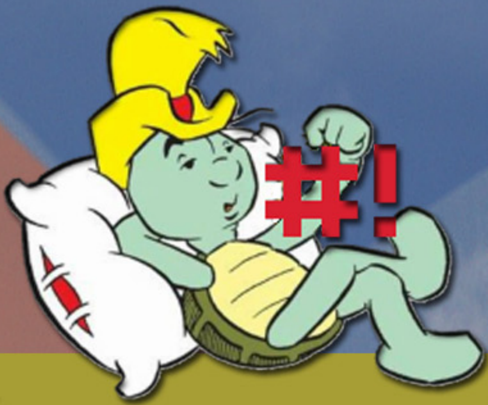
Shellman



#!BASH SCRIPTING

Remisa Yousefvand

Shellman Bash Scripting

Remisa Yousefvand

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

This version was published on 2020-07-11

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

Shellman[1] is a vscode[2] snippet extension and is made to provide a high level interface for writing shell scripts. That means easy way to accomplish the job without worrying about the details. As long as you understand the interface and how it is organized, you are good. The trade-off is you miss the details and it is also a good thing. After all that's the purpose of abstraction. When you need more control you can dig dipper and deal with details but before that stay at high ground as much as you can.

When I started shell scripting, even with the best tools available I found it unorganized. I couldn't find easily how to do file, string, array... related operations. I didn't care how a string is reversed in shell scripting as long as it works (I know about best practices, performance, compatibility... but they are not my primary concerns in a new field). Give some code the string `abc` which transforms it to `cba`. If you are coming from a `OOP` background you expect[3] such a function in `String` class. So `Shellman` organizes such operations under related abstract groups called `namespaces` and I just searched internet to find most fitting codes that do the job and organized them.

The hard part of *shell scripting* is not *shell scripting* itself, it is understanding Linux and 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[4].

Remisa Yousefvand

July 2020

[1] https://marketplace.visualstudio.com/items?itemName=Remisa.shellman

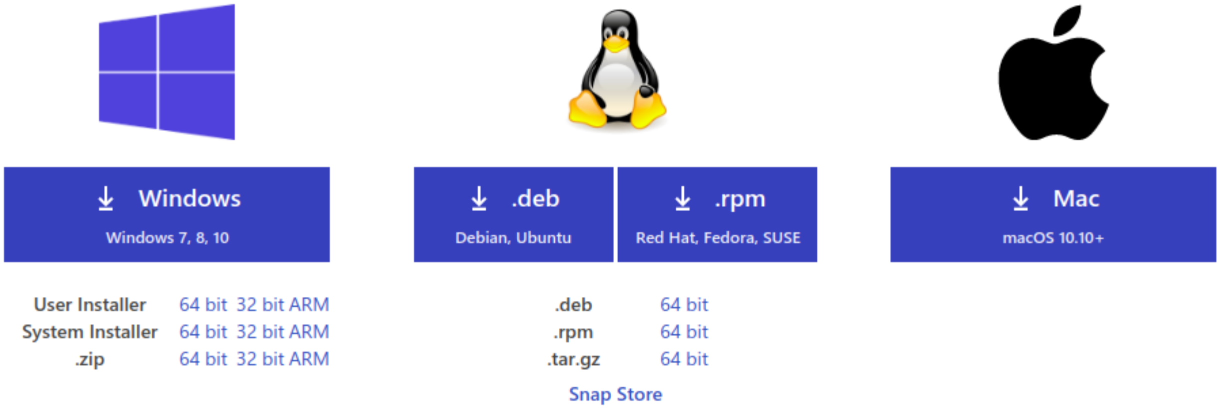
[2] https://code.visualstudio.com

[3] From cognitive/statistical point of view, coming from `OOP`, or at least expecting order, you find `Shellman` convenient because its structure matches your beliefs (prior). The probability distribution curve has the same shape, so you learn fast (little update to your curve is needed). Your wishes about where to find a function just comes true.

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

Prerequisites

- vscode[5] IDE

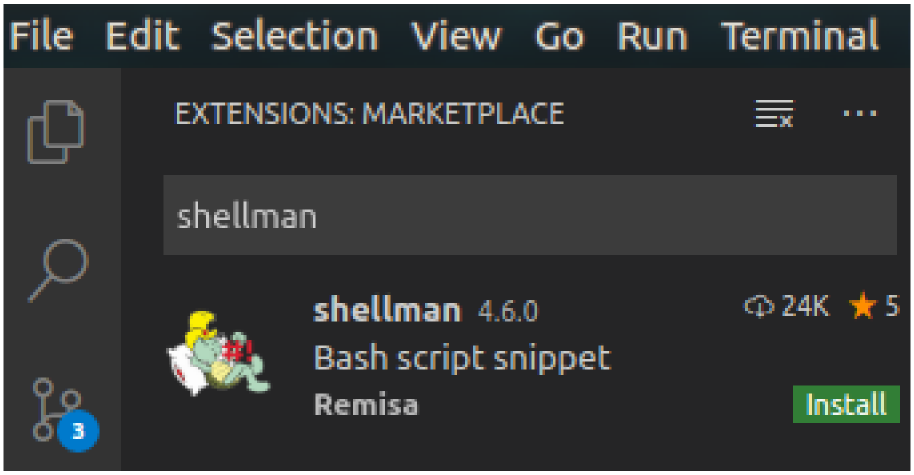


User Installer	64 bit	32 bit	ARM
System Installer	64 bit	32 bit	ARM
.zip	64 bit	32 bit	ARM

.deb	64 bit
.rpm	64 bit
.tar.gz	64 bit

vscode download

- Shellman[6] snippet



shellman

shellman 4.6.0
Bash script snippet
Remisa

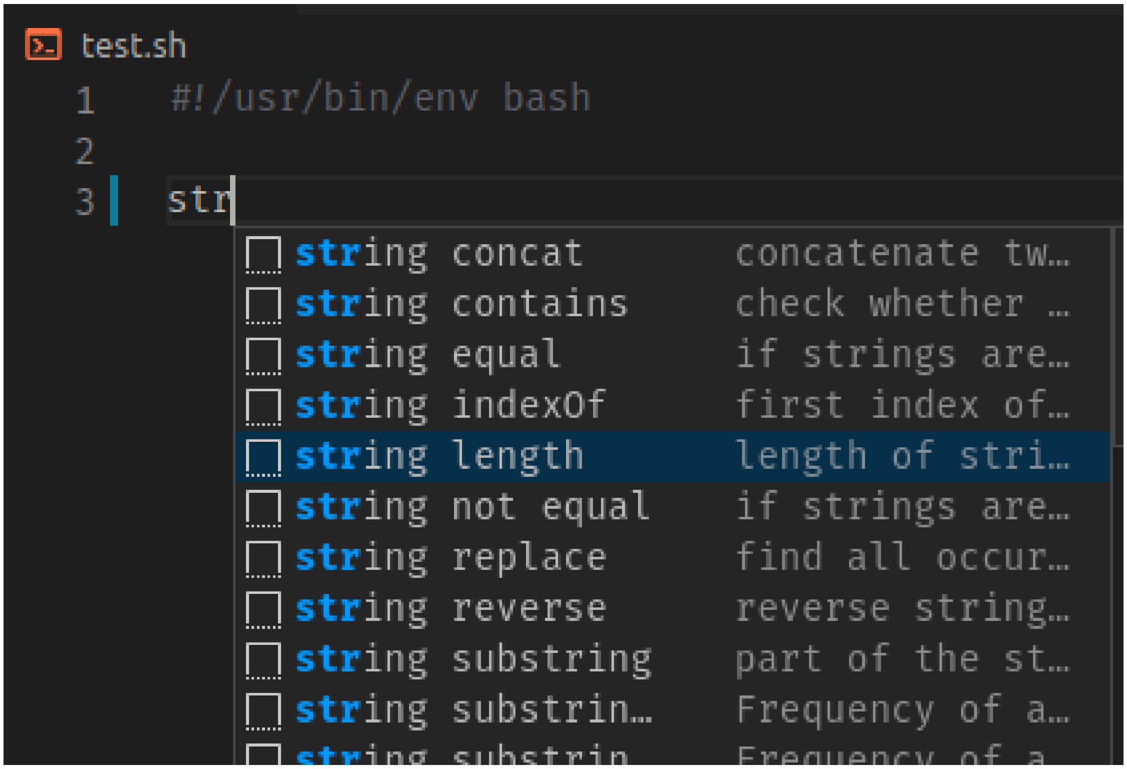
shellman install

[5] https://code.visualstudio.com

[6] 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.



test.sh
string concat	concatenate tw…
string contains	check whether …
string equal	if strings are…
string indexOf	first index of…
string length	length of stri…
string not equal	if strings are…
string replace	find all occur…
string reverse	reverse string…
string substring	part of the st…
string substrin…	Frequency of a…
string substrin	Frequency of a

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

str
```

String Namespace

When you press `ENTER` on an item like above picture, vscode inserts some code into your script which you can move into different parts using `TAB` key. This is called `snippet`. To access a snippet you start typing and vscode shows a menu of snippets with matching prefixes.

Snippet Alias

When a snippet can be activated by two or more prefixes, a `|` between prefixes is used to indicate that.

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, comments start with `#`. 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
```

This is available via `shebang` | `bash` snippet.

If a shell script doesn't contain `shebang` then whoever gonna execute such an script needs to specify the script engine manually and pass the script as an argument to it:

```
bash test.sh
```

Run a Bash Script

Bash script files by convention has **.sh** *file extension*[7]. 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 in your script as you do in terminal:

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

rm some-file
```

If the command needs **root**[8] 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`.

[7] 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*)

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

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

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

Above script is the same as:

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

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

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

```bash
#!/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 `$` to the variable name (also you can write `${variable}`).

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

```bash
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
# or
echo ${firstName}
```

Variable Access Rule

To access a variable value prefix it with `$`

Variables are case sensitive (like Linux filesystem):

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

var=1
Var=2

echo "$var" # 1
echo "$Var" # 2
```

As you may guessed in assignment rule, *space* has a special meaning in *shell scripting*. With *space* over = shell assumes variable is a command and = and variable value are parameters to that command.

```
firstName = Remisa
# firstName: command not found
```

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"
```

Consider you want to delete a file named `some file.txt` and you have save its name in a variable like:

```
fileName="some file.txt"
rm $fileName
```

With above script instead of deleting `some file.txt` you are telling `rm` to delete two files named `some` and `file.txt` and you will get an error (No such file or directory).

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 you need adding more characters between variables then use `"${variable}"` syntax (this syntax is recommended by many sources as the default syntax):

```
a="abc"
b="def"
c="ghi"
echo "${a}a ${b}b ${c}c"
# abca defb ghic
```

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

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

: "${variable:=default}"
```

In above example `variable` is set only if it is *empty*. We will use this snippet later after argument parsing to assign default values to variables when they are not passed (optional parameters) to script.

Variable Types

Bash supports **String**, **Integer** and **Array**. Most of the time you only need **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. Although you can define variables using declare keyword, in this book we define variables literally.

```
# Number or Sting:
var1=1234
var2=12.56
var3="some text" # use double quote when there is a space in string

# Array:
myArray=("one" "two" "three")
# or
myArray2=(
  "four"
  "five"
  "six"
)

echo "$var1"          # 1234
echo "$var2"          # 12.56
echo "$var3"          # some text
echo "${myArray[@]}"  # one two three
echo "${myArray2[@]}" # four five six
```

Function

Function in shell script is not what you expect from a function in other languages. They are like commands defined in your script just like echo and ls. To define a function named myfunc simply (there is a func snippet for that):

```
#!/usr/bin/env bash
function myfunc () {
  echo "$1"
}
```

Function definition should precedes its usage. function keyword is optional and can be omitted:

```
#!/usr/bin/env bash
myfunc () {
  echo "$1"
}
```

To access function arguments we use `$1`, `$2`, `$3`... or access all of them at once through an array:

```
#!/usr/bin/env bash
function myfunc () {
  arguments=("$@")
  # arguments is the array variable containing all function parameters
}
```

If you need to return some value from a function use `echo`. There is a `return` keyword in bash but you cannot use it for returning values from functions most of the time (unless your function return an integer between `0` and `255`) also it has its own meaning (`0` for success and `1`-`255` for error codes). If you want to terminate a function execution at some point use `return` (for example inside an `if` statement).

```
#!/usr/bin/env bash
function myfunc () {
  echo "this is the result"
  # we don't need "return" here because function already reaches its end
}
```

On the caller side we capture this result with command substitution.

```
function myfunc () {
  echo "this is the result"
}

result=`myfunc`
echo "$result"
```

For more function related operations see function snippets.

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 most references method two is recommended because it is the only one that works with nested command substitutions. For the sake of brevity and consistency, we will use method one (backtick) in this book unless we need nested command substitutions (Also to be able to read and understand shell scripts written by others).

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

```
output=`ls` # store ls results in a variable named output
# same as: output=$(ls)

echo "$output" # print output value (ls result)
```

There is a more advance technique for using a command output as another command input, namely **piping (|)**, which is beyond the scope of this book (if you have `functional programming` background you are already familiar with the idea).

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[9]. If the return value is *zero*, in means no error happened and other values indicates command **failure** (1-255).

Command success/failure

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

To check that, you can read *last command return value* via `$?`. 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:

[9] 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.

```
# following command will fail due to lack of permission
touch /not-enough-permission-to-create-file
```

`touch` command creates an empty file.

We are trying to create the empty file `not-enough-permission-to-create-file` at the root of your file system (`/`). Without **sudo** normally (unless user is `root`) 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:

```
echo "Hello World!"

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

Check *command exit code* **immediately** after that command or you may get wrong result:

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

touch /not-enough-permission-to-create-file

echo "checking operation..."

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

In above example your **`if`** statement won't print the `command failed` message since last command is `echo` and not `touch`.

Exit

It is a good practice to inform script caller (in case other scripts use yours) about success or failure of your script. To indicate success:

```
exit 0
```

And if an error happens use an exit code. Document exit codes at the top of of your script:

```
exit 5 # documented as "no internet connection"
```

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 (unless it is taken, like adding `version` to following list). 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`[10] directory. Here `-o` is a *switch* which takes one parameter (a path).

Flag vs Switch

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

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

[10] ~ is a shorthand for current user, *home directory*, which usually is `/home/username`. This path is also accessible via `$HOME` global variable.

```
POSITIONAL=()
while [[ $# > 0 ]]; do # while arguments count > 0
  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
```

The *while loop* keeps looping until there is no more arguments to process. Although the passed arguments to your script would not disappear themselves, we trim them from left using `shift` command. So if your script is executed like:

```
./greet.sh -m --name Remisa
```

Input arguments are `-m --name Remisa`. After a `shift` they become `--name Remisa` and so on. So if you need to process a switch with two arguments `shift 3`.

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 or store the value of switch parameter(s):

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

Repeat above procedure for more flags.

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

```
-o|--output)
outputPath=$2
```

In above example we are saving the switch value in `outputPath` for using later. We refer to first switch parameter with `$2` and the second with `$3` and so on because the `$1` refers to the switch itself. Then `shift` properly.

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 check solutions section.

As you have noticed, first argument can be accessed via `$1`, second argument via `$2`…

And yes, `$0` refers to script name itself at the time of execution.

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

Organizing your Bash Script

An organized script is easy to understand and maintain. Recommended structure of `script.sh` from top to bottom is:

1. shebang (`shebang` | `bash` snippet)
2. summary (`summary` snippet)
3. handler functions `region` (if any, see event namespace)
4. event handlers `region` (if any, see event namespace)
5. animation frames `region` (if any, see animation namespace)
6. functions `region`
7. argument parsing
8. setting default variable values
9. rest of code (minimize it to function calls)

Usually you only need *1, 2, 6, 9* from above list. *argument parsing* and *setting default variable values* can be done in reverse order. In that case create a `variables` region after summary and set default values. Later if argument parsing overrides some of your variables (passed as flag/switch) the rest of variables contain default values.

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

# Exit codes
# ==========
# 0   no error
# 1   script interrupted
# 2   error description
```

Event handling

If you need to run a set of specific tasks before your script exits or in case user terminates your script (pressing `CTRL + C`) you need to assign a `handler` function to appropriate event. The problem with event handlers is we use functions to run if a certain event happens so before assigning an event to a function we need to write the function. To capture events as soon as possible we need to assign event handlers early in our script. Thats why I have separated functions into two parts, event handlers, at the top of the script just before binding events to them and the rest of functions which are not needed so early. See event namespace for more information.

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. If function `B` calls function `A`, then function `A` definition should precede definition of function `B`.

```bash
#!/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:

```bash
var1="Hello World!"
echo "$var1"   # Hello World!
# OR
echo "${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" # "&$*
```

backtick is used for command substitution

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

Sample scripts

Apart from some examples in this book there is a samples directory[11] in project repository which contains the steps and reasoning behind writing some shell scripts using `Shellman`.

[11] https://github.com/yousefvand/shellman/tree/master/samples

Namespaces

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

There is a length limitation to namespaces in a snippet so some of them doesn't exist under exact namespace I write in this book. Fortunately they are few.

It happens when a single snippet is not enough to do the job and a function is needed. Such functions available in `Shellman`. See lib for more information.

loop

Contains `while`, `until` and `for`. Actually `for` doesn't have `loop` prefix so by typing `loop` you won't see it. That's because of readability limitations so if you need any kind of `for` type `for`.

while

`while` condition.

For arithmetic comparison use `(( ))`.

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

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

For string comparison use `[ ]`.

```bash
#!/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 `(( ))`.

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

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

For string comparison use `[ ]`.

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

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

for i

`for` loop.

```bash
#!/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.

```
#!/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:

```
#!/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.

```bash
#!/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`:

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

str1="bye"
str2="hello"

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

or `if/else`:

```bash
#!/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 int =

Check if two integers are equal.

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

int1=67
int2=67

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

if int !=

Check if two integers are not equal.

```bash
#!/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.

```bash
#!/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.

```
#!/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.

```
#!/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.

```
#!/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.

```
#!/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.

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

str="hello world!"

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

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* exists.

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

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

if file exists

If given *file* exists.

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

filepath="/path/to/file"

file if [ -f "$filepath" ]; then
  echo file exists
fi
```

if file =

Check if two files are equal.

```bash
#!/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 (file size > 0).

```bash
#!/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.

```bash
#!/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.

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

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

if file writable

Check if file is writable.

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

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

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.

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

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

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

func args

Access to function arguments.

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

function myFunction () {
  echo "$@"
}

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

func args count

Number of function arguments.

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

function myFunction () {
  echo $#
}

myFunction "some argument" "another argument"
# 2
```

func 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
```

string

Contains `String` related operations.

string concat

concatenates two strings

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

str1="a"
str2="b"
str="${str1}y${str2}z"
echo "$str" # aybz
```

string contains | if string contains

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

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

var="hello world!"

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

string equal | if string =

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
```

string not equal | if string !=

Checks if two strings are not equal.

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

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

string indexOf

Returns index of substring inside a string.

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

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

if string empty

Check if variable is an empty string.

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

var=""
if [ -z "$var" ]; then
  echo "Variable is empty string."
fi
# Variable is empty string.
```

if string not empty

Check if variable is not an empty string.

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

var="something"
if [ -n "$var" ]; then
  echo "Variable is not an empty string."
fi
# Variable is not an empty string.
```

string length

Returns *length* of given string.

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

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

string replace

Replace a substring with given string in another string.

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

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

string reverse

Reverse given string.

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

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

string 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`.

string substring count | string substring frequency

Finds the frequency of a substring in a string (may need character escaping).

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

frequency=`sed -E 's/(.)/\1\n/g' <<<"a!bc!def!" | grep -c "!"`
echo "${frequency}" # 3
```

string toLower

Returns lowercase of given string.

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

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

string toUpper

Returns uppercase of given string.

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

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

string 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
```

string 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
```

string 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
```

string 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
```

array

Contains `Array` related operations.

array declare

Declare a literal array.

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

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

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

# Alice
# Bob
# Eve
```

array add | array 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
```

array all

All items of array.

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

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

array at index

Returns item Nth from array (N = index).

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

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

array 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
```

array delete

Delete entire array.

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

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

array delete at

Delete Nth item in array (N = index)

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

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

array 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
```

array iterate | array forEach

Iterate over array items.

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

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

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

# Alice
# Bob
# Eve
```

array length

Returns length of array.

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

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

array 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
```

array slice | array 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)

array set element

Set element given value as Nth element.

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

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

cmd

Contains `command` execution related operations.

cmd

To run a command and use the returned value is named command substitution.

```bash
#!/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).

cmd success check

Check if last command has succeeded.

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

ls # this command will succeed

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

cmd failure check

Check if last command has failed.

```bash
#!/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
```

cmd 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`.

cmd 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

for p in $(pidof "chrome"); do sudo renice -n -5 -p "$p"; done
```

In above example we are changing priority of `chrome` process and its child processes 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.

math

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

math % (modulus)

Given two variables, returns reminder of dividing the first variable to the second.

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

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

math %= (modulus assign)

Given two variables, calculates reminder of dividing the first variable to the second and assigns the result to the first variable.

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

var1=13
var2=5
((var1 %= var2))
echo "$var1" # 3
```

math * (multiply)

Given two variables, returns product of them.

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

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

math *= (multiply assign)

Given two variables, calculates product of them and assigns the result to the first variable.

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

var1=2
var2=3
((var1 *= var2))
echo "$var1" # 6
```

Factorial

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

For the answer refer to Solutions section, factorial.

math + (add)

Given two variables, returns sum of them.

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

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

math ++ (increase)

Given a variables, adds one to it.

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

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

math += (add assign)

Given two variables, calculates sum of them and assigns the result to the first variable.

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

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

math - (subtract)

Given two variables, returns first minus second.

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

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

math -- (decrease)

Given a variable, subtracts one from it.

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

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

math -= (subtract assign)

Given two variables, calculates first variable minus the second and assigns the result to the first variable.

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

var1=19
var2=15
((var1 -= var2))
echo "$var1" # 4
```

math / (divide)

Given two variables, returns first divided by the second.

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

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

math /= (divide assign)

Given two variables, divides first variable by second and assigns the result to the first.

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

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

math 0.00 (precision)

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
```

math ^ (power)

Exponentiate *base* to the *power*.

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

echo $((2 ** 4)) # 16
echo $((3 ** 3)) # 27
```

math â ˆš (square root)

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

Calculate square root of 2 up to 7 decimal points.

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

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

math random

Generate random number between *min* and *max*

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

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

math constants

Some useful math constants.

- Ï€ = 3.14159265358979323846264338327950288
- e = 2.71828182845904523536028747135266249
- ð�›¾ = 0.57721566490153286060651209008240243
- Î© = 0.56714329040978387299996866221035554
- Ï• = 1.61803398874989484820458683436563811

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	Snippet
Black	0	`color black`
Red	1	`color red`
Green	2	`color green`
Yellow	3	`color yellow`
Blue	4	`color blue`
Magenta	5	`color magenta`
Cyan	6	`color cyan`
White	7	`color white`

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. You don't need to memorize codes, there is a snippet for every color. So for writing *hello world* in red, use `color red` snippet:

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

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

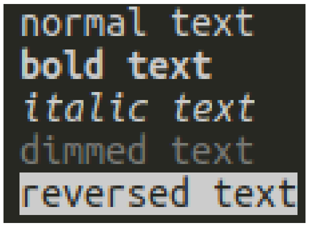
Colorful Text

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.



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

formated text

format bold

Write in **bold**.

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

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

format italic

Write in *italic*.

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

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

format dim

Write dim text.

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

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

format reverse

Write text in reverse contrast.

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

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

date

Contains `Date` related operations.

date now short

Short version of current system *date*.

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

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

date now UTC

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

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

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

date now year

Current year.

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

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

date now monthNumber

Current month number.

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

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

date 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
```

date now dayOfMonth

Current day of month.

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

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

date 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
```

date now dayOfYear

Current day of year (1-366).

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

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

time

Contains `Time` related operations.

time 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
```

time now UTC

Current UTC time.

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

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

time seconds epoch

Seconds from 01-01-1970 00:00.

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

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

file

Contains `File` related operations. For logical operations about `file` like if a given file is writable see logic section.

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 5 -type f -name "*.txt"`
echo "$result"
```

In above example all files (`-type f`) with `txt` extension in *home* (~) path up to `5` level of depth will be found. To search for directories use `-type d`.

file search | search in files | find in files

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.

```bash
#!/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 end of the file (previous contents remain there).

file write multiline

Write multiple lines into file.

```bash
#!/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.

```bash
#!/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.

directory

Contains `directory` related operations. For logical operations about `directory` like if a given directory exists see logic section.

directory create

Creates a directory.

```
mkdir "test dir"
```

Creates `test` directory at the current path.

directory 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.

directory delete nested | directory remove nested

Delete directory and all contents. **Use with caution**.

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

rm -rf /home/remisa/backups
```

In above example `backups` directory and all contents (files and directories) will be deleted from */home/remisa* path (home directory for user `remisa`).

directory find | file 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 d -name "backup*"`
echo "$result"
```

Above example finds all directories (`-type d`) up to `3` level depth (`-maxdepth 3`) in the current directory (`.`) where their names start with `backup` (`-name "backup*"`). To search for files use `-type f`.

event

Contains `event` related operations available via **event** namespace. There are two events supported by `Shellman`. `EXIT` and `CTRL+C`. Be careful about registering events multiple times. The last one you register takes control of what happens when event fires. If you have multiple things to do, move them all to a single function and register that function once.

event EXIT

If you need to run some commands before your script exits, you can put them in a function and call it everywhere your script may exits. But there is an easier way to do that. Register an `EXIT` handler function and it would be executed when your script execution is finished:

```
# Exit event handler
function on_exit() {
  tput cnorm # Show cursor. You need this if animation is used.
  # i.e. clean-up code here
  exit 0 # Exit gracefully.
}

# Put this line at the beginning of your script (after functions used by event handl\
ers).
# Register exit event handler.
trap on_exit EXIT
```

The `trap on_exit EXIT` part registers `on_exit` function to `EXIT` event. You need to register events as soon as possible in your script. But since it needs `on_exit` function, you need to define that function before registering the event.

The `tput cnorm` part ensures we have a visible cursor when script exits. If you are using animation feature of `Shellman` don't remove it. Anyway it is harmless and you can leave it there even if no animation is used.

event CTRL+C | event terminated

Available as `CTRL+C` | `terminated` under `event` namespace. If you need to do something in case your script gets interrupted (like when user presses `CTRL` and `C` keys on keyboard) you can register a handler function for it:

```
# CTRL+C event handler
function on_ctrl_c() {
  echo # Set cursor to the next line of '^C'
  tput cnorm # show cursor. You need this if animation is used.
  # i.e. clean-up code here
  exit 1 # Don't remove. Use a number (1-255) for error code.
}

# Put this line at the beginning of your script (after functions used by event handl\
ers).
# Register CTRL+C event handler
trap on_ctrl_c SIGINT
```

The `trap on_ctrl_c SIGINT` part registers `on_ctrl_c` function to `SIGINT` event. You need to register events as soon as possible in your script. But since it needs `on_ctrl_c` function, you need to define that function before registering the event.

The `tput cnorm` part ensures we have a visible cursor when script exits. If you are using animation feature of `Shellman` don't remove it. Anyway it is harmless and you can leave it there even if no animation is used.

archive

Contains `archive` related operations like compressing and decompressing files/directories. In Linux, `tar` combines files/folders into a single file without compression and mixing it with some compression utilities gives us for example `archive.tar.gz`. Looking at this file's extension the `tar` part tells us this file is a collection of other files/folders and `gz` part tells us this collection is compressed using `gzip`.

archive 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* directory (denoted by ~) into `archive.tar.gz` file in our *home* directory. This is useful for example if we are interested to backup `some-directory`.

archive decompress tar.gz

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

```bash
#!/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.

archive compress tar.xz

If you need more compression than previous method, use `tar.xz`:

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

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

tar -cJf ~/archive.tar.xz ~/some-directory
```

In above example we are compressing and archiving a directory (`some-directory`) from our *home* directory (denoted by ~) into `archive.tar.xz` file in our *home* directory. This file usually is smaller than its equivalent `archive.tar.gz` and the compression process is slower.

archive decompress tar.xz

Decompress an archive file (`.tar.xz`) into a path.

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

tar -C ~/Documents -xf ~/archive.tar.xz
```

In above example we are decompressing `archive.tar.xz` file from our *home* directory into `Documents` directory inside our *home* directory.

http

Contains `HTTP` related operations.

http GET

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

```bash
#!/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[12].

http DELETE

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

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

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

http POST

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

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

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

http POST file

Send file with *http POST*.

[12] 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.

http 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'
```

http 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'
```

http 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.

ftp list

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

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

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

ftp download

Download specified file from ftp server.

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

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

ftp upload

Upload specified file to ftp server at desired path.

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

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

ftp delete file

Delete specified file from ftp server.

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

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

ftp 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"
```

ip

Contains `ip` related operations.

ip local 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

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

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

crypto

Contains `Cryptography` related operations like encryption, decryption and hashing.

crypto 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.

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

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

crypto base64 decode

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

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

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

crypto hash

Hash variable content with desired algorithm.

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

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

Supported algorithms:

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

process

Contains `Process` related information and operations.

process 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
# ...
```

process ID

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

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

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

process 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.

system uptime

System uptime (hh:mm:ss).

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

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

system memory info

System memory information in kilobytes (KB). Available memory information:

- MemTotal
- MemFree
- MemAvailable
- Cached
- Buffers
- Active
- Inactive
- SwapTotal
- SwapFree
- SwapCached

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

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

system distro name

Operating System ID (i.e. Ubuntu).

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

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

system distro version

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

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

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

system distro codename

Operating System codename (i.e. xenial).

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

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

system kernel name

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

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

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

system kernel release

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

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

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

system processor type

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

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

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

system processor count

Number of processors (cores).

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

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

system 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
```

system 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
```

system service manage

Manage service (daemon) operations.

- enable
- disable
- start
- stop
- reload
- restart
- status

```
sudo systemctl status network-manager
```

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`
- Archlinux
  - `sudo pacman -S git`
- Mac
  - `brew install git`
- Windows
  - Download from https://gitforwindows.org/

git clone

Clone a repository to local machine.

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

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

git 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
```

git config list

List git configurations.

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

git config --list
```

git config set

Set a *git* configuration.

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

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

git commit

Commit changes.

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

git commit -m "fixed typo"
```

git commit search

Search for a commit which contains searchCriteria.

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

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

git commit undo

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

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

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

git commit list notPushed

List non pushed commits.

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

git log origin/master..HEAD
```

git branch create

Create a local branch and switch into it.

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

git checkout -b develop
```

git branch list

List all branches.

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

git branch
```

git branch push

Push branch to remote.

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

git push origin develop
```

git branch rename

Rename current branch.

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

git branch -m newName
```

git branch delete local

Delete local branch.

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

git branch --delete localBranch
```

git branch delete remote

Delete remote branch.

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

git push origin --delete remoteBranch
```

git changes revert

Revert tracked changes.

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

git checkout .
```

git patch create

Create a patch from changes.

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

git diff > patch1.patch
```

git patch apply

Apply a patch from file.

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

git apply < patch1.patch
```

git remote list

List all remotes.

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

git remote
```

git remote urlAdd

Add remote url.

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

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

git remote urlChange

Change remote url.

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

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

git tag list

List all tags.

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

git tag
```

git tag commit

Tag a commit.

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

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

git tag remote delete

Delete tag from remote.

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

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

git tag remote push

Push tag to remote.

```bash
#!/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`.

region

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

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

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

summary

Creates a commented summary for shell script. Use it at the top of your script.

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

# Title:         title
# Description:   description
# Author:        author <email>
# Date:          yyyy-mm-dd
# Version:       1.0.0

# Exit codes
# ==========
# 0   no error
# 1   script interrupted
# 2   error description
```

Document your script error codes under `Exit codes` section. These are code you have used in script when it exits due to an error (i.e. `exit 5` for lack of permission to do the job).

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
```

assign if empty | variable default value

Assigns a value to a variable if and only if the variable is empty. Useful for assigning default values.

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

var=""
: "${var:=default}"
echo "$var" # default

var="something"
: "${var:=default}"
echo "$var" # something
```

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"
```

service manager

Commands related to *services*. A *service* is a program which runs in background and doesn't need any user to login to be started (i.e. `ssh`).

```
#!/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. There are three snippets related to `stopwatch`, use at the given order:

1. `stopwatch start`: Starts stopwatch.
2. `stopwatch stop`: Stops stopwatch.
3. `stopwatch elapsed`: Calculates total 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 of library functions 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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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
```

Animation

There are three related snippets in `animation` namespace:

- `animation frame`: Use to define a frame of your animation.
- `fn animation animate`: A function used to animate frames.
- `fx animation animate`: Calling `animate` function.

See animation for more information.

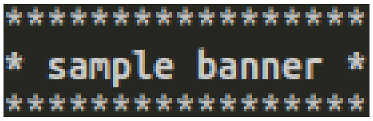
Misc

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`:

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

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

# call function
banner_simple "sample banner"
```



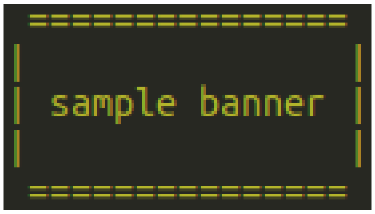
```
*****************
* sample banner *
*****************
```

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`:

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

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

# call function
banner_color yellow "sample banner"
```



```
 ===============
|               |
| 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.

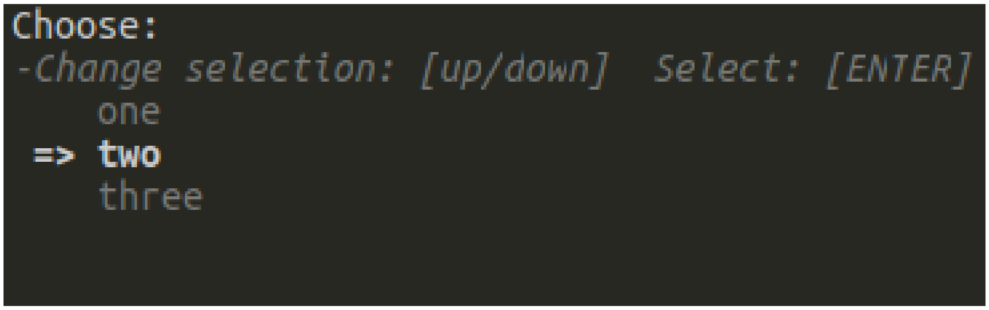
```
#!/usr/bin/env bash

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

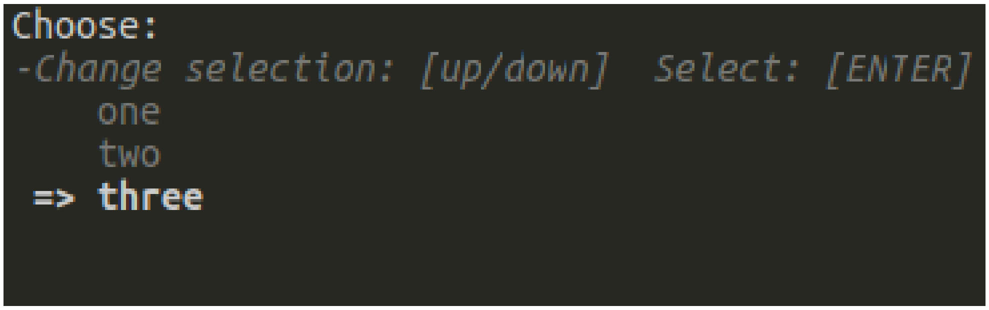
A function to print multi choice questions. To define the function use `fn options` at the top of script so later it can be called via `fx options` with question and choices. Default choice is zero based so `0` means first option, `1` means second...

```
#!/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
```



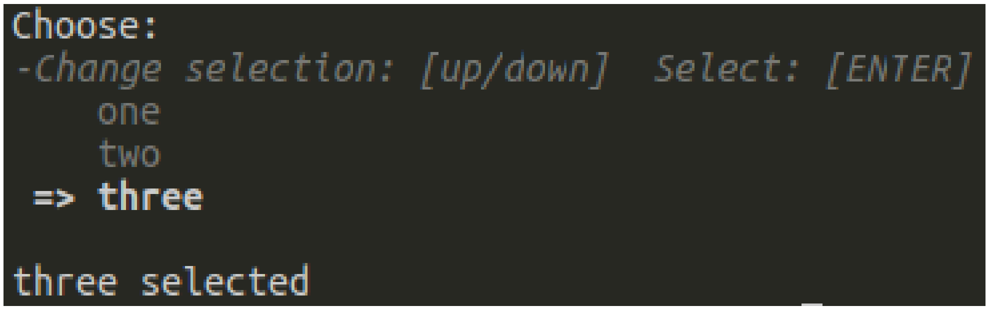
```
Choose:
-Change selection: [up/down]  Select: [ENTER]
    one
 => two
    three
```

options 1



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

options 2



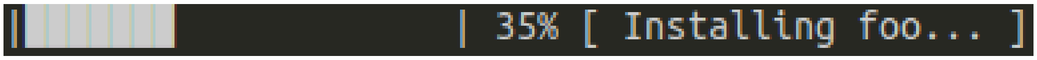
```
Choose:
-Change selection: [up/down]  Select: [ENTER]
    one
    two
 => three

three selected
```

choice

progress

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

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

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

progressBar .2 "Installing foo..."
```



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

progress

scan

Scan a host port range (tcp/udp).

```bash
#!/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
```

version compare | semver compare

There are different versioning standards out there but they are converging to a specific one called semver[13] over the last years. If you need to compare two different version strings which comply with `semver` semantic standard, then there is a ready to use function for it as `fn version compare` | `fn semver compare`. Insert the function into your function region:

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

# Usage: version_compare "1.2.3" "1.1.7"

function version_compare () {
  # function body omitted...
}
```

Later call the function according to usage or use `fx version compare` | `fx semver compare` snippet:

```bash
version_compare "major.minor.patch" "major.minor.patch"
```

`version_compare` function returns `>`, `<` or `=` as result. Here are some real example usage and the outputs:

```bash
version_compare "1.2.3" "1.1.7" # >
version_compare "1.1.1" "2.1.0" # <
version_compare "5.0.2" "5.0.2" # =
```

If the two version strings are not standard `semver`, `version_compare` function will compare each part (separated by `.`) until the shorter one reaches its end. If until that point both versions look like same it returns `=` as result:

[13] https://semver.org

```
version_compare "3.2.2.7" "3.2.2" # =
```

animation

Why on earth one may need ASCII animation in a shell script (you can use unicode as well but it may not work the same on all systems). To add some fun!

There are simple steps to make a beautiful animation with Shellman. You define frames and call animate function passing it the frames array and the interval between frames in seconds. Normally you need an animation at the end of your script, so animate function uses an infinite while loop of course with enough sleeps.

Your frames need to have the exact same width and height. If they are different in size, fill the unused space with spaces (no TABs).

Define animation frames at the top of your script. Here we make a simple spinner. We need frames |, /, -, \. Use animation frame snippet:

```
IFS='' read -r -d '' frames[1] <<"EOF"
|
EOF

IFS='' read -r -d '' frames[2] <<"EOF"
/
EOF

IFS='' read -r -d '' frames[3] <<"EOF"
-
EOF

IFS='' read -r -d '' frames[4] <<"EOF"
\
EOF
```

Now add animate function to your script using fn animation animate snippet and call it with frames array and desired delay between frames in seconds:

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

IFS='' read -r -d '' frames[1] <<"EOF"
|
EOF

IFS='' read -r -d '' frames[2] <<"EOF"
/
EOF

IFS='' read -r -d '' frames[3] <<"EOF"
-
EOF

IFS='' read -r -d '' frames[4] <<"EOF"
\
EOF

function animate () {
  local frames=("$@")
  ((last_index=${#frames[@]} - 1))
  local interval=${frames[last_index]}
  unset frames[last_index]

  # Comment out next two lines if you are using CTRL+C event handler.
  trap 'tput cnorm; echo' EXIT
  trap 'exit 127' HUP INT TERM

  tput civis # hide cursor
  tput sc # save cursor position

  while true; do
    for frame in "${frames[@]}"; do
      tput rc # restore cursor position
      echo "$frame"
      sleep "$interval"
    done
  done
}

animate "${frames[@]}" 0.2
```

You can find base frames for your animations by searching the web for `ascii art`. For more examples visit project repository page[14].

pacman

Using frames is not the best way to make animations but it is simple and straightforward. I have made another ready to use animation function for you which gets an string and eats it up.

Use `fn animation pacman` snippet to insert the function into your script and use `fx animation pacman` to call it:

```
function pac_man () {
  # Function body here...
}

pac_man "Hello World"
```

[14] https://github.com/yousefvand/shellman/tree/master/samples/animation

Solutions

Argument Parsing

Contents of `greet.sh`:

Argument Parsing

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

greeting="good night"
name="everyone"

# >>>>>>>>>>>>>>>>>>>>>>>> Argument parsing >>>>>>>>>>>>>>>>>>>>>>>>

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

# <<<<<<<<<<<<<<<<<<<<<<<< Argument parsing <<<<<<<<<<<<<<<<<<<<<<<<

echo "$greeting $name"
```

Nested Directories

Contents of `nested-directories.sh`:

Nested Directories

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

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

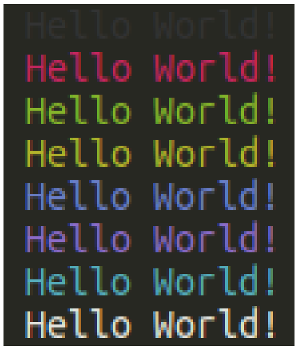
Colorful Text

Contents of `colorful-text.sh`:

Nested Directories

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

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

```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
```