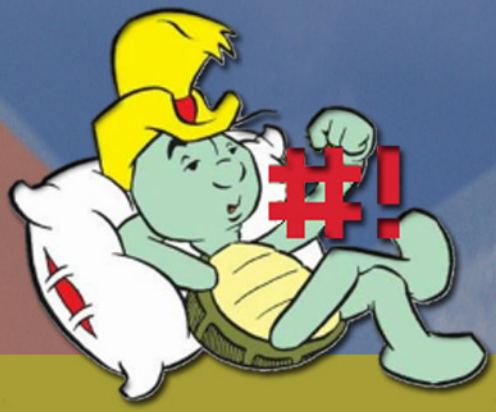
Shellman



#!BASH SCRIPTING

Remisa Yousefvand

Shellman Bash Scripting

Remisa Yousefvand

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

This version was published on 2024-07-04

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

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 in Unix systems.

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]. If you have any idea or issue you can contact me via: https://github.com/yousefvand/shellman-ebook/issues

Remisa Yousefvand

First edition July 2024

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

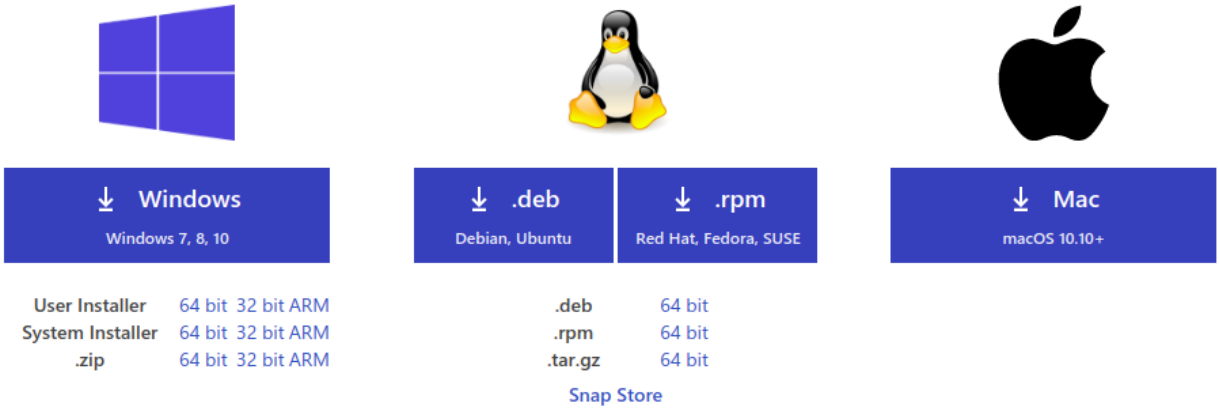
[2] https://code.visualstudio.com

[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

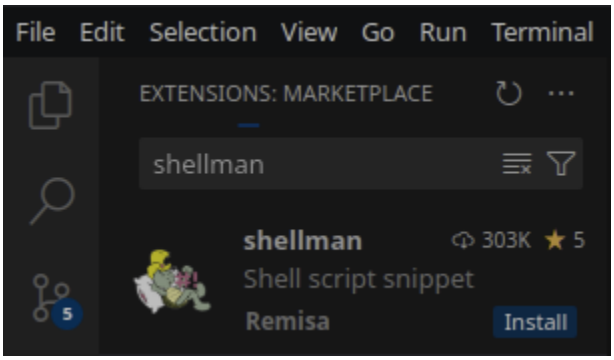
Prerequisites

- vscode[5] IDE



vscode download

- Shellman[6] snippet

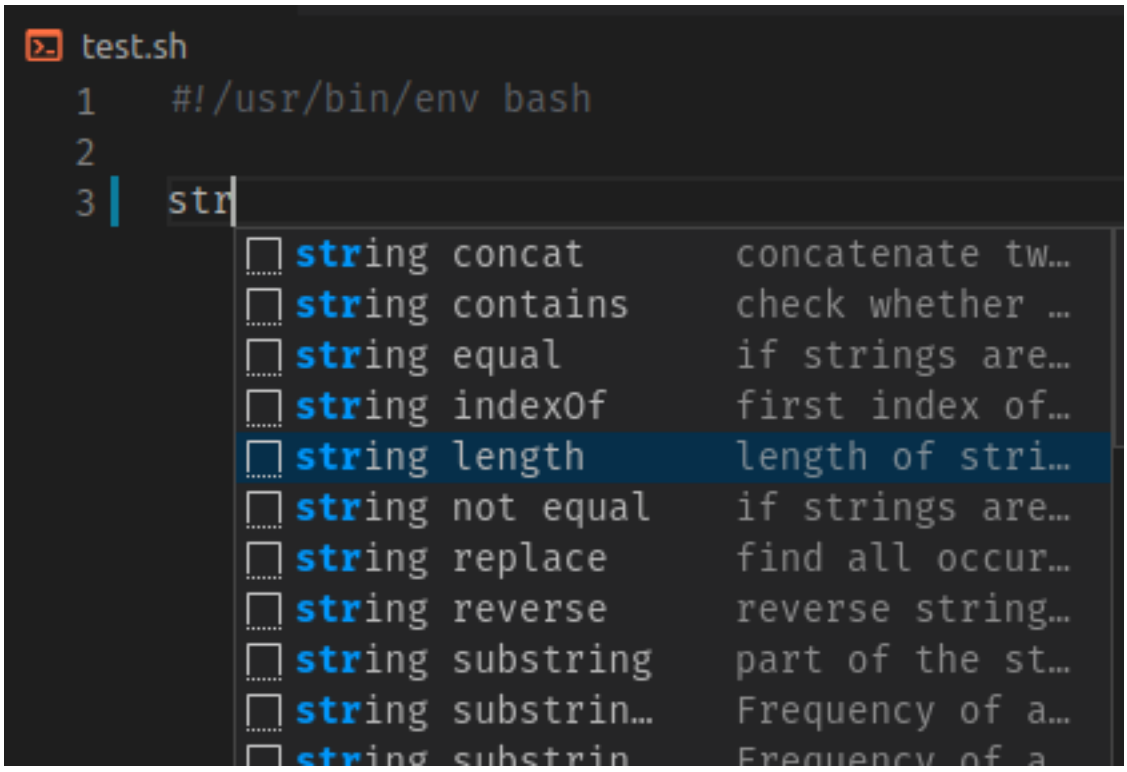


shellman install

[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.



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 most of shell scripts. 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 Shell Script

Shell script files by convention has **.sh** *file extension*[7]. To run a shell 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*).

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

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

Above script is the same as:

```
#!/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 (;).

```
#!/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`:

```
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}" # This syntax is encouraged
```

Variable Access Rule

To access a variable value prefix it with $

Variables are case sensitive (like Linux filesystem):

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

var=1 # small letter 'v'
Var=2 # capital letter 'V'

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

There are three variable types in shell scripting: **`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 as this is practical.

```
# 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 "${myArray[@]}"  # one two three
echo "${myArray2[@]}" # four five six
echo "${myArray[0]}"  # one
echo "${myArray2[2]}" # 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}"
}

myfunc hello # hello
```

Function definition should precedes its usage. `function` keyword is optional and can be omitted but for the sake of readability use it:

```
#!/usr/bin/env bash
myfunc () {
  echo "${1} ${2}"
}

myfunc hello world

# hello world
```

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.
  # Use shift (we will talk about it later) to process all 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)` # This method is encouraged

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 (|)**, 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*, it means no error happened and other values indicates command **failure** (1-255 equal to one byte).

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 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 refer to Solutions section, greet.

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 Shell 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`.

```
#!/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 "$var{1}" # Hello World!

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

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

```
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`.

To read global variables simply write: `echo $GLOBAL_VARIABLE`. One of the most famous global variables is **$PATH**. It consist of many paths separated by semicolon (;). When you run a command operating system searches these paths to find your command. If your command is not in those paths you need to enter the full path of your command.

[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*. If you have a background of Java, C# or other OOP languages then working with shellman is a piece of cake for you.

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 are available in `Shellman`. See fn-fx for more information.

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 | archive 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 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.

archive compress tar.xz | archive 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`)

```
#!/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 tar.gz

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

```
#!/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.

archive.compress-zip | archive zip

To compress popular and muliplatform your files use `zip` compression. Like this example:

```
zip -rq ./backup.zip ~/.config
```

archive decompress .zip | archive unzip

To unzip the zip file and get back your files and directories:

```
unzip -q ./archive.zip -d ./backup
```

array

Contains `Array` related operations.

array 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
```

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
```

Warning

Arrays start at zero index.

array concat

Returns an array made of concatenation of two given arrays.

```bash
#!/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.

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

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

array delete at

Delete Nth item in array (N = index)

```bash
#!/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.

```bash
#!/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
```

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

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

array iterate | array forEach

Iterate over array items.

```bash
#!/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
```

array contains

Check if the array contains an element.

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

if [[ "${myArray[*]}" =~ 'Bob' ]]; then
    echo 'array contains Bob'
fi
#array contains Bob
```

array reverse

Reverses order of array elements.

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

myArray=(1 2 3)

for((i=${#myArray[@]}-1;i>=0;i--)); do
    reversed+=("${myArray[i]}")
done

unset "myArray" # optional
echo "${reversed[@]}" # 3 2 1
```

command

Contains `command` execution related operations.

command substitution

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).

command success check | cmd success check

Check if last command has succeeded.

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

if ! touch ~/file.txt >/dev/null 2>&1; then
    echo "failed"
else
    echo "succeed"
fi
# command succeeded
```

command failure check | cmd failure check

Check if last command has failed.

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

if ! touch /file.txt >/dev/null 2>&1; then
    echo "failed"
else
    echo "succeed"
fi
# command failed
```

command nice | 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`.

command renice | 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 command exists | 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 or not.

hide command error | don't show command error

If a command fails don't show error (suppress stderr).

```
cp nofile.txt / 2> /dev/null
```

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=$(echo -n "Remisa Yousefvand" | base64)
echo "${base64Encoded}"
# UmVtaXNhIFlvdXNlZnZhbmQ=
```

crypto base64 decode

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

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

base64Decoded=$(echo -n "UmVtaXNhIFlvdXNlZnZhbmQ=" | base64 -d)
echo "${base64Decoded}" # Remisa Yousefvand
```

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

use desired algorithm when pressing TAB at second place.

date

Contains `Date` related operations.

date now short

Short version of current system *date*.

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

dateShort=$(date -I)
echo "$dateShort" # 2024-06-25
```

date now UTC

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

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

dateUTC=$(date -u)
echo "$dateUTC" # Tue Jun 25 11:32:55 AM UTC 2024
```

date now year

Current year.

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

year=$(date +%Y)
echo "$year" # 2024
```

date now monthNumber

Current month number.

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

monthNumber=$(date +%m)
echo "${monthNumber}" # 6
```

date now monthName

Current month name.

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

monthName=$(date +%B)
echo "${monthName}" # June
# %B for full month name, %b for abbreviated month name
```

date now dayOfMonth

Current day of month.

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

dayOfMonth=$(date +%d)
echo "$dayOfMonth" # 25
```

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" # 177
```

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.

file and directories

Contains `File` and `directory` related operations.

directory create

Create an empty directory.

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

mkdir "directory name"
```

if file exists

Check if the file exists.

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

if [ -f "/path/to/file" ]; then
    echo "File \"/path/to/file\" exists"
fi
```

if file not empty

Check if file size is greater than zero.

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

if [ -s "/path/to/file" ]; then
    echo "File \"/path/to/file\" is not empty"
fi
```

if file =

Check if files are equal.

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

if cmp -s "/path/to/file1" "/path/to/file2"; then
    echo "File \"/path/to/file1\" is equal to file \"/path/to/file2\""
fi
```

if file readable

Check if file can be read.

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

if [ -r "/path/to/file" ]; then
    echo "File \"/path/to/file\" is readable"
fi
```

if file writeable

if file is writeable.

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

if [ -w "/path/to/file" ]; then
    echo "File \"/path/to/file\" is writeable"
fi
```

if directory exists

Check if a directory exists.

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

if [ -d "/path/to/directory" ]; then
    echo "directory \"/path/to/directory\" exists"
fi
```

iterate files

Iterate files in a directory for a certain file (files).

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

# Searching in the current directory
for file in *.{jpg,png}; do
    echo "${file}"
done
```

In this example we are looking for pictures with `jpg` and `png` pictures.

iterate directories

Iterate around visible directories inside another directory.

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

# Make sure path ends with /
for directory in /*; do
    if [[ -d "${directory}" && ! -L "${directory}" ]]; then
        echo "${directory}"
    fi
done
```

directory create nested

Create directories even they don't exist.

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

mkdir -p a/b/c
```

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*. Home is denoted by ~.

directory delete nested | directory remove nested

Delete a directory and all subdirectory and files (use with caution).

```
rm -rf /path/to/directory
```

if path exists

If path exists (file, directory, link...)

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

if [ -e "/path/to/something" ]; then
    echo "Path \"/path/to/something\" exists"
fi
```

find file | find directory

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.

```
#!/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.

```
#!/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
```

if file executable

Check if file is executable.

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

if [ -x "/path/to/file" ]; then
    echo "file \"/path/to/file\" is executable"
fi
```

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.

if file link

If given path is a symbolic link.

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

if [ -h "/path/to/file" ]; then
    echo "Path \"/path/to/file\" is a symbolic link"
fi
```

if file newer

Check if file1 is newer than file2.

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

if [ "/path/to/file1" -nt "/path/to/file2" ]; then
    echo "Path \"/path/to/file1\" is newer than path \"/path/to/file2\""
fi
```

if file older

Check if file1 is older than file2.

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

if [ "/path/to/file1" -ot "/path/to/file2" ]; then
    echo "Path \"/path/to/file1\" is older than path \"/path/to/file2\""
fi
```

remove old/new files/directories

Find and remove files(f)/directories(d) older(+)/newer(-) than x days.

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

find "/path/to/directory/" -type f -mtime +days -delete
```

Mathematical operations

Mathematical operations on integers and non-integers in bash is different. But don't worry Shellman handles both. In this section we do non-integer math operations. It is suffice to add `float` or `double` in your snippet.

if float = | if double =

if two non-integers are equal:

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

i=7.6
j=7.6

if (( $(echo "${i} == ${j}" | bc -l) )); then
    echo "equal"
fi # equal
```

if float >= | if double >=

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

i=7.8
j=5.6

if (( $(echo "${i} >= ${j}" | bc -l) )); then
    echo "greater or equal"
fi # greater or equal
```

if float > | if double >

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

i=7.8
j=5.6

if (( $(echo "${i} > ${j}" | bc -l) )); then
    echo "greater"
fi # greater
```

if float <= | if double <=

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

i=4.8
j=5.6

if (( $(echo "${i} <= ${j}" | bc -l) )); then
    echo "lesser or equal"
fi # lesser or equal
```

if float < | if double <

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

i=4.8
j=5.6

if (( $(echo "${i} < ${j}" | bc -l) )); then
    echo "lesser"
fi # lesser
```

if float != | if double !=

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

i=4.8
j=5.6

if (( $(echo "${i} != ${j}" | bc -l) )); then
    echo "not equal"
fi # not equal
```

Ready functions

Sometimes you can do great things just if you have the right function. By having the right function you just need to call it. Sometimes calling the function needs to parametrize it which is very easy in compare to wring the whole function. Shellman have a namespace calling fn/fx which means by fn blah you insert a function in your code and bu fx blah you call your function. You can do amazing things like making animations by fn/fx.

{

#animation-fn-fx}][animation frame | fn animation animate | fx animation animate]{#animation-fn-fx}

besides `fn animation` and `fx animation animate` we need to borrow `animation frame` from `misc` namespace. The latter is responsible to hold your frames. Be careful your frames should be exact size. It means if a line is shorter than the longest line you should fill it by space. Then `fn animate` is responsible to animate fames as `fx animation animate` tells it. There are multiple example on website but I give a small taste here:

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

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

IFS='' read -r -d '' "frames[5]" <<"EOF"
# |
EOF

# Usage: animate framesArray interval
function animate () {
    local frames=("$@")

    ((lastIndex=${#frames[@]} - 1))
    local mode=${frames[lastIndex]}
    unset "frames[lastIndex]"

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

    # 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

```

```
    tput civis # hide cursor
    tput sc # save cursor position

    index=0
    max="${#frames[@]}"
    indices=()
    direction="forward"
    readarray -t forwardIndices < <(seq 0 1 "${max}")
    readarray -t backwardIndices < <(seq "${max}" -1 0)

    while true; do
        if [ "${mode}" = "circular" ]; then
            direction="forward"
        elif [ "${mode}" = "pendular" ]; then
            if (( index >= max )); then
                direction="backward"
            elif (( index <= 0 )); then
                direction="forward"
            fi
        else
            echo "Wrong mode! Valid modes: circular, pendular"
            exit 255
        fi

        if [ "${direction}" = "forward" ]; then
            indices=( "${forwardIndices[@]}" )
        else
            indices=( "${backwardIndices[@]}" )
        fi


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

# Usage: animate framesArray interval
animate "${frames[@]}" 0.1 circular
```

fn animation pacman / fx animation pacman

This is a ready to use animation. Pacman eats your letters.

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

# Usage: pacMan inputString interval pad
# Example: pacman "Hello World" 0.5 "*"
function pacMan () {
    local string="${1}"
    local interval="${2}"
    : "${interval:=0.2}"
    local pad="${3}"
    : "${pad:=.}"
    local length=${#string}
    local padding=""

    # 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

    for((i=0;i<=length;i++)); do
        tput rc
        echo "${padding}c${string:i:length}"
        sleep "$interval"
        tput rc
        echo "${padding}C${string:i:length}"
        sleep "${interval}"
        padding+="${pad}"
    done

    tput cnorm
    tput rc
    echo "${padding}"
}

# Usage: pacMan inputString interval pad
pacMan "Hello World" 0.1 "."
```

fn banner simple / fx banner simple

Create a simple banner around your text.

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

# Usage: bannerSimple "my title" "*"
function bannerSimple() {
    local msg="${2} ${1} ${2}"
    local edge
    edge=${msg//?/$2}
    echo "${edge}"
    echo "$(tput bold)${msg}$(tput sgr0)"
    echo "${edge}"
    echo
}

# Usage: bannerSimple "my title" "*"
bannerSimple "my title" "*"
```

fn banner color / fx banner color

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

# Usage: bannerColor "my title" "red" "*"
function bannerColor() {
    case ${2} in
        black) color=0
        ;;
        red) color=1
        ;;
        green) color=2
        ;;
        yellow) color=3
        ;;
        blue) color=4
        ;;
        magenta) color=5
        ;;
        cyan) color=6
        ;;
        white) color=7
```

```
        ;;
        *) echo "color is not set"; exit 1
        ;;
    esac

    local msg="${3} ${1} ${3}"
    local edge
    edge=${msg//?/$3}
    tput setaf ${color}
    tput bold
    echo "${edge}"
    echo "${msg}"
    echo "${edge}"
    tput sgr 0
    echo
}

# Usage: bannerColor "my title" "red" "*"
bannerColor "my title" "yellow" "*"
```

fn import / fx import

If you put your functions in a directory named `lib` by `import` you can easily import those functions into main script.

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

# Usage: import "mylib"
function import() {
    local file="./lib/${1}.sh"
    if [ -f "${file}" ]; then
        source "${file}"
    else
        echo "Error: Cannot find library at: ${file}"
        exit 1
    fi
}

# Usage: import "filename"
import "libname"
```

fn options / fx options | fn input choice / fx input choice

This function gives user multiple choices to select one.

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

# Usage: options=("one" "two" "three"); inputChoice "Choose:" 1 "${options[@]}"; cho\
ice=$?; echo "${options[$choice]}"
function inputChoice() {
    echo "${1}"; shift
    echo "$(tput dim)""- Change option: [up/down], Select: [ENTER]" "$(tput sgr0)"
    local selected="${1}"; shift

    ESC=$(echo -e "\033")
    cursor_blink_on()  { tput cnorm; }
    cursor_blink_off() { tput civis; }
    cursor_to()        { tput cup $(($1-1)); }
    print_option()     { echo "$(tput sgr0)" "$1" "$(tput sgr0)"; }
    print_selected()   { echo "$(tput rev)" "$1" "$(tput sgr0)"; }
    get_cursor_row()   { IFS=';' read -rsdR -p $'\E[6n' ROW COL; echo "${ROW#*[}"; }
    key_input()        { read -rs -n3 key 2>/dev/null >&2; [[ $key = ${ESC}[A ]] && \
echo up; [[ $key = ${ESC}[B ]] && echo down; [[ $key = "" ]] && echo enter; }

    for opt; do echo; done

    local lastrow
    lastrow=$(get_cursor_row)
    local startrow=$((lastrow - $#))
    trap "cursor_blink_on; echo; echo; exit" 2
    cursor_blink_off

    : selected:=0

    while true; do
        local idx=0
        for opt; do
            cursor_to $((startrow + idx))
            if [ ${idx} -eq "${selected}" ]; then
                print_selected "${opt}"
            else
                print_option "${opt}"
            fi
            ((idx++))
```

```
        done

        case $(key_input) in
            enter) break;;
            up)    ((selected--)); [ "${selected}" -lt 0 ] && selected=$(($# - 1));;
            down)  ((selected++)); [ "${selected}" -ge $# ] && selected=0;;
        esac
    done

    cursor_to "${lastrow}"
    cursor_blink_on
    echo

    return "${selected}"
}

# Usage: options=("one" "two" "three"); inputChoice "Choose:" 1 "${options[@]}"; cho\
ice=$?; echo "${options[$choice]}"
options=("one" "two" "three")
inputChoice "Choose:" 0 "${options[@]}"; choice=$?
echo "${options[$choice]}" selected
```

fn checkbox | fn input multichoice

User can have multiple choices.

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

# Usage: multiChoice "header message" resultArray "comma separated options" "comma s\
eparated default values"
function multiChoice {
    echo "${1}"; shift
    echo "$(tput dim)""- Change Option: [up/down], Change Selection: [space], Done: \
[ENTER]" "$(tput sgr0)"
    # little helpers for terminal print control and key input
    ESC=$( printf "\033")
    cursor_blink_on()   { printf "%s" "${ESC}[?25h"; }
    cursor_blink_off()  { printf "%s" "${ESC}[?25l"; }
    cursor_to()         { printf "%s" "${ESC}[$1;${2:-1}H"; }
    print_inactive()    { printf "%s   %s " "$2" "$1"; }
    print_active()      { printf "%s  ${ESC}[7m $1 ${ESC}[27m" "$2"; }
    get_cursor_row()    { IFS=';' read -rsdR -p $'\E[6n' ROW COL; echo "${ROW#*[}"; }
```

```
17      key_input()         {
18          local key
19          IFS= read -rsn1 key 2>/dev/null >&2
20          if [[ $key = ""      ]]; then echo enter; fi;
21          if [[ $key = $'\x20' ]]; then echo space; fi;
22          if [[ $key = $'\x1b' ]]; then
23              read -rsn2 key
24              if [[ $key = [A ]]; then echo up;    fi;
25              if [[ $key = [B ]]; then echo down;  fi;
26          fi
27      }
28      toggle_option()    {
29          local arr_name=$1
30          eval "local arr=(\"\${${arr_name}[@]}\")"
31          local option=$2
32          if [[ ${arr[option]} == 1 ]]; then
33              arr[option]=0
34          else
35              arr[option]=1
36          fi
37          eval "$arr_name"='("${arr[@]}")'
38      }
39
40      local retval=$1
41      local options
42      local defaults
43
44      IFS=';' read -r -a options <<< "$2"
45      if [[ -z $3 ]]; then
46          defaults=()
47      else
48          IFS=';' read -r -a defaults <<< "$3"
49      fi
50
51      local selected=()
52
53      for ((i=0; i<${#options[@]}; i++)); do
54          selected+=("${defaults[i]}")
55          printf "\n"
56      done
57
58      # determine current screen position for overwriting the options
59      local lastrow
```

```
    lastrow=$(get_cursor_row)
    local startrow=$((lastrow - ${#options[@]}))

    # ensure cursor and input echoing back on upon a ctrl+c during read -s
    trap "cursor_blink_on; stty echo; printf '\n'; exit" 2
    cursor_blink_off

    local active=0
    while true; do
        # print options by overwriting the last lines
        local idx=0
        for option in "${options[@]}"; do
            local prefix="[ ]"
            if [[ ${selected[idx]} == 1 ]]; then
                prefix="[x]"
            fi

            cursor_to $((startrow + idx))
            if [ $idx -eq $active ]; then
                print_active "$option" "$prefix"
            else
                print_inactive "$option" "$prefix"
            fi
            ((idx++))
        done

        # user key control
        case $(key_input) in
            space)  toggle_option selected $active;;
            enter)  break;;
            up)     ((active--));
                if [ $active -lt 0 ]; then active=$((${#options[@]} - 1)); fi;;
            down)   ((active++));
                if [ "$active" -ge ${#options[@]} ]; then active=0; fi;;
        esac
    done

    # cursor position back to normal
    cursor_to "$lastrow"
    printf "\n"
    cursor_blink_on

    indices=()
```

```
    for((i=0;i<${#selected[@]};i++)); do
        if ((selected[i] == 1)); then
            indices+=("${i}")
        fi
    done

    # eval $retval='("${selected[@]}")'
    eval "$retval"='("${indices[@]}")'
}

# Usage: multiChoice "header message" resultArray "comma separated options" "comma s\
eparated default values"
multiChoice "Select options:" result "One 1;Two 2;Three 3" "1;0;1"
echo "${result[@]}"
```

fn math average / fx math average

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

# Usage: average int1 int2 ...
i=5
j=8
k=12

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

# Usage: average int1 int2 ...
echo $(average "${i}" "${j}" "${k}") # 8
```

fn math factorial / fx math factorial

```bash
# Usage: factorial n
factorial ()
{
    if (( $1 < 2 )); then
        echo 1
    else
        echo $(( $1 * $(factorial $(( $1 - 1 ))) ))
    fi
}

# Usage: factorial n
readarray -t result < <(factorial 5)
echo "${result[@]}" # 120
```

fn math fibonacci series

An array of fibonacci series from zero to n.

```bash
# Usage: fibonacciSeries n
fibonacciSeries ()
{
    fib=()
    fib+=(0)
    fib+=(1)

    for((i=2;i<${1};i++)); do
        fib[i]=$((fib[i-1] + fib[i-2]))
    done

    echo "${fib[@]}"
}
# Usage: fibonacciSeries n
readarray -t result < <(fibonacciSeries 4)
echo "${result[@]}" # 0 1 1 2
```

fn math fibonacci / fx math fibonacci

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

# Usage: fibonacci n
fibonacci ()
{
    if (($1 < 2)); then
        echo "$1"
    else
        echo $(($(fibonacci $(($1 - 1))) + $(fibonacci $(($1 - 2)))))
    fi
}

# Usage: fibonacci n
echo "$(fibonacci 9)" # 34
```

fn math product / fx math product

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

# Usage: product int1 int2 ...
function product () {
    local result=1
    for int in "$@"; do
        ((result *= int))
    done
    echo "${result}"
}

# Usage: product int1 int2 ...
int1=2; int2=5; int3=6
echo $(product ${int1} ${int2} ${int3}) # 60
```

fn math sum / fx math sum

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

# Usage: sum int1 int2 ...
function sum () {
    local result=0
    for int in "$@"; do
        ((result += int))
    done
    echo "${result}"
}

# Usage: sum int1 int2 ...
int1=5; int2=7; int3=9
echo $(sum ${int1} ${int2} ${int3}) # 21
```

fn progress / fx progress

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

# Usage: progressBar "message" currentStep totalSteps
function progressBar() {
    local   bar='▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇'
    local space='....................'
    local wheel=('\' '|' '/' '-')

    local msg="${1}"
    local current=${2}
    local total=${3}
    local wheelIndex=$((current % 4))
    local position=$((100 * current / total))
    local barPosition=$((position / 5))

    echo -ne "\r|${bar:0:$barPosition}$(tput dim)${space:$barPosition:20}$(tput sgr0\
)| ${wheel[wheelIndex]} ${position}% [ ${msg} ] "
}

# Usage: progressBar "message" currentStep totalSteps
totalSteps=100

for ((currentStep=1; currentStep <= totalSteps; currentStep++)); do
    sleep 0.1 # simulating one step of job
    progressBar "Installing foo..." "${currentStep}" "${totalSteps}"
```

```
done

echo
```

fn scan local / fx scan local

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

# Usage: scan proto host fromPort toPort
function scan () {
    local openPortsArray=()
    for ((port=${3}; port<=${4}; port++)); do
        (echo >/dev/"${1}"/"${2}"/"${port}") >/dev/null 2>&1 && openPortsArray+=("${\
port}")
    done
    echo "${openPortsArray[@]}"
}

# Usage: scan proto host fromPort toPort
readarray -t openPorts < <(scan tcp "localhost" 1000 5000)
echo "${openPorts[@]}"
```

fn time format seconds

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

# Usage: formatSeconds 70 -> 1m 10s
function formatSeconds {
    local T=$1
    local D=$((T/60/60/24))
    local H=$((T/60/60%24))
    local M=$((T/60%60))
    local S=$((T%60))
    local result=""

    (( D > 0 )) && result="${D}d "
    (( H > 0 )) && result="${result}${H}h "
    (( M > 0 )) && result="${result}${M}m "
    (( S > 0 )) && result="${result}${S}s "
    echo -e "${result}" | sed -e 's/[[:space:]]*$//'
}
```

```
# Usage: formatSeconds 70 -> 1m 10s
readarray -t res < <(formatSeconds 80)
echo "${res[@]}"
```

fn urlencode / fx urlencode

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

# Usage: urlencode url
function urlencode () {
    local LC_ALL=C
    local c i n=${#1}
    for (( i=0; i<n; i++ )); do
        c="${1:i:1}"
        case "$c" in
            [[:alnum:].~_-]) printf '%s' "$c" ;;
            *) printf '%%%02x' "'$c" ;;
        esac
    done
}
# Usage: urlencode url
urlencode "Hello World!"
# Hello%20World%21
```

fn urldecode / fx urldecode

```
# Usage: urldecode url
function urldecode() {
: "${*//+/ }"; echo -e "${_//%/\x}";
}

urldecode "Hello%20World%21"
# Hello World!
```

fn version compare | fn semver compare

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

# Usage: versionCompare "1.2.3" "1.1.7"
function versionCompare () {
    function subVersion () {
        echo -e "${1%%"."*}"
    }
    function cutDot () {
        local offset=${#1}
        local length=${#2}
        echo -e "${2:((++offset)):length}"
    }
    if [ -z "${1}" ] || [ -z "${2}" ]; then
        echo "=" && exit 0
    fi
    local v1
    v1=$(echo -e "${1}" | tr -d '[:space:]')
    local v2
    v2=$(echo -e "${2}" | tr -d '[:space:]')
    local v1Sub
    v1Sub=$(subVersion "$v1")
    local v2Sub
    v2Sub=$(subVersion "$v2")
    if (( v1Sub > v2Sub )); then
        echo ">"
    elif (( v1Sub < v2Sub )); then
        echo "<"
    else
        versionCompare "$(cutDot "$v1Sub" "$v1")" "$(cutDot "$v2Sub" "$v2")"
    fi
}

# Usage: versionCompare "1.2.3" "1.1.7"
versionCompare "5.3.1" "5.2.8"
# >
```

ftp

Contains `FTP` related operations.

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 download

Download specified file from ftp server.

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

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

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 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"
```

ftp upload

Upload specified file to ftp server at desired path.

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

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

function

Contains `function` related operations. A function can return a number between `0` to `255` which can be retrieved through `$?` (available as `function return value` snippet).

function | func

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

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

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

# Usage:
myFunction "some argument" "another argument"
# some argument
# another argument
```

function arguments | func args

Access to function arguments.

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

function myFunction () {
  echo "$@"
}

myFunction "some argument" "another argument"
# Usage:
# some argument another argument
```

function arguments count | func args count

Number of function arguments.

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

function myFunction () {
  echo $#
}

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

function return | func ret val

Check the value of 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 => an error occurred
```

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 begin | git start

starts a git in the directory by running `git init` command and set `username` and `email` locally,globally or systemic. Repeating this command overwrite previous values.

```
git config --global user.name "remisa yousefvand"
git config --global user.email remisa.yousefvand@gmail.com
```

git branch create

Create a local branch and switch into it.

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

git checkout -b develop
```

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 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 changes revert

Revert tracked changes.

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

git checkout .
```

git clone branch https

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

git clone -b develop git@github.com:remisa/shellman.git
```

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 clone https

```
git clone https://github.com/yousefvand/shellman.git
```

git clone

Clone a repository to local machine.

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

git clone git@github.com:yousefvand/shellman.git
cd repository
```

git commit list notPushed

List non pushed commits.

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

git log origin/master..HEAD
```

git commit search

Search for a commit which contains searchCriteria.

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

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

git 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
```

git commit

```
git commit -m "fixes #12, #13, #15"
# or
git commit -m "fixed typo"
```

git config list

List a brief configuration of local git configuration..

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

git config --list
```

git patch apply

Apply a patch from file.

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

git apply < patch1.patch
```

git patch create

Create a patch from changes.

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

git diff > patch1.patch
```

git remote list

List all remotes.

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

git remote
```

git remote urlAdd https | git remote url add https

Add remote url using HTTPS.

```
git remote add origin https://github.com/yousefvand/shellman.git
```

git remote urlAdd | git remote url add | git remote urlAdd ssh | git remote url add ssh

Add remote url using SSH.

```
git remote set-url origin git@github.com:yousefvand/shellman.git
```

git remote urlChange https | git remote url change https

Change remote url using HTTPS.

```
git remote set-url origin git@github.com:yousefvand/shellman.git
```

git remote urlChange | git remote url change | git remote urlChange ssh | git remote url change ssh

Change remote url using SSH.

```
git remote set-url origin git@github.com:yousefvand/shellman.git
```

git tag commit

Tag a commit.

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

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

git tag list

List all tags.

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

git tag
```

git tag remote delete

Delete tag from remote.

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

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

git tag remote push

Push tag to remote.

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

git push origin tagName
```

http

Contains `HTTP` related operations.

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'
```

http 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[12].

http header

Send http request with custom header(s).

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

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

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

http POST file

Send file with *http POST*.

```
#!/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 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'
```

http DELETE

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

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

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

User input

Now it is password and simple text but in the future some other methods like single and multichoice should move here.

input password

```
echo "Please enter your password: " # 1234
read -rs password
echo "${password}" # 1234
```

input text | ask question

Get simple text from user and put in a variable.

```
read -rep "what are you reading? " -i "Shellman ebook" answer
echo "${answer}"
```

Math operations on integers

if int =

if some integer is equal to other integer.

```
if (( 78 == 78 )); then
    echo "equal"
fi
# equal
```

if int >=

if some integer is greater or equal of another integer.

```
if (( 45 >= 44 )); then
    echo "greater or equal"
fi
```

if int >

if some integer is greater than another integer.

```
if (( 65 > 44 )); then
    echo "greater"
fi
```

if int <=

```
if (( 89 <= 90 )); then
    echo "lesser or equal"
fi
```

if int <

```
if (( 22 < 35 )); then
    echo "lesser"
fi
```

if int !=

if two integers are not equal.

```
if (( 68 != 69 )); then
    echo "not equal"
fi
```

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
```

infinite loop

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

while true; do
  echo "message every 2 seconds"
  sleep 2s
done
```

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
```

if

This snippet is a full if/else if/else.

```
if [ condition ]; then
     # if body
elif [ condition ]; then
     # else if body
else
     # else body
fi
```

if

This is a simple if.

```
if condition; then
    command ...
fi
```

iff

Short circuit AND.

```
[ condition ] && command
```

if condition is true then command executed otherwise command doesn't executed.

iff not

Short circuit OR.

```
[ condition ] || command
```

if condition is true or false then command executed.

switch case

```
case "${item}" in
    1)
        echo "item = 1"
    ;;
    2|3)
        echo "item = 2 or item = 3"
    ;;
    *)
        echo "default (none of above)"
    ;;
esac
```

ip

Contains `ip` related operations.

ip local IPs

Array of local IPs.

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

IPS=$(hostname)
echo "$IPS"
```

ip info

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

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

"$(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"
```

math

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

math + (add)

Given two variables, returns sum of them.

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

var1=2
var2=3
result=$((var1 + var2))
echo "$result" # 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 * (multiply)

Given two variables, returns product of them.

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

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

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 ^ (power)

Exponentiate *base* to the *power*.

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

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

math 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
```

math ++ (increase)

Given a variables, adds one to it.

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

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

math – (decrease)

Given a variable, subtracts one from it.

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

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

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 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 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 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 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 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.

- Pi = 3.14159265358979323846264338327950288
- Euler's number = 2.7182818284590452353602 8
- Gamma = 0.57721566490153286060651209008240
- Omega = 0.56714329040978387299996866221035
- Phi = 1.61803398874989484820458683436563 81

miscellaneous

Contains `other` operations not available in namespaces.

am I not root | am I not sudo

```bash
if (( $(id -u) != 0 )); then
    echo "I'm not root"
fi
```

am I root | am I sudo

```bash
if (( $(id -u) == 0 )); then
    echo "I'm root"
fi
```

animation frame

```
# Your frames need to have the exact same width and height.
# If they are different in size, fill unused space with `space`s (no `TAB`s).
IFS='' read -r -d '' "frames[1]" <<"EOF"
# Frame here
EOF
```

argument parsing

```
POSITIONAL=()
while (( $# > 0 )); do
    case "${1}" in
        -f|--flag)
        echo flag: "${1}"
        shift # shift once since flags have no values
        ;;
        -s|--switch)
        numOfArgs=1 # number of switch arguments
        if (( $# < numOfArgs + 1 )); then
            shift $#
        else
            echo "switch: ${1} with value: ${2}"
            shift $((numOfArgs + 1)) # shift 'numOfArgs + 1' to bypass switch and it\
s value
        fi
        ;;
        *) # unknown flag/switch
        POSITIONAL+=("${1}")
        shift
        ;;
    esac
done

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

echo text | print text

```
echo 'text here'
```

echo variable | print variable

```
echo "${variable}"
```

exit code

```
exit 0
```

os is

```
OS=$(awk -F'=' '/^ID=/ { gsub("\"","",$2); print tolower($2) }' /etc/*-release 2> /d\
ev/null)
echo "$OS" #arch
```

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).

shebang | bash | first line

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

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
```

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
```

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
```

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
```

output

color black

```
echo "$(tput setaf 0)"black text"$(tput sgr0)"
```

color blue

```
echo "$(tput setaf 4)"blue text"$(tput sgr0)"
```

color cyan

```
echo "$(tput setaf 6)"cyan text"$(tput sgr0)"
```

color green

```
echo "$(tput setaf 2)"green text"$(tput sgr0)"
```

color magenta

```
echo "$(tput setaf 5)"magenta text"$(tput sgr0)"
```

color red

```
echo "$(tput setaf 1)"red text"$(tput sgr0)"
```

color white

```
echo "$(tput setaf 7)"white text"$(tput sgr0)"
```

color yellow

```
echo "$(tput setaf 3)" text"$(tput sgr0)"
```

format bold

```
echo "$(tput bold)"bold text"$(tput sgr0)"
```

format dim

```
echo "$(tput dim)"dimmed text"$(tput sgr0)"
```

format italic

```
echo "$(tput sitm)"italic text"$(tput sgr0)"
```

format reverse

```
echo "$(tput rev)"reversed text"$(tput sgr0)"
```

process

Contains `Process` related information and operations.

process ID(s)

Get process ID(s) by its name. Many Linux commands need *process ids* (PIDs).

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

readarray -t arr < <(pgrep code)
echo "${arr[@]}"
# 2005 2010 2011 2015 2098 2110 2168 2250 2251 2276 2316 3002 3708 3759 3771 3903 44\
12 5847 5863 6872 6901
```

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. The easier solution is `killall firefox` or `pkill firefox`.

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 name by id

find process name by its id. If process has many ids each one do the job. Since the process may have only one id we use zero member of array. Open firefox.

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

# let's get id of firefox
readarray -t arr < <(pgrep firefox)
id="${arr[0]}"

# now let's get name of above id
processName=$(ps -p "${id}" -o comm=)
echo "$processName"
```

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.

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

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

string indexOf

Returns the first index of a substring inside a string.

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

string="hello world"
substring="world"
prefix=${string%%"$substring"*}
index=${#prefix}
if [[ index -eq ${#string} ]]; then
    echo -1
else
    echo "$index" # 6
fi
```

if string empty

Check if variable is an empty string.

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

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

if string not empty

Check if variable is not an empty string.

```
#!/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 a given string.

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

var="abcdefg"
length=${#var}
echo "$length" # 7
```

string random

Generates a random string from specified characters with desired length.

```
    randomString=$(tr -dc A-Za-z0-9 </dev/urandom | head -c 16 ; echo '')
    echo "$randomString"
```

In above example random string is built up from characters `A-Z`, `a-z` and `0-9` and length of it would be 16 characters.

string replace once

Replace the first substring with given string in another string.

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

new="Everyone"
old="world"
string="Hello world"

result="${string/$old/$new}"
echo "$result" # Hello Everyone
```

string replace all

Replace all the substrings with given string in another string.

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

new="Everyone"
old="world"
string="Hello world, bye world"

result="${string//$old/$new}"
echo "$result" # Hello Everyone, bye 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*.

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

string="abcdefg"
offset=2
length=3

substring=$(echo -e "${string:${offset}:${length}}")
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).

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

string="abcdefgcd"
substring="c"
tmp="${string//$substring}" && frequency=$((${#string} - ${#tmp}) / ${#substring}))
echo "${frequency}" # 2
```

string toLower

Returns lowercase of given string.

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

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

string toUpper

Returns uppercase of given string.

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

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

string trim

Removes leading and trailing whitespace(s).

```bash
#!/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).

```bash
#!/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
```

system

Contains `System` related information and operations.

system distro codename

Operating System codename (i.e. xenial).

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

distroCodeName=$(lsb_release -c | awk '{print $2}')
echo "${distroCodeName}" # n/a
```

system distro name

Operating System ID (i.e. Ubuntu).

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

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

system distro version

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

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

distroVersion=$(lsb_release -r | awk '{print $2}')
echo "${distroVersion}" # rolling
```

system kernel name

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

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

kernelName=$(uname -s)
echo "${kernelName}" # Linux
```

system kernel release

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

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

kernelRelease=$(uname -r)
echo "${kernelRelease}" # 6.9.7-arch1-1
```

system memory info

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

- MemTotal
- MemFree

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

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

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

system processor architecture

Processor architecture (i.e. x86_64).

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

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

system processor count

Number of processors (cores).

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

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

system processor model

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

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

cpuModel=$(lscpu | grep 'Model name' |cut -d ' ' -f 3- | sed -e 's/^[[:space:]]*//')
echo "${cpuModel}" # Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
```

system processor type

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

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

cpuType=$(uname -p)
echo "${cpuType}" # unknown
```

system service manage

Manage service (daemon) operations.

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

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

system uptime seconds

system uptime in seconds.

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

# Use 'fn time format seconds' snippet for formatting
systemUptime=$(awk '{print $1}' /proc/uptime)
echo "${systemUptime}" # 4432.54
```

system uptime

System uptime (hh:mm:ss).

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

Uptime=$(uptime -p)
echo "${Uptime}" # up 18 minutes
```

time

Contains `Time` related operations.

time seconds epoch

Seconds from 01-01-1970 00:00.

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

timeNowSecondsEpoch=$(date +%s)
echo "${timeNowSecondsEpoch}"
#1719942046
```

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}"
# 17:45
```

Working with variables

variable assign | variable set

Setting a variable.

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

variable='value'
```

variable default value | assign if empty

Set default value for a variable if it never set by programmer.

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

variable read | variable expand

Standard variable format.

```
"${result}"
"${variable}"
```

Solutions

Argument Parsing

Contents of `greet.sh`:

Argument Parsing

```
#!/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

```
#!/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 $i)"cyan text"$(tput sgr0)"
done
```

Output:

colorful text

Greet

Contents of `greet.sh`:

Greet

```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"
```

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
```