Lets Learn Shell Programming

Topics Covered

- Variables
- Return code
- Debugging
- Conditional Statements
- Arrays
- Associate Arrays
- Arithmetic Operations
- Bash Functions
- Files
- JSON

BASH

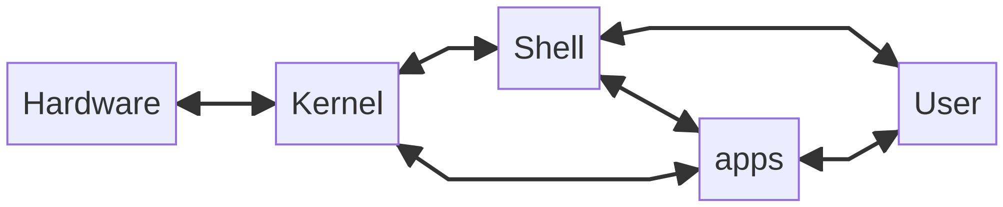
THE BOURNE-AGAIN SHELL

Note: This document will be always be in work in progress, thus few topics will feel like not fully developed.

Mayank Johri, Meenakshi Johri, Ver: 0.0.1

Introduction

One of the core component in any UNIX or GNU/Linux OS is Shell, which sits between Kernel and User



*inx Shells

A *inx shell is both a

- **Command interpreter**: The shell provides the user interface to the rich set of GNU utilities
- **Programming language**: It allow these GNU utilities to be combined to perform a specific task.

There are many Shells available, for full list visit (https://en.wikipedia.org/wiki/Comparison_of_command_shells).

- Tcsh (Tenex C Shell)
- Zsh (Z-Shell)
- Fish (Friendly Interactive Shell)
- Ksh (Korn Shell)
- Bourne Shell (sh)
- Bash (GNU Bourne-Again Shell)

Most common features of the Shell's are

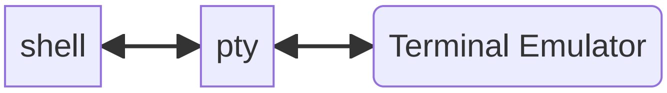
Name	Description
Background execution	Ability to run a command without user interaction in the terminal
Completions	Ability to help users in typing commands, by suggesting matching words for incomplete typed ones and is generally requested by pressing often the Tab ↹ key.
Command history	Ability to allow users find, modify and run previously executed command with its parameters
Automatic suggestions	Ability to auto suggest the commands based on few criteria
Auto-correction	Ability to use spell check to automatically correct common typing mistakes

Name	Description
Coloured directory listings	Ability to allow user set a colour scheme for displaying file and directory names in directory listings based on their extensions, type and permissions

The Terminal Emulator

Users cannot directly access the shell, we use Terminal Emulators to access shell. Terminal Emulator is a graphical program which facilitate the users to interact with the Shell. It also interpret & data sent by the Shell and displays the data in appropriate format which is usually textual in nature.

Both terminal emulator and shell are connected using `pty` (pseudo terminal) providing bi-directional asynchronous communication channel communication.

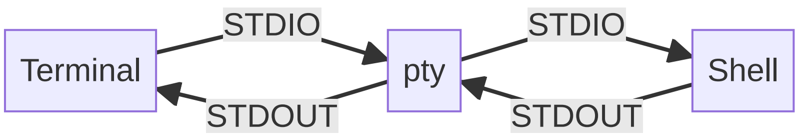


Working of terminal

When terminal starts, it initiates the `pty` and `shell` is also started. When the user enters the command(s), they are stored in `pty` and passed to shell only when `Enter` key is pressed. Shell processed the command(s) and returns the data back to terminal using `pty`

> A **pseudo-terminal** (sometimes abbreviated "pty") is a pair of virtual character devices that provide a bidirectional communication channel. One end of the channel is called the master; the other end is called the slave.
>
> > **Reference**: https://man7.org/linux/man-pages/man7/pty.7.html

The communication channel which communicate from terminal to shell is called `STDIO` and one which communicates from shell to terminal is called `STDOUT` as shown in the below diagram.



Here are some free, commonly-used terminal emulators by operating system (from: https://packages.gentoo.org/categories/x11-terms):

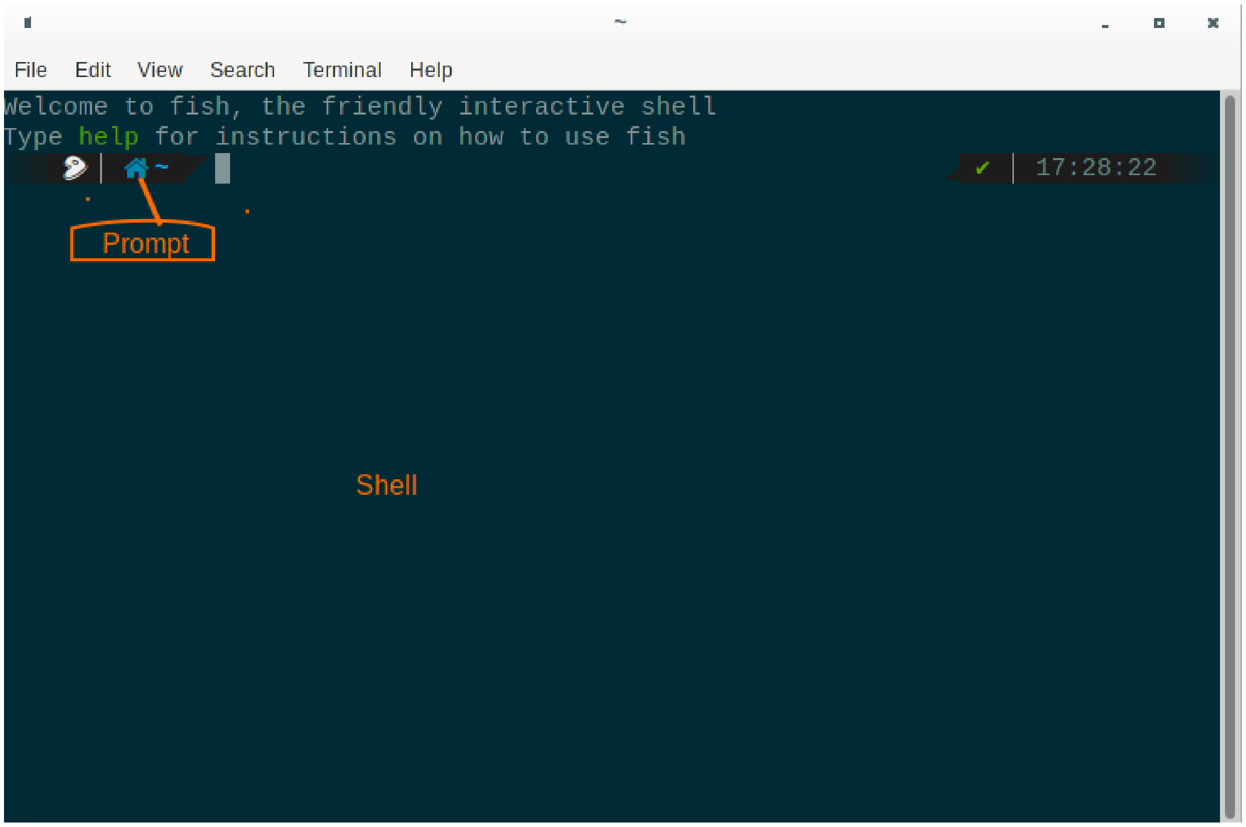
Name	URL	Description
alacritty	https://github.com/alacritty/alacritty	GPU-accelerated terminal emulator
aterm	http://www.afterstep.org/aterm.php	rxvt compatible terminal emulator with transparency support

Name	URL	Description
cool-retro-term	https://github.com/Swordfish90/cool-retro-term	terminal emulator which mimics the look and feel of the old cathode tube screens
Eterm	https://github.com/mej/Eterm	Terminal emulator intended as a replacement for xterm and designed for the Enlightenment desktop
gnome-terminal	https://wiki.gnome.org/Apps/Terminal	A terminal emulator for GNOME
guake	http://guake-project.org/	Drop-down terminal for GNOME
kitty	https://github.com/kovidgoyal/kitty	Fast, feature-rich, GPU-based terminal
mate-terminal	https://www.mate-desktop.org/	The MATE Terminal
roxterm	https://roxterm.sourceforge.net/	A terminal emulator designed to integrate with the ROX environment
rxvt-unicode	http://software.schmorp.de/pkg/rxvt-unicode.html	rxvt clone with xft and unicode support
sakura	https://www.pleyades.net/david/projects/sakura	GTK/VTE based terminal emulator
terminator	https://gnome-terminator.org	Multiple GNOME terminals in one window
terminology	https://www.enlightenment.org/about-terminology	Feature rich terminal emulator using the Enlightenment Foundation Libraries
tilda	https://github.com/lanoxx/tilda/	A drop down terminal, similar to the consoles found in first person shooters
wezterm	https://wezfurlong.org/wezterm/	A GPU-accelerated cross-platform terminal emulator and multiplexer
xfce4-terminal	https://docs.xfce.org/apps/terminal/start	A terminal emulator for the Xfce desktop environment
xterm	https://invisible-island.net/xterm	Terminal Emulator for X Windows
zutty	https://tomscii.sig7.se/zutty/	A high-end terminal for low-end systems.

All the above terminal emulators are great for their respective jobs.

The terminal consists of few items.

- The Command Prompt
- Shell



```
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
 ~                                                          ✔ │ 17:28:22
```

The Command Prompt

Although `prompt` is not part of Shell, it helps in improving the overall experience. The prompt can be customised using special characters. whose full list as it appears in the Bash manual:

Char	Action
`\a`	A bell character
`\d`	The date, in “Weekday Month Date” format
`\D{format}`	The format is passed to strftime(3) and the result is inserted into the prompt string; an empty format results in a locale-specific time representation. The braces are required
`\e`	An escape character:
`\H`	The full hostname:
`\h`	The hostname, up to the first `‘.’`
`\j`	The number of jobs currently managed by the shell
`\l`	The basename of the shell’s terminal device name

Char	Action
`\n`	A newline
`\r`	A carriage return
`\s`	The name of the shell, the basename of $0 (the portion following the final slash)
`\t`	The time, in 24-hour `HH:MM:SS` format
`\T`	The time, in 12-hour `HH:MM:SS` format:
`\@`	The time, in 12-hour AM/PM format
`\A`	The time, in 24-hour `HH:MM` format
`\u`	The username of the current user
`\v`	The version of Bash (such as: 5.2)
`\V`	The release of Bash with its associate patch level (such as 5.2.2)
`\w`	The current working directory, with $HOME abbreviated with a tilde (uses the $PROMPT_DIRTRIM variable)
`\W`	The basename of $PWD, with $HOME substituted with a tilde `~`
`\!`	The history number of this command
`\#`	The command number of this command
`\nnn`	The character whose ASCII code is the octal value nnn:
`\\`	Backslash
`\[`	The start of non-printing characters
`\]`	The end of non-printing characters

Example:

```
set PS1="\u \d >"
```

Output: The prompt changed to

```
mayank Wed Mar 29 >
```

Example:

```
PS1="\u@\h: \w\a\]$"
```

Output: The prompt changed to

```
mayank@mayank-testbox:~$
```

Example: The date and time along with working directory

```
PS1="[\d \t] \u@\h\n\w\$ "
```

Output:

```
[Wed Mar 29 08:04:23] mayank@mayank-testbox
~$
```

Prompt Variable	Meaning
`PS1`	The primary prompt display
`PS2`	The secondary prompt string. (It is also used to display when a long command is broken into sections with the `\` sign.)
`PS3`	The prompt for the select command.
`PS4`	The prompt for running a shell script in debug mode

> We can even try https://ohmyz.sh/#install to customise your shell.

Shell Script

Shell script is a **plain text file** containing **collection of commands**. Shell script is just like batch file is to MS-DOS/Windows but have more power than the MS-DOS batch file. `powershell` might come near to what *inx shell scripting can do.

How to write shell scripting

Shell scripts can be written using any text editor including `vi`, `emacs`, `pico`, `nano`, etc. Once the script is completed, we need to set permission for it to be able to execute using one of the following command

- ***chmod +x*** *# Only adds execute permission for everyone. And do not change other permissions*
- **chmod 755** # Read/Write/Execute permission to author and Read /Execute but not write permission to everyone else.
- **chmod 555** # Read /Execute but not write permission to everyone.

Once the permission is set then the script can be executed using "`./<filename>`" command if the file is in the same folder from where you are executing it.

Sample shell script

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

echo Hello World
echo “Hello World”
echo \“Hello World\”
```

Lets explain few things about the above file.

- **first line** is **very important** as it tells the shell which command to use to execute this script and is called ***Shebang***. Say you are using `fish` as your shell then, script written for `bash` might not work, thus if you provide the details of bash on the top of script then `fish` will call `bash` to execute this script.

Syntax:

```
#!interpreter [arguments]
```

There are few other ways it can be populated

- Using entire path

  ```
  #!/bin/bash

  echo Hello World
  echo “Hello World”
  echo \“Hello World\”
  ```

  One problem with this method is that if the `bash` command is not located in `/bin/bash` then the file will fail to execute.

- Using `env`

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

  echo "using env"
  ```

  In this example, we are using env to provide the path of bash. So we don't have to know the location of the program and `env` will find the `bash` and use it to execute the file.

Overriding the *Shebang*

We can run the shell script using our shell of choosing, as we are using `ksh` to execute the script `users.sh`

```
ksh users.sh
```

or if your script has been developed with `fish` shell in mind, then run the script using `fish` command as shown below.

```
fish users.sh
```

Basic Shell Features

BASH is an acronym for *Bourne Again Shell* as its based on *Bourne Shell* and is feature compatible.

Shell Syntax

When the shell reads input, it proceeds through a sequence of operations. If the input indicates the beginning of a comment, the shell ignores the comment symbol (‘#’), and the rest of that line.

Otherwise, roughly speaking, the shell reads its input and divides the input into words and operators, employing the quoting rules to select which meanings to assign various words and characters.

The shell then parses these tokens into commands and other constructs, removes the special meaning of certain words or characters, expands others, redirects input and output as needed, executes the specified command, waits for the command’s exit status, and makes that exit status available for further inspection or processing.

Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion

Escape Character `\`

Similar to `C` & `python` language, escape characters are used to remove the special meaning from a single character. A non-quoted backslash, \, is used as an escape character in Bash.

Single Quotes

Single quotes ('') are used to preserve the literal value of each character enclosed within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

Double Quotes

Using double quotes the literal value of all characters enclosed is preserved, except for the dollar sign, the backticks (backward single quotes, ``) and the backslash.

ANSI-C Quoting

Char	Meaning
\a	alert (bell)

Char	Meaning
\b	backspace
\e \E	an escape character (not ANSI C)
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab \v \ backslash \' single quote " double quote \? question mark
\v	vertical tab
\b	backslash
'	single quote
"	Double Quote
\?	Question mark

Comments

We can use `#` to denote comments. Shell will ignore all the text following `#` on that line.

Other components of Shell

- **Shell keywords** such as `if..else`, `do..while`.
- **Shell commands** such as `pwd`, `echo`, `continue`, type. (Detailed in Section 1)
- **Linux binary commands** such as `ls`, `who`, `whoami` etc.. (Detailed in Section 1)
- **Text processing utilities** such as `grep`, `awk`, `cut`. (Detailed in Section 1)
- **Functions** - add frequent actions together via functions. For example, `/etc/init.d/functions` file contains functions to be used by most or all system shell scripts in the `/etc/init.d` directory.
- **Control flow** statements such as `if..then..else` or shell loops to preform repeated actions.
- **Compound Commands**, etc

Variables

There are two types of variables

- **System Variable**: They are defined & maintained by Linux (OS) itself. They are defined by capital letters such as PATH, SHELL, OSTYPE, etc. They are also called environment variables.

VAR	Meaning

VAR	Meaning
`CDPATH`	**A colon-separated list of directories used as a search path for the cd builtin command.**
`HOME`	The current user’s home directory; the default for the cd builtin command. The value of this variable is also used by tilde expansion.
`IFS`	**A list of characters that separate fields; used when the shell splits words as part of expansion.** (Needs Examples)
`MAIL`	If this parameter is set to a filename or directory name and the `MAILPATH` variable is not set, Bash informs the user of the arrival of mail in the specified file or Maildir-format directory.
`MAILPATH`	A colon-separated list of filenames which the shell periodically checks for new mail. Each list entry can specify the message that is printed when new mail arrives in the mail file by separating the filename from the message with a ‘?’. When used in the text of the message, $_ expands to the name of the current mail file.
`OPTARG`	The value of the last option argument processed by the `getopts` builtin. (to be used within the script file)
`OPTIND`	The index of the last option argument processed by the `getopts` builtin. (to be used within the script file)
`PATH`	A colon-separated list of directories in which the shell looks for commands. A zero-length (null) directory name in the value of PATH indicates the current directory. A null directory name may appear as two adjacent colons, or as an initial or trailing colon.
`PS1`	The primary prompt string. The default value is ‘\s-\v$ ’.
`PS2`	The secondary prompt string. The default value is `>`’. `PS2` is expanded in the same way as `PS1` before being displayed.

> - The system variables might change depending on the shell on which you are running, such as `PS1`, `PS2` might not be populated on other shell such as fish.

- **User Defined Variable**: They are defined & maintained by users and should be defined by using lower cases letters such as `hello`, `var1`, etc. They can be defined by using the following syntax

```
Var=12
username="Mayank Johri"
# And not like, the below one will return an error
var = 12
```

There cannot be any space between the variable, `=` and its corresponding value as shown at line 4 in above example. And will result in error as shown in the below output

Output:

```
$> bash 00_variable_error.py
00_variable_error.py: line 4: var: command not found
```

Example: Fixed spacing issue

```
# 00_variable_error_fixed.py
Var=12
username="Mayank Johri"
# And not like, the below one will return an error
var=12
```

Naming convention for Variables

- Variable names must begin with alpha (***not numeric***) character or underscore (_) followed by one or more alphanumeric characters
- Spaces are not allowed on either side of the equal sign when assigning values to variable
- `Null` variable can be defined as

  ```
  nu1l=
  nul1=
  null=""
  ```

- Variables are case-sensitive
- Special characters such as `? , ( ) - . *` etc can not be used in variable names. Only exception is underscore.

Exercise

Q: ***Find the valid variable names***

- Welcome
- welcome
- welc0me
- 1Welcome
- _welcome
- +welcome
- wel+come
- user@india
- _

Defining Variable

```
1  # cat surya_namaskar.sh
2  #!/usr/bin/bash
3  
4  yoga_001="सूर्य नमस्कार "
5  # Do not leave spaces between the variable and data similar
6  # to the one shown below.
7  # yoga_001 = "सूर्य नमस्कार "
8  echo $yoga_001
```

output:

```
सूर्य नमस्कार
```

we can define variables using `yoga_001="सूर्य नमस्कार "` and access `yoga_001` variable prefixing `$` sign before the variable name as shown in line 8.

In the above example, `yoga_001` can only store one value, thus its called **scalar variables**, as it can hold only one value at a given time.

Lets read value from a command and store the output of it to a variable.

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

folder_contents=$(ls -l)
echo "Folder Contains:"
echo "$folder_contents"
```

Output:

```
 Folder Contains:
 total 12
 -rw-r--r-- 1 mayank mayank  94 Aug 25 07:52 listdir.sh
 -rw-r--r-- 1 mayank mayank 104 Aug 25 04:00 readonly.sh
 -rw-r--r-- 1 mayank mayank 107 Aug 25 06:59 unset.sh
```

We can also use ` for running the shell commands as shown at **line 5** in below example .

```
1  
2  #!/usr/bin/env bash
3  # listfolder.sh
4  
5  folder_contents=`ls -l`
6  echo "Folder Contains:"
7  echo "$folder_contents"
```

Output:

```
Folder Contains:
total 16
-rw-r--r-- 1 mayank mayank  94 Aug 25 07:52 listdir.sh
-rw-r--r-- 1 mayank mayank  93 Aug 25 07:56 listfolder.sh
-rw-r--r-- 1 mayank mayank 104 Aug 25 04:00 readonly.sh
-rw-r--r-- 1 mayank mayank 107 Aug 25 06:59 unset.sh
```

Accessing Values

As shown in the above example we have prefixed the variable with `$` sign in line `7`.

Read-only Variables

We can mark a variable by using `readonly` keyword as shown in below example and any attempt to update it will raise an exception.

```
1  #!/usr/bin/env bash
2
3  username="Demo"
4  readonly username
5  echo $username
6  username="Demo2"
7  echo $username
```

Output:

```
1   ❯ bash readonly.sh
2   Demo
3   readonly.sh: line 6: username: readonly variable
4   Demo
```

In `readonly.sh` we have converted a variable `username` as `readonly` in line `4`. so when we try to change the value of it in line `6`, it returns an error message as shown in line `11`.

`unset` Variables (delete variable)

By using `unset` the variables can be deleted/`unset`

```

#!/usr/bin/env bash
# unset.sh

username="demo"
echo "Welcome ".$username
unset username
echo "Welcome .$username."
```

Output:

```
❯ bash unset.sh
Welcome .demo
Welcome ..
```

In the example, in line `5` we are setting the value `demo` to variable `username` and in line `7` we are deleting the variable. And when we later used in in line `8`, it returns nothing.

Bash variables and data type

Bash variables are similar to python variables, that they can point to different data as and when needed.

```
1  #!/usr/bin/env bash
2  # data_types_and_variables.sh
3  
4  var_one="This is a test"
5  echo $var_one
6  var_one=10
7  echo $var_one
```

Output:

```
❯ bash data_types_and_variables.sh
This is a test
10
```

In the above example, we were able to assign `string` data type to `var_one` (line `4`) and later were able to assign `int` data type (line `6`)

Positional parameters

Arguments passed to the script from the command line:

```
$0, $1, $2, $3 . . .
```

`$0` is the name of the script itself, `$1` is the first argument, `$2` the second, `$3` the third, and so forth

Quoting Variables

When referencing a variable, it is generally advisable to enclose its name in double quotes. This prevents reinterpretation of all special characters within the quoted string -- except $, ` (backquote), and \ (escape). Keeping $ as a special character within double quotes permits referencing a quoted variable ("$variable"), that is, replacing the variable with its value

```
List="one two three"

for a in $List     # Splits the variable in parts at whitespace.
do
```

```
 5    echo "$a"
 6  done
 7  # one
 8  # two
 9  # three
10
11  echo "---"
12
13  for a in "$List"   # Preserves whitespace in a single variable.
14  do #     ^     ^
15    echo "$a"
16  done
17  # one two three
```

Output:

```
$> bash quote_variable.sh
one
two
three
---
one two three
```

In the above example, in line `3`, `for` loop is used with data `$List`, thus it acts as a collection, whereas in line `13` we used the `"$List"`, thus it acted as a string and only one iteration of loop is executed.

Variables by content

Apart from dividing variables in local and global variables, we can also divide them in categories according to content type of the variable.

In this respect, variables come in 4 types:

- String variables
- Integer variables
- Constant variables
- Array variables

Exporting variables

Normally the variables created only available in the shell they are created and thus called `local` variables. These variables will not be visible to either child process not parent process. `Bash` provides a method `export` using which we can make selected variables visible to child processes. These types of variables are also called **environment** variables.

```
export VARNAME="value"
```

Special parameters

Table: Special bash variables

Character	Definition
`$*`	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the `IFS` special variable.
`$@`	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.
`$#`	Expands to the number of positional parameters in decimal.
`$?`	Expands to the exit status of the most recently executed foreground pipeline.
`$-`	A hyphen expands to the current option flags as specified upon invocation, by the **set** built-in command, or those set by the shell itself (such as the `-i`).
`$$`	Expands to the process ID of the shell.
`$!`	Expands to the process ID of the most recently executed background (asynchronous) command.
`$0`	Expands to the name of the shell or shell script.
`$_`	The underscore variable is set at shell startup and contains the absolute file name of the shell or script being executed as passed in the argument list. Subsequently, it expands to the last argument to the previous command, after expansion. It is also set to the full pathname of each command executed and placed in the environment exported to that command.

Exiting running script & its exit status

The `exit` command terminates a script, just as in a **C** program. It can also return a value, which is available to the script's parent process.

```
#!/bin/bash

ls
ls -l /Testing
ls -l /home
# Will exit with status of last command.
exit
```

output:

```
bash exit_code_01.sh
00_variable_error.py         exit_code_01_01.sh       quoting_variables_02.sh
00_variable_error_fixed.py   listdir.sh           readonly.sh
02_export_example.sh         listfolder.sh        surya_namaskar.sh
data_types_and_variables.sh  nonexistingvariables.sh  system_variables.sh
dummy                positional.sh        unset.sh
exit_code_01.sh          quote_variable.sh
ls: cannot access '/Testing': No such file or directory
total 24
drwx------  2 root   root   16384 Oct  2 03:13 lost+found
drwxr-xr-x 77 mayank mayank  4096 Mar 15 18:13 mayank
drwxr-xr-x  3 mjtest mjtest  4096 Mar 15 17:32 mjtest
```

or we can use

```
1  #!/bin/bash
2
3  ls
4  ls -l /Testing
5  ls -l /home
6  # Will exit with status of last command.
7  exit $?
```

output:

```
bash exit_code_01_01.sh
00_variable_error.py         exit_code_01_01.sh       quoting_variables_02.sh
00_variable_error_fixed.py   listdir.sh           readonly.sh
02_export_example.sh         listfolder.sh        surya_namaskar.sh
data_types_and_variables.sh  nonexistingvariables.sh  system_variables.sh
dummy                positional.sh        unset.sh
exit_code_01.sh          quote_variable.sh
ls: cannot access '/Testing': No such file or directory
```

or,We can view the error code of the last code executed by running `echo $?`, either inside the script or at the shell prompt. When we run the `echo $?` at shell prompt, it will return 0.

error codes	Meaning
0	No error.

Now lets remove the code at line 7 and observe the output and exit code.

```
#!/bin/bash

ls
ls -l /Testing
ls -l /home
# Will exit with status of last command.
```

output:

```
00_variable_error.py         exit_code_01_01.sh       quote_variable.sh
00_variable_error_fixed.py   exit_code_01_02.sh       quoting_variables_02.sh
02_export_example.sh         listdir.sh           readonly.sh
data_types_and_variables.sh  listfolder.sh        surya_namaskar.sh
dummy                nonexistingvariables.sh  system_variables.sh
exit_code_01.sh          positional.sh        unset.sh
ls: cannot access '/Testing': No such file or directory
total 24
drwx------  2 root   root   16384 Oct  2 03:13 lost+found
drwxr-xr-x 77 mayank mayank  4096 Mar 15 18:13 mayank
drwxr-xr-x  3 mjtest mjtest  4096 Mar 15 17:32 mjtest
```

Debugging shell script options

verbose Mode (`-v`)

```
> bash -v script.sh
```

We can also add it in the script by using `set` command (`-` (dash) for enabling an option and `+` for disabling)

```
#!/usr/bin/env bash
set -v
```

noexec Mode (`-n`)

We can validate the syntax using `-n` as shown below

```
> bash -n script.sh
```

In this bash will read the shell script but not execute it.

xtrace Mode (`-x`)

Using this option, we can find the flow of code, This mode prints the trace of commands for each line after they are expanded but before they are executed.

```
> bash -x script.sh
```

Finding Unset Variables (`-u`)

This reports any use of `unset` variable.

```
bash -u unset.sh
```

Combining Debugging Options

Most common combining of debugging options are

- combine the *verbose* and *xtrace* mode to get more precise debug information
- combine both -x and -v options to see how statements look like before and after variable substitutions

Debugging Specific Parts of the Script

We previously stated that we can apply the debugging options using `set` command within the script as shown in the below examples

```
v="welcome"
set -x
echo "$v"
unset v
echo "$v"
set +x
echo "$v"
```

Redirecting Only the Debug Output to a File

We can either use `>>` redirection as discussed in `redirection` section, or use the following code snippet to store the output of the script to file

```
exec 5> debug_output.txt
BASH_XTRACEFD="5"
PS4='$LINENO: '
set -x
```

or, if `logger` command is available, then we can write debug output via `syslog` with `timestamp`, `script name` and `line number`:

```
 exec 5> >(logger -t $0)
 BASH_XTRACEFD="5"
 PS4='$LINENO: '
 set -x
```

You can use option `-p` of `logger` command to set an individual facility and level to write output via local syslog to its own logfile.

Using `trap`

```
#! /bin/bash
trap 'echo "Line- ${LINENO}: five_val=${five_val}, two_val=${two_val},
total=${total}" ' DEBUG
five_val=5
two_val=2
total=$((five_val+two_val))
echo "Total is: $total"
total=0 && echo "Resetting Total"
```

In this example, we specified the *echo* command to print the values of variables *five_val*, *two_val*, and *total.* Subsequently, we passed this echo statement to the *trap* command with the *DEBUG* signal. In effect, prior to the execution of every command in the script, the values of variables get printed.

Using external logging/debugging utilities

- `shellcheck`: It is an external command which can be downloaded from https://github.com/koalaman/shellcheck#user-content-installing
- `bashdb`: can be downloaded from http://bashdb.sourceforge.net, its a source-code debugger for bash that follows the `gdb` command syntax.
- `log4bash`: https://github.com/fredpalmer/log4bash, can be used for logging similar to other log4* loggers.

Summary

Table: Overview of set debugging options

Short notation	Long notation	Result
set -f	set -o noglob	Disable file name generation using metacharacters (globbing).
set -v	set -o verbose	Prints shell input lines as they are read.
set -x	set -o xtrace	Print command traces before executing command.

More Bash options

Displaying options (`-o`)

If we wish the find all the options applied to the script, we can use `-o` as shown in the below example

```
❯ cat display.sh
set -vu
echo "welcome"
set -o
```

```

❯ bash display.sh
echo "welcome"
welcome
set -o
allexport          off
braceexpand        on
emacs              off
errexit            off
errtrace           off
functrace          off
hashall            on
histexpand         off
history            off
ignoreeof          off
interactive-comments    on
keyword            off
monitor            off
noclobber          off
noexec             off
noglob             off
nolog              off
notify             off
nounset            on
onecmd             off
physical           off
pipefail           off
posix              off
privileged         off
verbose            on
vi                 off
xtrace             off
```

Changing options

We can change the above displayed option as shown in the below example (- (dash) for enabling an option, + for disabling)

```
set -o functrace
```

Exercise

- How to debug bash script by enabling verbose “-v” option
- How to debug bash script using xtrace “-x” option
- How to debug bash script using noexec “-n” option
- How to identify the unset variables while debugging bash script
- How to debug the specific part of the bash script
- How to debug a bash script using the “trap” command

- How to debug a bash script by eliminating file globbing using the “-f” option
- How to combine debugging options to debug a shell script
- How to redirect debug-report to a file

Conditional statements

There are instances where we need to validate some condition, and based on its output need to execute a piece of code. Bash provides us few control statements

`if`

if is very useful in executing a group of code if a condition is `True`

Syntax:

```
if <test-condition(s)> ; then <commands> fi
```

or, better to use the following syntax for better readability

```
if <test-condition(s)> ; then
    <commands>
fi

if [ <test-condition(s)> ] ; then
    <commands>
fi
```

or, the below one

```
if <test-condition(s)>
then
    <commands>
fi
```

`TEST-COMMAND` is executed, and if its return status is `zero`, then the `commands` list is executed else they are skipped. `if` conditional statement return either the the exit code of the last executed command or `zero` if condition tested `false`.

Basic Rules

- Always keep spaces between the brackets and the actual check/comparison, else it will fail as shown below

```
set -eou pipefail nounset                    # Treat unset variables as an error

name="Mayank"
username="Mayank"

if [$name == $username ]; then
    echo "Both the names are same"
fi
```

```

if [ $name == $username]; then
    echo "Both the names are same"
fi

# Only this will work
if [ $name == $username ]; then
    echo "Both the names are same"
fi
```

Output:

```
➜  bash 01_bracket_err.sh
01_bracket_err.sh: line 25: [Mayank: command not found
01_bracket_err.sh: line 29: [: missing `]'
Both the names are same
```

We can avoid this error by adding a space between the condition and `[` `]` brackets.

- Terminate the line before putting a new keyword like "then" using `;` as shown below

```
if [name == username ]; then
```

or move it to new line. Please note that above syntax is suggested to be used or move it to new line. Please note that above syntax is suggested to be used a bash it is more concise or move it to new line. Please note that above syntax is suggested to be used a bash it is more concise

```
if [name == username ]
then
```

Otherwise we will experience below error

code with issue:

```
name="Mayank"
username="Mayank"
if [ $name == $username ]
then
    echo "Both the names are same"
fi

if [ $name == $username ]; then
    echo "Both the names are same"
fi

if [ $name == $username ] then
    echo "Both the names are same"
fi
```

Output:

```
✗  bash 02_err_newline.sh
Both the names are same
Both the names are same
02_err_newline.sh: line 41: syntax error near unexpected token `fi'
```

Good Idea

- Putting an "!" in front of a condition inverts it, also `!` should be placed **inside** the `[]` brackets and not outside.
- variables in a condition should **always** be encapsulated by `""`

`test-condition(s)`

Bash provides many options for test commands. Most common of conditions checked are

- numerical or string comparison
- bash commands which can return zero on success and other on fail
- Linux commands output
- Unary expression for checking file/folder/device status

Primary expressions for `test-condition(s)`

Table: File expressions

Expression	Meaning
[`-a` `FILE`]	True if `FILE` exists.
[`-b` `FILE`]	True if `FILE` exists and is a block-special file.
[`-c` `FILE`]	True if `FILE` exists and is a character-special file.
[`-d` `FILE`]	True if `FILE` exists and is a directory.
[`-e` `FILE`]	True if `FILE` exists.
[`-f` `FILE`]	True if `FILE` exists and is a regular file.
[`-g` `FILE`]	True if `FILE` exists and its SGID bit is set.
[`-h` `FILE`]	True if `FILE` exists and is a symbolic link.
[`-k` `FILE`]	True if `FILE` exists and its sticky bit is set.
[`-p` `FILE`]	True if `FILE` exists and is a named pipe (FIFO).
[`-r` `FILE`]	True if `FILE` exists and is readable.
[`-s` `FILE`]	True if `FILE` exists and has a size greater than zero.
[`-t` `FD`]	True if file descriptor `FD` is open and refers to a terminal.

Expression	Meaning
[`-u` `FILE`]	True if `FILE` exists and its SUID (set user ID) bit is set.
[`-w` `FILE`]	True if `FILE` exists and is writable.
[`-x` `FILE`]	True if `FILE` exists and is executable.
[`-O` `FILE`]	True if `FILE` exists and is owned by the effective user ID.
[`-G` `FILE`]	True if `FILE` exists and is owned by the effective group ID.
[`-L` `FILE`]	True if `FILE` exists and is a symbolic link.
[`-N` `FILE`]	True if `FILE` exists and has been modified since it was last read.
[`-S` `FILE`]	True if `FILE` exists and is a socket.
[`FILE1` `-nt` `FILE2`]	True if `FILE1` has been changed more recently than `FILE2`, or if `FILE1` exists and `FILE2` does not.
[`FILE1` `-ot` `FILE2`]	True if `FILE1` is older than `FILE2`, or is `FILE2` exists and `FILE1` does not.
[`FILE1` `-ef` `FILE2`]	True if `FILE1` and `FILE2` refer to the same device and inode numbers.

Table : **String Expression**

Expression	Meaning
`[ o OPTIONNAME ]`	True if shell option "OPTIONNAME" is enabled.
`[ -z STRING ]`	True of the length if "STRING" is zero.
`[ -n STRING ] or [ STRING ]`	True if the length of "STRING" is non-zero.
`[ STRING1 == STRING2 ]`	True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.
`[ STRING1 != STRING2 ]`	True if the strings are not equal.

Expression	Meaning
`[ STRING1 < STRING2 ]`	True if "STRING1" sorts before "STRING2" lexicographically in the current locale.
`[ STRING1 > STRING2 ]`	True if "STRING1" sorts after "STRING2" lexicographically in the current locale.
`[ ARG1 OP ARG2 ]`	"OP" is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt` or `-ge`. These arithmetic binary operators return true if "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively. "ARG1" and "ARG2" are integers.

Combining Expressions

multiple Test conditions can be joined using one or more of the following expressions.

Table: **Combining expressions**

Operation	Effect
`[ ! EXPR ]`	True if **EXPR** is false.
`[ ( EXPR ) ]`	Returns the value of **EXPR**. This may be used to override the normal precedence of operators.
`[ EXPR1 -a EXPR2 ]`	True if both **EXPR1** and **EXPR2** are true.
`[ EXPR1 -o EXPR2 ]`	True if either **EXPR1** or **EXPR2** is true.

Examples

- Checking if file exist

```
 cat example1.sh
#!/usr/bin/env bash
# GNU 3 License

echo "This scripts validates if /etc/os-release is present"
echo "Checking..."
file_name="/etc/os-release"

if [ -f $file_name ]; then
    echo "1. $file_name exists."
fi
echo "...done."
```

```

file_name="/etc/os-release-old"
if [ -f $file_name ]; then
    echo "2. $file_name exists."
fi
echo "...done."
```

Output:

```
➜  bash example1.sh
This scripts validates if /etc/os-release is present
Checking...
1. /etc/os-release exists.
...done.
...done.
```

Brackets

BASH supports both single bracket `[ ]` and `[[ ]]`. Although on initial usage they both look similar, but they both have few major differences.

Single Bracket

Single bracket `[` is actually an `alias` for the `test` command, it's not syntax in other words is a **POSIX compliant** *__command__* and will have the same functionality as the `condition`. Now lets see what that means using few examples.

```
num=10

if [ $num -gt 1 ]; then
    echo "Number is greater"
fi
```

Output:

```
➜  bash 002_if_with_numbers.sh
Number is greater
```

Now, lets delete the variable `num` using `unset num`and try again

```
num=10

unset num
if [ $num -gt 1 ]; then
    echo "Number is greater"
fi
```

Output:

```
003_if_with_numbers.sh: line 4: [: -gt: unary operator expected
```

as you can see that it was not able to handle a scenario where variable was non existing. We can avoid this case by encapsulating the variables in `""` as shown in below example.

```
num=10

unset num
# Treating int values as strings.
# Not a good solution, use double brackets as
# permanent solution
# We can also use `set -o nounset` to avoid these
# cases in more orderly manner.
if [ "$num" = "1" ]; then
    echo "Number is greater"
fi
```

Double Bracket

Double Bracket `[[ ]]`, on the other hand, is *syntax* and is more capable than `[ ]`. On the other hand it's **not POSIX** compliant and its also not a command. Thus will only work on handful of shells, such as `bash` and `ksh`.

```
name="Mayank"
username="Mayank"

# This one will pass
if [[ $name = $username ]]; then
    echo "1. Both the names are same"
fi

unset name
if [[ $name = $username ]]; then
    echo "2. Both the names are same"
fi
```

output:

```
➜  bash double.sh
1. Both the names are same
```

final solution

Use `set -o nounset` in the script to get proper error message as shown below, i have added `xv` for more details logging.

```
set -xvo nounset

```

```

name="Mayank"
username="Mayank"

# This one will pass
if [ "$name" = "$username" ]; then
    echo "Both the names are same"
fi

# This one will pass
if [ "$name" == "$username" ]; then
    echo "Both the names are same"
fi


unset name
if [ "$name" = "$username" ]; then
    echo "Both the names are same"
fi
```

output:

```
name="Mayank"
+ name=Mayank
username="Mayank"
+ username=Mayank

# This one will pass
if [ "$name" = "$username" ]; then
    echo "Both the names are same"
fi
+ '[' Mayank = Mayank ']'
+ echo 'Both the names are same'
Both the names are same

# This one will pass
if [ "$name" == "$username" ]; then
    echo "Both the names are same"
fi
+ '[' Mayank == Mayank ']'
+ echo 'Both the names are same'
Both the names are same


unset name
+ unset name
if [ "$name" = "$username" ]; then
    echo "Both the names are same"
fi
001_single.sh: line 38: name: unbound variable
```

`if`-`else`

Although `if` statement is very useful, but not very practical, as most of the time we have scenarios where we have to execute different sets of instructions depending upon the condition return code (`0` or not zero).

```
if [ <test-condition(s)> ] ; then
    <commands>
else
    <commands>
fi
```

Example:

```
set -o nounset                              # Treat unset variables as an error

cost=100
sale_amount=210

# In this the `else` code block will run
if [ $cost -ge $sale_amount ]; then
    echo "Sorry we are in loss"
else
    echo "Yeppy, we are in profit"
fi

# In this the `if` code block will run
if [ $cost -lt $sale_amount ]; then
    echo "Yeppy, we are in profit"
else
    echo "Sorry we are in loss"
fi
```

Output:

```
➜  bash -x 001_basic.sh
+ set -o nounset
+ cost=100
+ sale_amount=210
+ '[' 100 -ge 210 ']'
+ echo 'Yeppy, we are in profit'
Yeppy, we are in profit
+ '[' 100 -lt 210 ']'
+ echo 'Yeppy, we are in profit'
Yeppy, we are in profit
```

Another example from "Bash Beginners Guide"

```
set -xo nounset                              # Treat unset variables as an error

echo "This script does a very simple test for checking disk space."

space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut
-d "%" -f1 -`
alertvalue="80"

if [ "$space" -ge "$alertvalue" ]; then
  echo "At least one of my disks is nearly full!" | mail -s "daily diskcheck"
root
else
  echo "Disk space normal" | mail -s "daily diskcheck" root
fi
```

Output:

```
➜  bash disk_test.sh
+ echo 'This script does a very simple test for checking disk space.'
This script does a very simple test for checking disk space.
++ df -h
++ tail -1
++ sort -n
++ grep %
++ cut -d % -f1 -
++ grep -v Use
++ awk '{print $5}'
+ space=19
+ alertvalue=80
+ '[' 19 -ge 80 ']'
+ echo 'Disk space normal'
+ mail -s 'daily diskcheck' root
```

Bash also support nested `if`/`if-else` statements

`if/then/elif/else` statements

We have scenarios where, we have to check multiple conditions in top to bottom approach and run the respective code block for which the condition is fulfilled, otherwise run the `else` code block. Lets view its usage in below example.

```
set -eou  nounset                        # Treat unset variables as an error

temp=39

if [[ $temp -le 0 ]]; then
    echo "Its freezing, lets stay at home"
elif [[ $temp -le 10 ]]; then
    echo "Its very cold, only in emergency leave home"
elif [[ $temp -le 25 ]]; then
```

```bash
    echo "Its good weather, lets visit park"
elif [[ $temp -le 35 ]]; then
    echo "Its starting to get hot"
elif [[ $temp -le 40 ]]; then
    echo "Its very hot, only in emergency leave home"
elif [[ $temp -ge 40 ]]; then
    echo "Its very hot, dont leave home"
else
    echo "Sorry, I am not able to understand the temprature provided"
fi
```

Please run the above script against multiple values of `temp` value and check the response, some suggested values are 0, 12, 8, 24, 44, 32

output: for various values of `temp`

```
➜  bash greeting_according_to_temp.sh
Its 32°C
Its starting to get  hot

➜  bash greeting_according_to_temp.sh
Its 22°C
Its good weather, lets visit park

➜  bash greeting_according_to_temp.sh
Its 59°C
Its very hot, dont leave home

➜  bash greeting_according_to_temp.sh
Its 39°C
Its very hot, only in emergency leave home
```

Table: Most frequently used operators

Operators	Meaning
`-n VAR`	True if the length of VAR is greater than zero.
`-z VAR`	True if the VAR is empty.
`STRING1 = STRING2`	True if STRING1 and STRING2 are equal.
`STRING1 != STRING2`	True if STRING1 and STRING2 are not equal.
`INTEGER1 -eq INTEGER2`	True if INTEGER1 and INTEGER2 are equal.
`INTEGER1 -gt INTEGER2`	True if INTEGER1 is greater than INTEGER2.

Operators	Meaning
`INTEGER1 -lt INTEGER2`	True if INTEGER1 is less than INTEGER2.
`INTEGER1 -ge INTEGER2`	True if INTEGER1 is equal or greater than INTEGER2.
`INTEGER1 -le INTEGER2`	True if INTEGER1 is equal or less than INTEGER2.
`-h FILE`	True if the FILE exists and is a symbolic link.
`-r FILE`	True if the FILE exists and is readable.
`-w FILE`	True if the FILE exists and is writable.
`-x FILE`	True if the FILE exists and is executable.
`-d FILE`	True if the FILE exists and is a directory.
`-e FILE`	True if the FILE exists and is a file, regardless of type (node, directory, socket, etc.).
`-f FILE`	True if the FILE exists and is a regular file (not a directory or device).

`case` statement

`case` statement can be used to simplify complicated `if/else` statements, or to have case of running code blocks against unique patterns.

Syntax

```
case EXPRESSION in

  PATTERN_1 | PATTERN_2)
    STATEMENTS
    ;;

  PATTERN_2)
    STATEMENTS
    ;;

  PATTERN_N)
    STATEMENTS
    ;;

  *)
    STATEMENTS
    ;;
esac
```

The `EXPRESSION` is evaluated and matched against the *PATTERNs* and if none of them matches then if provided code block for `*` is executed.

Examples

```
set -o nounset                              # Treat unset variables as an error

what="boy"

case $what in
    boy)
        msg="Ja, Ich bin ein Junge"
        ;;
    girl)
        msg="Ja, Ich bin eine Mädchen"
        ;;
    *)
        msg="Sorry not able to understand"
esac
echo $msg
```

Output:

```
➜  bash 001_basic.sh
Ja, Ich bin ein Junge
```

Example: License Agreement

```
set -eou  nounset                     # Treat unset variables as an error

echo "Do you agree to the GNU3 License terms"
read resp

case $resp in
    [yY] | [yY][eE][sS] )
        msg="Thanks a lot"
        ;;
    [nN] | [nN][oO] )
        msg="Sorry for it, but in that case you cannot run the program"
        ;;
    *)
        msg="Invalid response"
        ;;
esac
echo $msg
```

Output:

```
➜  bash check_resp.sh
Do you agree to the GNU3 License terms
```

```
y
Thanks a lot

➜  bash check_resp.sh
Do you agree to the GNU3 License terms
    n
Sorry for it, but in that case you cannot run the program

➜  bash check_resp.sh
Do you agree to the GNU3 License terms
Tes
Invalid response

➜  bash check_resp.sh
Do you agree to the GNU3 License terms
Mo
Invalid response

➜  bash check_resp.sh
Do you agree to the GNU3 License terms
NO
Sorry for it, but in that case you cannot run the program
```

Example: Another version

```
set -eou  nounset                    # Treat unset variables as an error

echo "Do you agree to the GNU3 License terms"
read resp

# Convert everything to lower case
resp=$( echo "$resp" | tr -s  '[:upper:]'  '[:lower:]' )

case $resp in
    y | yes)
        msg="Thanks a lot"
        ;;
    n | no )
        msg="Sorry for it, but in that case you cannot run the program"
        ;;
    *)
        msg="Invalid response"
        ;;
esac
echo $msg
```

Output:

```
➜  bash 002_check_resp.sh
Do you agree to the GNU3 License terms
G
```

```
Invalid response

➜  bash 002_check_resp.sh
Do you agree to the GNU3 License terms
YeS
Thanks a lot

➜  bash 002_check_resp.sh
Do you agree to the GNU3 License terms
yes
Thanks a lot

➜  bash 002_check_resp.sh
Do you agree to the GNU3 License terms
nO
Sorry for it, but in that case you cannot run the program
```

Example: Guess the number Game

```
# Version 0.0.1
set -o nounset                              # Treat unset variables as an error

echo "Lets play a game of gussing the number"

# We can use the below one also
# num=`shuf -i 1-5 -n 1`
# You can increase the numbers by replacing 5 with the number you want
num=$(( ( RANDOM % 5 ) + 1 ))

echo "Please enter your guss from (0-5): "
read gussed_num

case $gussed_num in
    $num)
        echo "You gussed the right number"
        ;;
    *)
        echo "Sorry, the correct number was $num"
        ;;
esac
```

Variables Continued

Till now we have only coverted variables which store only one value, in this section we will learn about variables which can store more than one value.

`declare` statement

We already saw that same variable can be used for different data types, it can be confusing sometimes. Bash provides a method to limit the scope

Syntax:

```
declare OPTION(s) VARIABLE=value
```

Table: Options to the declare built-in

Option	Description
-f	All names are treated as the names of functions, not variables.
-F	When displaying information about a function, display only the function's name and attributes. Do not display the contents of the function.
-g	When **declare** is used within a shell function, the **-g** option causes all operations on variables to take effect in global scope. If not used in a shell function, **-g** has no effect.
-p	Display the attributes and values of each variable. Please note that when `-p` is used all other additional options are ignored

Table: Variable attribute modifying attributes

Option	Meaning
`-a`	Variable is an array.
`-i`	The variable is to be treated as an integer
`-r`	Make variables read-only. These variables cannot then be assigned values by subsequent assignment statements, nor can they be unset.
`-t`	Give each variable the *trace* attribute (Still await for its complete implementation ;)).
`-x`	Mark each variable for export to subsequent commands via the environment.

> - Using `+` instead of `-` turns off the attribute instead
> - When used in a function, declare creates local variables

> - Command `typeset` is an alias for `declare`

Lexical scope

When **declare** is used inside a shell function, all named items are declared in a local scope, unless the **-g** option is used. This behaviour is the same as using the **local** builtin command.

Exit status

The exit status of **declare** is success (zero), unless an invalid option is specified or an error occurs, in which case the status is failure (non-zero).

Options in details

(`-p`) Display the variable attributes & value

Examples

```
set -o nounset                                # Treat unset variables as an error

user="Rakesh"
Rakesh="Ramesh"

# Will list all the variables.
declare -p
echo "------------------"
# Will details about variable user
declare -p user
echo "------------------"
declare -p "$user"    # declare -p Rakesh
```

Output:

```
➜  bash 001_declare.sh
declare -x ANT_HOME="/usr/share/ant"
declare -- BASH="/bin/bash"
declare -r
. . . . . . . . . . . . . .
declare -- _=""
declare -- user="Rakesh"
------------------
declare -- user="Rakesh"
------------------
declare -- Rakesh="Ramesh"
```

(`-r`) "read-only" variable

We cannot `unset` a variable declared using `-r` as shown in below example

```
user_name="Mayank"
declare -r username="Rahul"

echo $user_name
echo $username

unset user_name
unset username

echo $user_name
# This will still print
echo $username
```

Output:

```
 ➜  bash 003_readonly.sh
Mayank
Rahul
003_readonly.sh: line 9: unset: username: cannot unset: readonly variable

Rahul
```

(`-i`) Integer variables

Example

```
user="Rakesh"

rakesh=100
declare -i user=190
echo $user
echo $rakesh
echo "------------------"
declare -i rakesh=222
echo $rakesh
user="Mohan"   # It will not get Mohan value instead 0
rakesh=2
echo $user
echo $rakesh
```

Output

```
➜  bash 002.1_int.sh
190
100
------------------
222
0
2
```

Example 2: Evaluating data before assigning

```
declare -i a=100
declare -i b=$a*100
echo $a
echo $b
```

Output:

```
100
10000
```

Example 3: Declare and then later assigning value

```
declare -i data
data="20*100"
echo $data
```

Output:

```
2000
```

Example 4:

```
declare -i data

data="22/3"
echo $data
data="24/5"
echo $data
echo "---------------"
data="10.1/3"
echo $data
```

Output:

```
➜  bash 002_004_int.sh
7
4
---------------
002_004_int.sh: line 29: 10.1/3: syntax error: invalid arithmetic operator (error token is ".1/3")
4
```

(`-a`) Arrays

```
# set -o nounset
declare -a capitals

# This will return nothing
echo ${capitals}
capitals[0]="New Delhi"
capitals[1]="Hyderabad"
capitals[2]="Chennai"
echo ${capitals[2]}

# This will raise an error
declare +a capitals
```

Output:

```
Chennai
005_01_array.sh: line 31: declare: capitals: cannot destroy array variables in this way
```

Examples

Example: Check if variable exist.

```
set -o nounset

if [ -z ${capitals+x} ]; then
    echo "variable not set";
else
    echo "variable set";
fi
```

Output:

```
variable not set
```

References

- https://lists.gnu.org/archive/html/info-gnu/2002-07/msg00005.html
- https://unix.stackexchange.com/questions/524132/what-is-the-use-of-declare-with-option-t

Bash Arrays

Bash supports single dimension array as shown in previous examples. Array may be initialised using `var[indx]` notation. Or we can explicitly create an array as shown in the previous chapter using `declare -a`.

Creating Your First Array

Bash allows to create arrays using two ways. First we can directly assign value to a new variable as shown in the below example

```
set -eou pipefail nounset

capital[0]="New Delhi"
echo $capital
```

Output:

```
New Delhi
```

or, we can use `declare -a` as shown in previous chapter.

We can even populate the list while declaring the variable as shown in below example

```
set -eou pipefail nounset

capital=("New Delhi" "Bhopal" "Lucknow" "Chennai")
echo $capital
```

Accessing Array Elements

We can use `var[<indx>]` syntax to access the elements of the variables

```
declare -a capitals

# This will return nothing
echo ${capitals}
capitals[0]="New Delhi"
capitals[1]="Lucknow"
capitals[2]="Bhopal"
echo ${capitals[2]}
```

Output

```
Bhopal
```

Finding Length of elements

We can use `#` to find the length of array elements as shown in the below example

Gotchas

```
declare -a capitals

# This will return nothing
echo ${capitals}
capitals[0]="Delhi"
capitals[1]="Lucknow"
capitals[2]="Bhopal"
echo ${capitals[2]}
echo "Total numbers of elements in capitals is ${#capitals}"

# This is another way we can create array
capitals=( [0]="New Delhi" [1]="Bhopal")

# The below code will return the length of the first element
# which in our case is "New Delhi"
echo "Total numbers of elements in capitals is ${#capitals}"
echo ${capitals[2]}
echo ${capitals[1]}

```

Output:

```
➜  bash 005_03_find_length_array.sh

Bhopal
Total numbers of elements in capitals is 5
Total numbers of elements in capitals is 9

Bhopal
```

Example:

```
declare -a capitals

# This will return nothing
echo ${capitals}
capitals[0]="New Delhi"
capitals[1]="Lucknow"
capitals[2]="Bhopal"
echo ${capitals[2]}
echo ${capitals[@]:0}
echo "Total numbers of elements in capitals is ${#capitals[@]}"

capitals=( [0]="New Delhi" [1]="Bhopal")
echo "Total numbers of elements in capitals is ${#capitals[@]}"
echo ${capitals[2]}
echo ${capitals[1]}
echo ${capitals[@]:0}
```

associative array

Associative arrays are similar to `dictionary` in python. They can be used to store `key\value` pairs and are declared using `-A` option as shown below

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

declare -A capitals=(
    [India]="New Delhi"
    [Japan]="Tokyo"
    [Afghanistan]="Kabul"
    [England]="London"
    [Ireland]="Dublin"
    [Nepal]="Kathmandu"
)

declare -A selected=()

# Both will display the values only
echo ${capitals[@]}
echo ${capitals[*]}


# Both will display the keys only
echo ${!capitals[*]}
echo ${!capitals[@]}

for key in "${!capitals[@]}"
do
    echo "Capital of ${key} is ${capitals[${key}]}."
done


echo "~~~~~~~~~~~~~~~~~~~~~"
echo "Using * in for loop will not work"
for key in "${!capitals[*]}"
do
    echo "Capital of ${key} is ${capitals[${key}]}."
done

```

Update/Add elements

We can update the associative arrays as shown in the below code, it will either update or create the value depending on if key already exist.

```
capitals[Russia]="Mascow"
selected[Trinidad and Tobago]="Port of Spain"
```

Output:

```
→ bash check_associate.sh
Russia is missing
\nLets add Russia
Russia present
```

`readonly` Associate

We can also have `readonly` associate using `readonly` or `-r` as shown below

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

declare -r -A capitals=(
    [India]="New Delhi"
    [Japan]="Tokyo"
    [Afghanistan]="Kabul"
    [England]="London"
    [Ireland]="Dublin"
    [Nepal]="Kathmandu"
)

echo "\nLets add Russia"
capitals[Russia]="Mascow"
```

Output:

```
→ bash readonly_associate.sh
\nLets add Russia
readonly_associate.sh: line 14: capitals: readonly variable

```

Validate if element is present

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

declare -A capitals=(
    [India]="New Delhi"
    [Japan]="Tokyo"
```

```bash
    [Afghanistan]="Kabul"
    [England]="London"
    [Ireland]="Dublin"
    [Nepal]="Kathmandu"
)

if [[ -n "${capitals[Russia]}" ]]
then
    echo "Russia present"
else
    echo "Russia is missing"
fi

echo "Lets add Russia"
capitals[Russia]="Mascow"


if [[ -n "${capitals[Russia]}" ]]
then
    echo "Russia present"
else
    echo "Russia is missing"
fi

```

Output:

```
→ bash readonly_associate.sh
Lets add Russia
readonly_associate.sh: line 14: capitals: readonly variable
```

Remove elements

We can use `unset` to remove an element as shown in below example

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

declare -A capitals=(
    [India]="New Delhi"
    [Japan]="Tokyo"
    [Afghanistan]="Kabul"
    [England]="London"
    [Ireland]="Dublin"
    [Nepal]="Kathmandu"
)

unset capitals[England]

for key in "${!capitals[@]}"
do
    echo "Capital of ${key} is ${capitals[${key}]}."
```

```
done
```

Output:

```
→ bash remove_associate.sh
Capital of Afghanistan is Kabul.
Capital of Ireland is Dublin.
Capital of India is New Delhi.
Capital of Japan is Tokyo.
Capital of Nepal is Kathmandu.

```

Length of Array

We can use `${#capitals[@]}` to find the length of `capitals` array

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

declare -A capitals=(
    [India]="New Delhi"
    [Japan]="Tokyo"
    [Afghanistan]="Kabul"
    [England]="London"
    [Ireland]="Dublin"
    [Nepal]="Kathmandu"
)


echo "${#capitals[@]} elements found"

unset capitals[England]

echo "${#capitals[@]} elements found"


for key in "${!capitals[@]}"
do
    echo "Capital of ${key} is ${capitals[${key}]}."
done
```

Output:

```
6 elements found
5 elements found
Capital of Afghanistan is Kabul.
Capital of Ireland is Dublin.
Capital of India is New Delhi.
Capital of Japan is Tokyo.
Capital of Nepal is Kathmandu.

```

Shuffle elements of an Array

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

declare -A capitals=(
    [India]="New Delhi"
    [Japan]="Tokyo"
    [Afghanistan]="Kabul"
    [England]="London"
    [Ireland]="Dublin"
    [Nepal]="Kathmandu"
)

for key in "${!capitals[@]}"
do
    echo "Capital of ${key} is ${capitals[${key}]}."
done

echo "Lets shuffle ..."

shuf -e "${capitals[*]}"


for key in "${!capitals[@]}"
do
    echo "Capital of ${key} is ${capitals[${key}]}."
done
```

Output:

```
|→ bash shuf_associate.sh
Capital of Afghanistan is Kabul.
Capital of England is London.
Capital of Ireland is Dublin.
Capital of India is New Delhi.
Capital of Japan is Tokyo.
Capital of Nepal is Kathmandu.
Lets shuffle ...
Kabul London Dublin New Delhi Tokyo Kathmandu
Capital of Afghanistan is Kabul.
Capital of England is London.
Capital of Ireland is Dublin.
Capital of India is New Delhi.
Capital of Japan is Tokyo.
Capital of Nepal is Kathmandu.
```

Subset of an Array

We can have subset (slice) of arrays using the following syntax

```
echo "${users[@]:2:5}"
#                  | |
#                  | └> Length of slice (Optional)
#                  └──> Starting index of slice
```

Example

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

declare -a capitals=(
    "New Delhi"
    "Tokyo"
    "Kabul"
    "London"
    "Dublin"
    "Kathmandu"
)

echo ">>"
echo "${capitals[@]:2:3}"

for key in "${capitals[@]:2:3} "
do
    echo "Capital is ${key} "
done

# It will print all the elements from 2nd index
for key in "${capitals[@]:2} "
do
    echo "Capital is ${key} "
done
```

Output:

```
→ bash subset_array.sh
>>
Kabul London Dublin
Capital is Kabul
Capital is London
Capital is Dublin
Capital is Kabul
Capital is London
Capital is Dublin
Capital is Kathmandu
```

Copying Arrays

```
declare -a capitals=(
    "Tokyo"
    "Kabul"
    "London"
    "Dublin"
    "Kathmandu"
    "New Delhi"
)

echo ""
echo "Lets copy using slicing"
echo ""
selected=("${capitals[@]:2:2}")
for key in "${selected[@]} "
do
    echo "Capital is ${key} "
done
```

Output:

```
Lets copy using slicing

Capital is London
Capital is Dublin

```

Concatenating arrays

We can use many ways to concatenate the arrays, one of my favorite is as follows, in which we are adding arrays `capitasls_asia` and `capitals_africa` in `capitals`

```
capitals+=( "${capitals_asia[@]}" "${capitals_africa[@]}" )
```

More fun with arrays

Example: From ABS

```
#!/bin/bash
# array-strops.sh: String operations on arrays.
# URL: https://tldp.org/LDP/abs/html/arrays.html
# Script by Michael Zick.
# Used in ABS Guide with permission.
# Fixups: 05 May 08, 04 Aug 08.

#  In general, any string operation using the ${name ... } notation
#+ can be applied to all string elements in an array,
#+ with the ${name[@] ... } or ${name[*] ...} notation.
```

```


arrayZ=( one two three four five five )

echo

# Trailing Substring Extraction
echo ${arrayZ[@]:0}     # one two three four five five
#                ^        All elements.

echo ${arrayZ[@]:1}     # two three four five five
#                ^        All elements following element[0].

echo ${arrayZ[@]:1:2}   # two three
#                  ^      Only the two elements after element[0].

echo "---------"


# Substring Removal

# Removes shortest match from front of string(s).

echo ${arrayZ[@]#f*r}   # one two three five five
#               ^       # Applied to all elements of the array.
                        # Matches "four" and removes it.

# Longest match from front of string(s)
echo ${arrayZ[@]##t*e}  # one two four five five
#               ^^      # Applied to all elements of the array.
                        # Matches "three" and removes it.

# Shortest match from back of string(s)
echo ${arrayZ[@]%h*e}   # one two t four five five
#               ^       # Applied to all elements of the array.
                        # Matches "hree" and removes it.

# Longest match from back of string(s)
echo ${arrayZ[@]%%t*e}  # one two four five five
#               ^^      # Applied to all elements of the array.
                        # Matches "three" and removes it.

echo "----------------------"


# Substring Replacement

# Replace first occurrence of substring with replacement.
echo ${arrayZ[@]/fiv/XYZ}   # one two three four XYZe XYZe
#               ^           # Applied to all elements of the array.

# Replace all occurrences of substring.
```

```
echo ${arrayZ[@]//iv/YY}    # one two three four fYYe fYYe
                            # Applied to all elements of the array.

# Delete all occurrences of substring.
# Not specifing a replacement defaults to 'delete' ...
echo ${arrayZ[@]//fi/}      # one two three four ve ve
#               ^^          # Applied to all elements of the array.

# Replace front-end occurrences of substring.
echo ${arrayZ[@]/#fi/XY}    # one two three four XYve XYve
#                ^          # Applied to all elements of the array.

# Replace back-end occurrences of substring.
echo ${arrayZ[@]/%ve/ZZ}    # one two three four fiZZ fiZZ
#                ^          # Applied to all elements of the array.

echo ${arrayZ[@]/%o/XX}     # one twXX three four five five
#                ^          # Why?

echo "-----------------------------"


replacement() {
    echo -n "!!!"
}

echo ${arrayZ[@]/%e/$(replacement)}
#                ^  ^^^^^^^^^^^^^^
# on!!! two thre!!! four fiv!!! fiv!!!
# The stdout of replacement() is the replacement string.
# Q.E.D: The replacement action is, in effect, an 'assignment.'

echo "------------------------------------"

#  Accessing the "for-each":
echo ${arrayZ[@]//*/$(replacement optional_arguments)}
#                ^^ ^^^^^^^^^^^^^
# !!! !!! !!! !!! !!! !!!

#  Now, if Bash would only pass the matched string
#+ to the function being called . . .

echo

exit 0

#  Before reaching for a Big Hammer -- Perl, Python, or all the rest --
#  recall:
#    $( ... ) is command substitution.
#    A function runs as a sub-process.
#    A function writes its output (if echo-ed) to stdout.
#    Assignment, in conjunction with "echo" and command substitution,
#+   can read a function's stdout.
#    The name[@] notation specifies (the equivalent of) a "for-each"
#+   operation.
#  Bash is more powerful than you think!
```

Summary on Arrays

```
${uers[@]}         # Returns all indizes and their items (except associative arrays)
${#uers[$n]}       # Length of $nth item
${uers[*]}         # Returns all items
${!uers[*]}        # Returns all indizes
${#uers[*]}        # Number elements
```

Basic Arithmetic Operators

(`=`) assignment

It helps in either *initialising* or *updating* the value of the variable

```
user="Shashant"
echo $user
user="Rishabh"
echo $user
```

Arithmetic Operators

Basic Operators

Operator	Name	Meaning
`+`	Plus	Adds two items
`-`	minus	Subtract
`*`	Multiplication	Multiplication
`/`	Division	
`%`	modulo, or mod	returns the remainder of an integer division operation
`**`	Exponent	

- **Examples**:

```
mango=100
apple=23

echo $(( apple + mango ))
echo $(( apple - mango ))
echo $(( apple * mango ))
echo $(( apple % mango ))
echo $(( apple / mango ))
echo $(( apple**3  ))
```

Output:

```
450
```

- **Example**:

```
a=100
b=350

echo $(( $a-$b ))
```

Output:

```

```

- **Example**:

```
a=100
b=350

echo $(( $a*$b ))
```

Output:

```

```

- **Example**:

```
a=100
b=350

echo $(( $a/$b ))
```

Output:

```

```

- **Example**:

  ```
  a=100
  b=350

  echo $(( $a**$b ))
  ```

  Output:

  ```

  ```

- **Example**:

  ```
  a=100
  b=350

  echo $(( $a%$b ))
  ```

  Output:

  ```

  ```

Operations Methods

- `let`

  ```
  let val1=9*3
  ```

- double parentheses `((` `))`

  ```
  val1=$((10*5+15))
  echo $val1
  ```

- `expr`

  ```
  expr '10 + 30'
  ```

- `bc`

  ```
  echo "scale=2; 55/3" | bc
  ```

Compound Operators

Operator	Name	Meaning
+=	Plus-equal	increment variable by a constant or right side variable
-=	*minus-equal*	decrements variable by a constant
*=	times-equal	
/=	slash-equal	
%=	mod-equal	
(post)++		
(post)--		
++(pre)		
--(pre)		

- **Example**:

  ```
  a=200
  let "a += 3"
  echo $a
  ```

  Output:

  ```
  
  ```

- **Example**:

  ```
  a=200
  let "a -= 3"
  echo $a
  ```

  Output:

  ```
  
  ```

- **Example**:

  ```
  a=200
  let "a *= 3"
  echo $a
  ```

  Output:

```

```

- **Example**:

  ```
  a=200
  let "a /= 3"
  echo $a
  ```

  Output:

  ```

  ```

- **Example**:

  ```
  a=200
  let "a %= 3"
  echo $a
  ```

  Output:

  ```

  ```

bitwise operators

Operator	Name	Meaning
`<<`	bitwise left shift	multiplies by `2` for each shift position
`<<=`	left-shift-equal	multiplied by `2` and assign it to variable
`>>`	bitwise right shift	divides by `2` for each shift position
`>=`	right-shift-equal	inverse of `<<=`
`&`	bitwise-AND	
`&=`	bitwise-AND-equal	
`|`	bitwise-OR	
`|=`	bitwise OR-equal	
`~`	bitwise NOT	
`^`	bitwise XOR	
`^=`	bitwise XOR-equal	

Table: Binary Table

A	B	Equivalent Integer
0	0	0
0	1	1
1	0	2
1	1	3

Table: Binary AND Table

A	B	Result
0	0	0
0	1	0
1	0	0
1	1	1

Table: Binary OR Table

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1

- **Example**:

```
set -o nounset

grapes=12

echo $(( grapes<<2 ))
echo "Grapes=" $grapes

grapes=12
echo $(( grapes<<=2 ))
echo "Grapes=" $grapes

grapes=12
echo $(( grapes>>2 ))
echo "Grapes=" $grapes
```

```

grapes=12
echo $(( grapes>>=2 ))
echo "Grapes=" $grapes
echo $grapes
```

Output:

```
48
Grapes= 12
48
Grapes= 48
3
Grapes= 12
3
Grapes= 3
3
```

- **Example**:

```
set -o nounset

grapes=12

echo $(( grapes&2 ))
echo "Grapes=" $grapes
grapes=12
echo $(( grapes&=2 ))
echo "Grapes=" $grapes
grapes=12
echo $(( grapes|2 ))
echo "Grapes=" $grapes
grapes=12
echo $(( grapes|=2 ))
echo "Grapes=" $grapes
grapes=12
echo $(( ~grapes ))
echo "Grapes=" $grapes
grapes=12
echo $(( grapes^2 ))
echo "Grapes=" $grapes
grapes=12
echo $(( grapes^=2 ))
echo "Grapes=" $grapes
```

Output:

```
0
Grapes= 12
0
```

```
Grapes= 0
14
Grapes= 12
14
Grapes= 14
-13
Grapes= 12
14
Grapes= 12
14
Grapes= 14
```

Logical Operators

Operator	Name	Meaning
`&&`	AND	
`||`	OR	
`!`	NOT	
`?:`	Ternary	similar to `if` statement
`,`	Comma	to execute multiple statements in a line

- **Example**:

```
 set -o nounset                              # Treat unset variables as an error

 user="mayank"
 passwd="welcome"

 # Both the user and password should be correct
 if [[ $user = "mayank" && $passwd = "welcome" ]]
 then
   echo "welcome to the game"
 else
   echo "Invalid user or password"
 fi

 # Both the user and password should be correct
 if [[ $user = "mayank" && $passwd = "welcome" ]]
 then
   echo "welcome to the game"
 else
   echo "Invalid user or password"
 fi
```

Output:

```
welcome to the game
Invalid user or password
```

- **Example**:

```
set -o nounset                               # Treat unset variables as an error

user="mayank"
passwd="welcome"

# Both the user and password should be correct
if [[ $user = "mayank" || $passwd = "welcome" ]]
then
  echo "welcome to the game"
else
  echo "Invalid user or password"
fi

# Both the user and password should be correct
if [[ $user = "mayank" || $passwd = "welcome" ]]
then
  echo "welcome to the game"
else
  echo "Invalid user or password"
fi

# Both the user and password should be correct
if [[ $user = "Roshan" || $passwd = "Sat" ]]
then
  echo "welcome to the game"
else
  echo "Invalid user or password"
fi
```

Output:

```
welcome to the game
welcome to the game
Invalid user or password
```

```
terminate=$1
if [[ !$terminate  ]]
then
  echo "Program is running"
else
  echo "Program is terminated"
fi
```

```
n=20
v1=100
v2=200
echo $(( n>=20 ? v1 : v2 ))
```

string operator

Operator	Name	Meaning
`<`	less	
`>`	greater	
`-z`	length zero	checks if the length is zero or not
`-n`	non zero	Checks if the length is non-zero or not

Logical Operators

Operator	Name	Meaning
`-a`	Boolean AND	
`-o`	Boolean OR	

Iterators

`for`

Syntax:

```
for (( Initialization; Control Condition; Iteration ))
do

    -- Block of Commands --

done
```

Examples:

```
set -o nounset                                    # Treat unset variables as an error

for (( i=1; i<=10; i+=2 ))
do
   echo "Loop $i"
done
for (( ; ; ))
do
   echo "You are in an Infinite Loop. Press CTRL + C to Exit.."
done
```

Range of integers

Syntax:

```
for VAR in 1 2 3 .. N
do
    -- Block of Commands --
done
for i in 1 2 3 4 5
do
    echo "You are in Loop No. $i"
done
```

Syntax for start and end:

```
for VAR in {START..END}
do
    -- Block of Commands --
done
```

```
set -o nounset

for num in {1..4}
do
    echo "loop numer:" $num
done
```

Syntax for start, end and Step:

```
for VAR in {START..END..STEP}
do
    -- Block of Commands --
done
```

Example:

```
set -o nounset

for i in {2..20..4}
do
    echo "Loop No. $i"
done

echo "--------------------"
for i in {-10..20..4}
do
    echo "Loop No. $i"
done
```

Output:

```
❯ bash 05_range_start_end_step.sh
Loop No. 2
Loop No. 6
Loop No. 10
Loop No. 14
Loop No. 18
--------------------
Loop No. -10
Loop No. -6
Loop No. -2
Loop No. 2
Loop No. 6
Loop No. 10
Loop No. 14
Loop No. 18
```

Using `seq` Command

Syntax:

```
seq END
seq START END
seq START INC END
```

Example:

```
#!/bin/bash
for i in $(seq 0 3 15)
do
   echo "You are in Loop No. $i"
done
```

`break` in `for`

Syntax:

```
for VAR in {RANGE}
do
    -- Some Commands --

    if [ condition ]
    then
        break
    fi

    -- More commands --
done
```

Example:

```
#!/bin/bash

for file in /etc/init.d/*
do
    if [ "$file" == "/etc/init.d/networking" ]
    then
        echo "Found!"
        break
    fi
done
echo "Search Complete!"
```

`continue` in `for`

Syntax:

```
 for VAR in {RANGE}
 do
     -- Some Commands --

     if [ condition ]
     then
         continue
     fi

     -- More commands --
 done

```

```bash
for i in {1..20}
do
    j=$(expr $i % 2)

    if [ $j == 0 ]
    then
        continue
    fi

    echo "$i"
done
```

`while`

Bash also provide `while` construct, which will run the command in its code block as long as the condition is true.

Syntax:

```bash
while [ condition ]
do
   < command >
done
```

Examples

- Simple while..do

```bash
set -o nounset          # Treat unset variables as an error

declare -i count=4
# count=4

while [[ $count > 0 ]]
do
    echo $count
    # count=$count-1      # When using declare -1
    # let "count-=1"    # when not using declare
    ((count--))         # This one works in both
done
```

- Guss the number

```bash
echo "Lets play a game of gussing the number"

# We can use the below one also
# num=`shuf -i 1-5 -n 1`
# You can increase the numbers by replacing 5 with the number you want
```

```
num=$(( ( RANDOM % 5 ) + 1 ))

declare -i gussed_number=-1

while [[ $num -ne $gussed_number ]]
do
    echo "Please enter your guss from (0-5): "
    read gussed_number

    case $gussed_number in
        $num)
            echo "You gussed the right number"
            ;;
        *)
            echo "Sorry, the gussed number was wrong"
            ;;
    esac
done
```

- Emulating do..while

```
while true; do
    body
    [[ condition ]] || break
done
```

- Another do..while

```
num=`shuf -i 1-5 -n 1`
declare -u gussed_number

while
  echo "Enter number your guess (1-5): "
  read gussed_number
  [[ $num -ne $gussed_number ]]
do true; done

```

`until`

`until` allows to run the code block commands till the `condition` is met. it is reverse of `while` loop.

Syntax:

```
until [ Condition ]
do
    < commands >
done
```

Examples:

```
#!/bin/bash

n=1

until [ $n -gt 5 ]
do
    echo "Iterated $n times"
    n=$( expr $n + 1 )
done
```

- **Example**: Infinite

```
#!/bin/bash

until [ 1 -ne 1 ]
do
   echo "Press CTRL + C to Quit.."
done
```

`break` in `until`

Syntax:

```
until [ condition ]
do
    -- Some Commands --
    if [ condition ]
    then
        break
    fi
    -- More commands --
done
```

```
until [ 5 -gt 5 ]
do
    echo "Enter a single-digit number :"
    read n

    if [ $n -gt 9 ]
    then
        echo "Wrong Entry! Program will terminate now.."
        break
    fi

    echo "Good!"
done
```

`continue` in `until`

```
until [ 5 -gt 5 ]
do
    echo "Enter a single-digit number :"
    read n

    if [ $n -gt 9 ]
    then
        echo "Wrong Entry! Program will terminate now.."
        break
    elif [ $n -gt 7 ]
    then
        echo "Wrong Entry! Plese try again"
        continue
    fi

    echo "Good!"
done
```

Bash Functions

Just like other languages, `bash` also provides functions which can be used to club a set of command which we wish to call multiple times.

Syntax

- **multi line version**

```
function_name () {
  < commands >
}
```

or,

```
function function_name {
  < commands >
}
```

- **single line version**

```
function_name () { <commands> ; }
```

or

```
function function_name { < commands > ; }
```

> **Note**
>
> ---
>
> - **Body of function**: All code between `{` `}` It can be multi line or single line, and each line can separate using `;` (semi-colon).
> - As usual, there should be a space or newline between brackets and code
> - Function must be placed before it is called
> - Function can be called by using its `function_name`
> - Just like variable names, function names should also be descriptive and unique
> - When using single-line format, last command must end with `;` semi-colon

Examples

```
set -o nounset                                  # Treat unset variables as an error

function namaskar() {
    # function_body
    echo "Namaskar"
}

namaskar
```

Output:

```
Namaskar
```

Variables Scope

Global Variables

All variables, in bash, are global variables unless explicitly defined. You can access the variables defined with-in a function from outside the function without any issue as shown in the below example.

```
Namaskar() {
    echo $wish
}

wish="!!! Guten Morgan !!!"
Namaskar
```

Output:

```
!!! Guten Morgan !!!
```

```

Namaskar () {
    echo $wish
    wish="!!! Guten Morgan !!!"
    echo $wish
}

wish="!!! Good Morning !!!"
Namaskar
echo $wish

```

Any changes done to the global variable within the function, will be reflected outside the function also.

Local Variables

As shown in the above section, all variables in bash. One exception to the rule is when we declare the variable with `local` keyword as shown in the below example

```
Namaskar() {
    local msg="Namaskar"
    echo $wish $msg
}

wish="!!! Guten Morgan !!!"
Namaskar
```

```

Namaskar() {
    local msg="Namaskar"
    echo $wish $msg
}

msg="Good morning"
wish="!!! Guten Morgan !!!"
Namaskar
echo $msg
```

If same variable name is used for a global & local variable, then with-in the function `local` variable will rule.

Return Values

```
Namaskar() {
    local msg="Namaskar"
    echo $wish $msg
    return 100
}

wish="!!! Guten Morgan !!!"
Namaskar
echo $?
```

Passing Arguments to Bash Functions

```
Namaskar () {
  echo "Namaskar $1 Ji"
}

Namaskar "Rakesh"
```

Files

Read file

Method 1: Read file line by line - One liner

```
while read -r line; do COMMAND; done < input.file
```

Method 2: Read file line by line - Basic Version

```
#!/bin/bash
input="input.file"

while IFS= read -r line
do
  echo "$line"
done < "$input"
```

`JSON` files

Check `jq` installed

```
#!/bin/bash

function checkJQ
{
    dummy=$(echo "{ }" | jq)
    if [ "$dummy" != "{}" ]; then
        echo "jq not installed, or not working"
        exit 1
    else
        echo "jq installed and working"
    fi
}

checkJQ
```

Output:

```
→ bash jq_check.sh
jq installed and working

```

Read `json` file

```
readJsonFile() {
    echo "$1"
        echo $json | jq -r $1
}

json=$(cat "capitals.json")
readJsonFile
```

Query `json` file

```
data='{"data": ["Mango", "Apple"], "address": {"street": "1 India", "Colony": "Hindu colony", "City": "Ayodhya", "State": "UP"}}'

echo "${data}"
processed=`echo "${data}" | jq '.data'`

echo "After processing:"
echo "${processed}"

echo "Query address"

processed=`echo "${data}" | jq '.address'`

echo "After processing:"
echo "${processed}"


echo "Query address street"

processed=`echo "${data}" | jq '.address.street'`

echo "After processing:"
echo "${processed}"
```

Output:

```
{"data": ["Mango", "Apple"], "address": {"street": "1 India", "Colony": "Hindu colony", "City": "Ayodhya", "State": "UP"}}
After processing:
[
  "Mango",
  "Apple"
]
Query address
After processing:
{
  "street": "1 India",
  "Colony": "Hindu colony",
  "City": "Ayodhya",
  "State": "UP"
}
Query address street
After processing:
"1 India"

```

Example: Advance Query

```
readJsonFile() {
    echo $json

    echo "get all keys in users"
    echo $json | jq -r '.users[] | keys '

    echo "get all keys in length"
    echo $json | jq -r '.users | length '


    echo "get true and false for users who have location"
    echo $json | jq -r '.users' | jq -r  'map(has("location")) '

    echo "Lets get id and name"
    # Add `| tostring` to convert other datatypes to string
    echo $json | jq -r '.users[] | .name + " " + ( .id | tostring ) '

    # echo "Lets get id and name as csv"
    # echo $json | jq -r '.users[] | [ .name  ,   ( .id | tostring ) ]  | @csv'

}

json=$(cat "small.json")
readJsonFile

```

Output:

```
→ bash query_adv.sh
{ "data": [ "Test", "test1" ], "1": "a", "2": "b", "users": [ { "id": "001",
"name": "Rahul" }, { "id": "002", "name": "Rakesh" }, { "id": "003", "name":
"Shashank", "location": "India" } ] }
get all keys in users
[
  "id",
  "name"
]
[
  "id",
  "name"
]
[
  "id",
  "location",
  "name"
]
get all keys in length
3
get true and false for users who have location
[
  false,
  false,
  true
]
Lets get id and name
Rahul 001
Rakesh 002
Shashank 003
```

Convert json output to csv file

```
# part of query_adv.sh script
echo "Lets get id and name as csv"
echo $json | jq -r '.users[] | [ .name  ,   ( .id | tostring ) ]  | @csv'
```

Output:

```
→ bash query_adv.sh
{ "data": [ "Test", "test1" ], "1": "a", "2": "b", "users": [ { "id": "001",
"name": "Rahul" }, { "id": "002", "name": "Rakesh" }, { "id": "003", "name":
"Shashank" } ] }
Lets get id and name
Rahul 001
Rakesh 002
Shashank 003
Lets get id and name as csv
"Rahul","001"
"Rakesh","002"
"Shashank","003"
```

Validate json file

```
function checkJsonFile
{
    cat "$1" | jq empty; echo $?
}

checkJsonFile "$1"
```

Output: Good File

```
→ bash check.sh  "astronotes.json"
0

```

Output: Bad File

```
→ bash check.sh  "w file.json"
parse error: Objects must consist of key:value pairs at line 2, column 24
4

```

Prettify JSON

```
data='{"data": ["Mango", "Apple"], "address": {"street": "1 India", "Colony":
"Hindu colony", "City": "Ayodhya", "State": "UP"}}'

echo "${data}"
processed=`echo "${data}" | jq '.'`

echo "After processing:"
echo "${processed}"
```

Output:

```
→ bash pretty_json.sh
{"data": ["Mango", "Apple"], "address": {"street": "1 India", "Colony": "Hindu colony", "City": "Ayodhya", "State": "UP"}}
After processing:
{
  "data": [
    "Mango",
    "Apple"
  ],
  "address": {
    "street": "1 India",
    "Colony": "Hindu colony",
    "City": "Ayodhya",
    "State": "UP"
  }
}

```

Appendix

Good Practices

Comment your code

Just like programming, commenting bash scripts is also a good idea.

```
# substracting base price with sale value
base_price=12   # Base cost
sale_value=32   # Sale price
raw_profit=$(( $sale_value - $base_price )) # We might wish to add other expenditures in future
echo $raw_profit
```

Keep your code clean and use a style guide

```
#!/bin/bash
#================================================================================
#
#          FILE: 001_basic.sh
#
#         USAGE: ./001_basic.sh
#
#   DESCRIPTION:
#
#       OPTIONS: ---
#  REQUIREMENTS: ---
#          BUGS: ---
#         NOTES: ---
#        AUTHOR: Mayank Johri (Mayank), johri
#  ORGANIZATION: johrimayank@yandex.com
#       CREATED: 09/06/2021 06:55:02 AM
#      REVISION:  ---
#================================================================================

set -o nounset                                  # Treat unset variables as an
error

#######################################
# Prints Namaskar
# Globals:
#   None
# Arguments:
#   None
# Outputs:
#   Writes Namaskar to stdout
#######################################
function namaskar() {
    # function_body
    echo "Namaskar"
}

namaskar
```

Add `usage` & `help` functions

We should have `usage` or `help` function in our main script, which will help the users to run the script

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

function help
{
    echo "Usage: "
    echo " -h:   Displays this screen"
    echo " -c:   User country"
}

function process_opts
{
    while (( "$#" ))
    do
        case "$1" in
            "-h"|"--help")
                help
                exit
                ;;
            "-c"|"--country")

                if [ -n "${2}" ] && [ ${2:0:1} != "-" ]
                then
                    echo $2
                    country=$2
                    shift 2
                fi
        esac
    done
}


declare country
process_opts "${@}"

echo "User is from ${country:-Unknown} country."
```

Using `getopts`

```
TODO
```

Validate the parameters before using them

```
TODO
```

Set default values for parameters

```
function add {

    a=${1:-10}
    b=${2:-23}
    echo $(( a + b ))

}

add 14 45
add 14
add
```

Output:

```
59
37
33
```

Script should exit when it fails

We can force the script to exit if any of the code fails by settings `set -o errexit` or `set -e` command as shown in the below example

```
set -o errexit
# or
set -e
```

Tips

Cheat Sheet Tables

Table A-1. Common Shell Features

Command	Meaning
>	Redirect output
>>	Append to file
<	Redirect input
<<	"Here" document (redirect input)
|	Pipe output
&	Run process in background.
;	Separate commands on same line
*	Match any character(s) in filename
?	Match single character in filename
[ ]	Match any characters enclosed
( )	Execute in subshell
	Substitute output of enclosed command
" "	Partial quote (allows variable and command expansion)
' '	Full quote (no expansion)
\	Quote following character
$var	Use value for variable
$$	Process id
$0	Command name
$n	nth argument (n from 0 to 9)
`#`	Begin comment
bg	Background execution

Command	**Meaning**
break	Break from loop statements
cd	Change directories
continue	Resume a program loop
echo	Display output
eval	Evaluate arguments
exec	Execute a new shell
fg	Foreground execution
jobs	Show active jobs
kill	Terminate running jobs
newgrp	Change to a new group
shift	Shift positional parameters
stop	Suspend a background job
suspend	Suspend a foreground job
time	Time a command
umask	Set or list file permissions
unset	Erase variable or function definitions
wait	Wait for a background job to finish

Table A-2: Differing Shell Features

sh	**bash**	**ksh**	**csh**	**Meaning/Action**
$	$	$	%	Default user prompt
	>|	**>|**	**>!**	Force redirection
> `file` **2>&1**	**&>** `file` or **>** `file` **2>&1**	**>** `file` **2>&1**	**>&** `file`	Redirect stdout and stderr to `file`
	{ }		**{ }**	Expand elements in list
`command`	`command` or **$(command)**	**$(command)**	`command`	Substitute output of enclosed **command**
`$HOME`	`$HOME`	`$HOME`	`$home`	Home directory
	~	~	~	Home directory symbol
	`~+`, `~-`, **dirs**	`~+`, `~-`	`=-`, `=N`	Access directory stack
`var` **=value**	`VAR` **=value**	`var` **=value**	**set** `var` **=value**	Variable assignment
export `var`	**export** `VAR` **=value**	**export** `var` **=val**	**setenv** `var` `\*val\*`	Set environment variable

sh	bash	ksh	csh	Meaning/Action
	`${nnnn}`	`${nn}`		More than 9 arguments can be referenced
"`$@`"	"`$@`"	"`$@`"		All arguments as separate words
`$#`	`$#`	`$#`	`$#argv`	Number of arguments
`$?`	`$?`	`$?`	`$status`	Exit status of the most recently executed command
`$!`	`$!`	`$!`		PID of most recently backgrounded process
`$-`	`$-`	`$-`		Current options
. `file`	**source** `file` or **.** `file`	**.** `file`	**source** `file`	Read commands in file
	alias x='y'	**alias x=y**	**alias x y**	Name **x** stands for command **y**
case	**case**	**case**	**switch** or **case**	Choose alternatives
done	**done**	**done**	**end**	End a loop statement
esac	**esac**	**esac**	**endsw**	End **case** or **switch**
exit `\*n\*`	**exit** `\*n\*`	**exit** `\*n\*`	**exit** `\*(expr)\*`	Exit with a status
for/**do**	**for**/**do**	**for**/**do**	**foreach**	Loop through variables
	set `-f`, **set** `-o nullglob|dotglob|nocaseglob|noglob`		**noglob**	Ignore substitution characters for filename generation
hash	**hash**	**alias** `-t`	**hashstat**	Display hashed commands (tracked aliases)
hash `\*cmds\*`	**hash** `\*cmds\*`	**alias** `-t` `\*cmds\*`	**rehash**	Remember command locations
hash `-r`	**hash** `-r`		**unhash**	Forget command locations
	history	**history**	**history**	List previous commands
	ArrowUp+**Enter** or **!!**	**r**	**!!**	Redo previous command
	!`\*str\*`	**r** `\*str\*`	**!**`\*str\*`	Redo last command that starts with "str"
	!`\*cmd\*`**:s/**`\*x\*`**/**`\*y\*`**/**	**r** `\*x\*`**=**`\*y\*` `\*cmd\*`	**!**`\*cmd\*`**:s/**`\*x\*`**/**`\*y\*`**/**	Replace "x" with "y" in most recent command starting with "cmd", then execute.
if [`$i` **-eq** `\*5\*` **]**	**if [** `$i` **-eq** `\*5\*` **]**	**if ((** `i` **==** `\*5\*` **))**	**if (** `$i` **==** `\*5\*` **)**	Sample condition test
fi	**fi**	**fi**	**endif**	End **if** statement
ulimit	**ulimit**	**ulimit**	**limit**	Set resource limits
pwd	**pwd**	**pwd**	**dirs**	Print working directory
read	**read**	**read**	**$<**	Read from terminal
trap `\*2\*`	**trap** `\*2\*`	**trap** `\*2\*`	**onintr**	Ignore interrupts
	unalias	**unalias**	**unalias**	Remove aliases
until	**until**	**until**		Begin **until** loop
while/**do**	**while**/**do**	**while**/**do**	**while**	Begin **while** loop

Quests

Basics

1. What is Shell?
2. Why do we need shell script?
3. Explain few advantages of shell scripting.
4. Explain few limitations of shell scripting.
5. When not to use shell programming/scripting?
6. What is the shell script files called
7. Write few major types of shells used in Linux.
8. List few similarity and differences between *Bourne Shell* and *C Shell*
9. List few similarity and differences between *Bourne Shell* and *fish Shell*
10. How to create a Shell variable?
11. What are different types of variables used in shell scripting?
12. List few shell script commands?
13. Explain positional parameters in functions and scripts.
14. What are control instructions?
15. List all types of control instructions?
16. What is the *shebang* line in shell scripting and what is its use?
17. What is a file system?
18. Explain four core components of the Linux file system.
19. How do you include comments in a shell script?
20. How many ways we can run a shell script?
21. How to do input and output redirection in a shell script?
22. What are some common shell environments?
23. How do you use command line arguments and options in a shell script?
24. How do you use loops (e.g., for, while) in a shell script?
25. What are the default permissions of a file when it is created?
26. How can we make a variable `readonly`?
27. which all ways a shell script can be executed.
28. How to find the number of arguments passed to the script?
29. Describe the shell variable's scope'?
30. What is `$#` and its uses?
31. How can we run a script in the background?

Advance

1. Describe `crontab` and its uses.
2. List all the shells available in the system?
3. Name the command that is used to compare the strings in a shell script.
4. Write the difference between `$*` and `$@`
5. Describe all supported types of loops in bash.
6. How to run the script in interactive and non-interactive mode?
7. What does "`s`" permission bit in a file means?
8. Describe all the debug processes in detail.
9. What are *metacharacters*?
10. What are the differences between “=” and “==”.
11. How to list all Shell environment variables.
12. Write few examples of managing and manipulating files and directories?
13. How to open a read-only file in Shell.
14. How to know how long the system has been running.
15. Write the difference between `$$` and `$!`
16. What is the difference between `grep` and `find` command.
17. How to create a *function* in shell script?
18. How to enhance the performance of the shell script?
19. How to manipulate and manage processes, such as killing, starting, or stopping them?
20. How do you use shell scripts to interact with APIs or other external services?
21. How you will check if a file exists on the filesystem.
22. What is the difference between `[[ $string == "efg*" ]]` and `[[ $string == efg* ]]`
23. Find the number of lines in a file that contains the letter "LINUX."
24. Describe about Zombie Processes in shell scripting?
25. How GUI components are added to shell Scripting? [**Ans**: use `zenity`]

Free Online Shell service providers

Large list can be found at https://shells.red-pill.eu/

- http://www.openshells.net/
- https://www.xshellz.com/
- https://blinkenshell.org/wiki/Start

References

- Bash Guide for Beginners: https://tldp.org/LDP/Bash-Beginners-Guide/html/ **{Must Read}**
- Bash Guide: https://www.gnu.org/software/bash/manual/bash.pdf
- Debugging: https://unix.stackexchange.com/questions/155551/how-to-debug-a-bash-script **{Must Read}**
- **Google Style Guide: https://google.github.io/styleguide/shellguide.html {Must Read}**
- https://en.wikipedia.org/wiki/Shebang_(Unix)
- https://en.wikipedia.org/wiki/List_of_terminal_emulators
- https://packages.gentoo.org/categories/x11-terms
- https://wiki.archlinux.org/title/List_of_applications#Terminal_emulators
- https://poor.dev/blog/terminal-anatomy/ **{Must Read}**