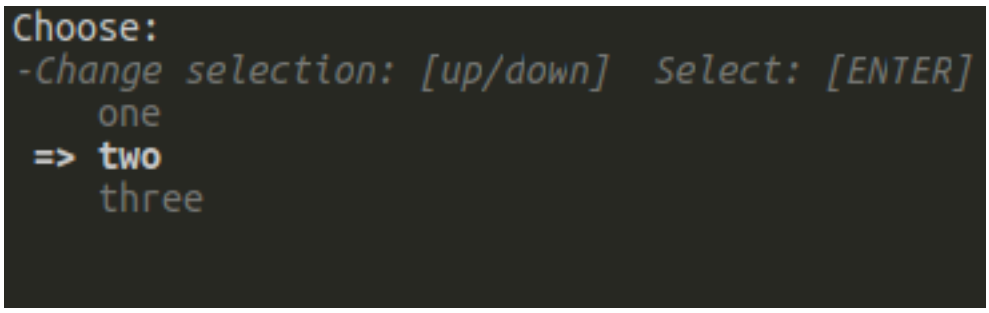
```
#!/usr/bin/env bash

function import() {
  # function body...
}

import "mylib"

# Call some function from mylib.sh
```

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

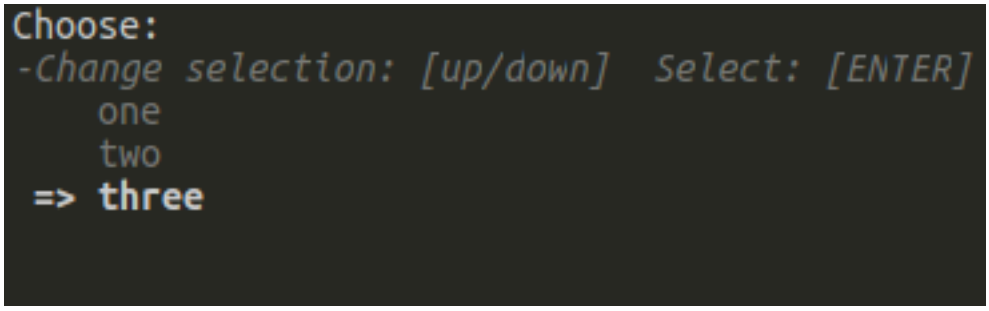
options

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...

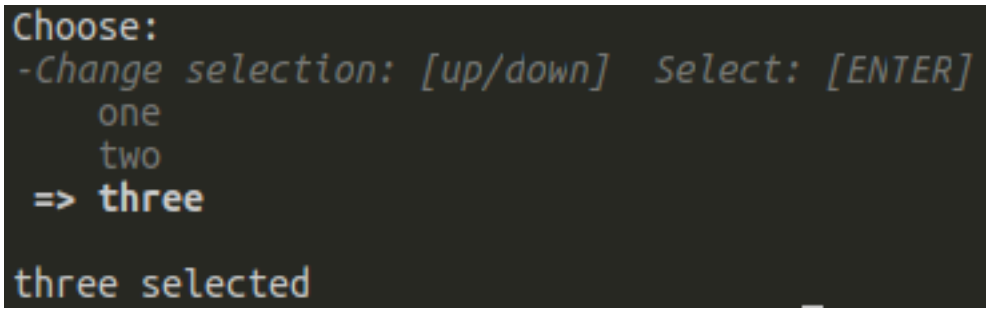
```
#!/usr/bin/env bash

function chooseOption() {
  # function body...
}

options=("one" "two" "three") # array of options
chooseOption "Choose:" 1 "${options[@]}"; choice=$? # call function
echo "${options[$choice]}" selected # print selected item by user
```



options 1



options 2

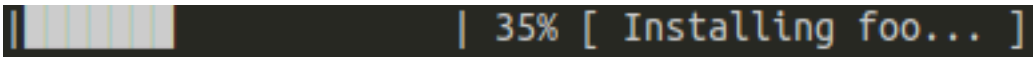


choice

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..."
```



progress

scan

Scan a host port range (tcp/udp).

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

function scan () {
  # function body...
}

# scan tcp ports 5000-10000 of localhost
scan tcp localhost 5000 10000

# tcp 8081 => open
# tcp 9000 => open
```

Advanced

piping

To do...

shift command

Argument parsing

Solutions

Argument Parsing

Contents of `greet.sh`:

Argument Parsing

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

greeting="good night"
name="everyone"

# >>>>>>>>>>>>>>>>>>>>>>>> Shellman snippet >>>>>>>>>>>>>>>>>>>>>>>>

POSITIONAL=()
while [[ $# > 0 ]]; do
  case "$1" in
    -m|--morning)
    greeting="good morning"
    shift # shift once since flags have no values
    ;;
    -n|--name)
    name="$2"
    shift 2 # shift twice to bypass switch and its value
    ;;
    *) # unknown flag/switch
    POSITIONAL+=("$1")
    shift
    ;;
  esac
done

set -- "${POSITIONAL[@]}" # restore positional params

# <<<<<<<<<<<<<<<<<<<<<<<< Shellman snippet <<<<<<<<<<<<<<<<<<<<<<<<

echo "$greeting $name"
```

Nested Directories

Contents of `nested-directories.sh`:

Nested Directories

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

mkdir -p test/{a..z}/{1..100}
```

Colorful Text

Contents of `colorful-text.sh`:

Nested Directories

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

for((i=0;i<=7;i++)); do
  echo `tput setaf $i`Hello World!`tput sgr0`
done
```

Output:

colorful text

Factorial

Contents of `factorial.sh`:

Factorial

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

function fact () {
  result=1
  for((i=2;i<=$1;i++)); do
    result=$((result * i))
  done
  echo $result
}

# example: 4! = 4 * 3 * 2 = 24
fact 4
```