bash-handbook

 npm v2.9.7 chat on gitter

This document is written for those who want to learn Bash without diving in too deeply.

> **Tip**: Try **learnyoubash** — an interactive workshopper based on this handbook!

Node Packaged Manuscript

You can install this handbook using `npm`. Just run:

```
$ npm install -g bash-handbook
```

You should be able to run `bash-handbook` at the command line now. This will open the manual in your selected `$PAGER`. Otherwise, you may continue reading on here.

The source is available here: https://github.com/denysdovhan/bash-handbook

Table of Contents

Introduction

If you are a developer, then you know the value of time. Optimizing your work process is one of the most important aspects of the job.

In that path towards efficiency and productivity, we are often posed with actions that must be repeated over and over again, like:

- taking a screenshot and uploading it to a server
- processing text that may come in many shapes and forms
- converting files between different formats
- parsing a program's output

Enter **Bash**, our savior.

Bash is a Unix shell written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell. It was released in 1989 and has been distributed as the Linux and macOS default shell for a long time.

So why do we need to learn something that was written more than 30 years ago? The answer is simple: this *something* is today one of the most powerful and portable tools for writing efficient scripts for all Unix-based systems. And that's why you should learn bash. Period.

In this handbook, I'm going to describe the most important concepts in bash with examples. I hope this compendium will be helpful to you.

Shells and modes

The user bash shell can work in two modes - interactive and non-interactive.

Interactive mode

If you are working on Ubuntu, you have seven virtual terminals available to you. The desktop environment takes place in the seventh virtual terminal, so you can return to a friendly GUI using the `Ctrl-Alt-F7` keybinding.

You can open the shell using the `Ctrl-Alt-F1` keybinding. After that, the familiar GUI will disappear and one of the virtual terminals will be shown.

If you see something like this, then you are working in interactive mode:

```
user@host:~$
```

Here you can enter a variety of Unix commands, such as `ls`, `grep`, `cd`, `mkdir`, `rm` and see the result of their execution.

We call this shell interactive because it interacts directly with the user.

Using a virtual terminal is not really convenient. For example, if you want to edit a document and execute another command at the same time, you are better off using virtual terminal emulators like:

- GNOME Terminal
- Terminator
- iTerm2
- ConEmu

Non-interactive mode

In non-interactive mode, the shell reads commands from a file or a pipe and executes them. When the interpreter reaches the end of the file, the shell process terminates the session and returns to the parent process.

Use the following commands for running the shell in non-interactive mode:

```
sh /path/to/script.sh
bash /path/to/script.sh
```

In the example above, `script.sh` is just a regular text file that consists of commands the shell interpreter can evaluate and `sh` or `bash` is the shell's interpreter program. You can create `script.sh` using your preferred text editor (e.g. vim, nano, Sublime Text, Atom, etc).

You can also simplify invoking the script by making it an executable file using the `chmod` command:

```
chmod +x /path/to/script.sh
```

Additionally, the first line in the script must indicate which program it should use to run the file, like so:

```
#!/bin/bash
echo "Hello, world!"
```

Or if you prefer to use `sh` instead of `bash`, change `#!/bin/bash` to `#!/bin/sh`. This `#!` character sequence is known as the shebang. Now you can run the script like this:

```
/path/to/script.sh
```

A handy trick we used above is using `echo` to print text to the terminal screen.

Another way to use the shebang line is as follows:

```
#!/usr/bin/env bash
echo "Hello, world!"
```

The advantage of this shebang line is it will search for the program (in this case `bash`) based on the `PATH` environment variable. This is often preferred over the first method shown above, as the location of a program on a filesystem cannot always be assumed. This is also useful if the `PATH` variable on a system has been configured to point to an alternate version of the program. For instance, one might install a newer version of `bash` while preserving the original version and insert the location of the newer version into the `PATH` variable. The use of `#!/bin/bash` would result in using the original `bash`, while `#!/usr/bin/env bash` would make use of the newer version.

Exit codes

Every command returns an **exit code** (**return status** or **exit status**). A successful command always returns `0` (zero-code), and a command that has failed returns a non-zero value (error code). Failure codes must be positive integers between 1 and 255.

Another handy command we can use when writing a script is `exit`. This command is used to terminate the current execution and deliver an exit code to the shell. Running an `exit` code without any arguments, will terminate the running script and return the exit code of the last command executed before `exit`.

When a program terminates, the shell assigns its **exit code** to the `$?` environment variable. The `$?` variable is how we usually test whether a script has succeeded or not in its execution.

In the same way we can use `exit` to terminate a script, we can use the `return` command to exit a function and return an **exit code** to the caller. You can use `exit` inside a function too and this will exit the function *and* terminate the program.

Comments

Scripts may contain *comments*. Comments are special statements ignored by the `shell` interpreter. They begin with a `#` symbol and continue on to the end of the line.

For example:

```
#!/bin/bash
# This script will print your username.
whoami
```

> **Tip:** Use comments to explain what your script does and *why*.

Variables

Like in most programming languages, you can also create variables in bash.

Bash knows no data types. Variables can contain only numbers or a string of one or more characters. There are three kinds of variables you can create: local variables, environment variables and variables as *positional parameters*.

Local variables

Local variables are variables that exist only within a single script. They are inaccessible to other programs and scripts.

A local variable can be declared using `=` sign (as a rule, there **should not** be any spaces between a variable's name, `=` and its value) and its value can be retrieved using the `$` sign. For example:

```
username="denysdovhan"  # declare variable
echo $username          # display value
unset username          # delete variable
```

We can also declare a variable local to a single function using the `local` keyword. Doing so causes the variable to disappear when the function exits.

```
local local_var="I'm a local value"
```

Environment variables

Environment variables are variables accessible to any program or script running in current shell session. They are created just like local variables, but using the keyword `export` instead.

```
export GLOBAL_VAR="I'm a global variable"
```

There are *a lot* of global variables in bash. You will meet these variables fairly often, so here is a quick lookup table with the most practical ones:

Variable	Description
`$HOME`	The current user's home directory.
`$PATH`	A colon-separated list of directories in which the shell looks for commands.
`$PWD`	The current working directory.
`$RANDOM`	Random integer between 0 and 32767.
`$UID`	The numeric, real user ID of the current user.
`$PS1`	The primary prompt string.
`$PS2`	The secondary prompt string.

Double and single quotes

There is an important difference between double and single quotes. Inside double quotes variables or command substitutions are expanded. Inside single quotes they are not. For example:

```
echo "Your home: $HOME" # Your home: /Users/<username>
echo 'Your home: $HOME' # Your home: $HOME
```

Take care to expand local variables and environment variables within quotes if they could contain whitespace. As an innocuous example, consider using `echo` to print some user input:

```
INPUT="A string  with   strange    whitespace."
echo $INPUT   # A string with strange whitespace.
echo "$INPUT" # A string  with   strange    whitespace.
```

The first `echo` is invoked with 5 separate arguments — $INPUT is split into separate words, `echo` prints a single space character between each. In the second case, `echo` is invoked with a single argument (the entire $INPUT value, including whitespace).

Now consider a more serious example:

```
FILE="Favorite Things.txt"
cat $FILE   # attempts to print 2 files: `Favorite` and `Things.txt`
cat "$FILE" # prints 1 file: `Favorite Things.txt`
```

While the issue in this example could be resolved by renaming FILE to `Favorite-Things.txt`, consider input coming from an environment variable, a positional parameter, or the output of another command (`find`, `cat`, etc). If the input *might* contain whitespace, take care to wrap the expansion in quotes.

Array expansion

Individual array elements are expanded similar to other variables:

```
echo ${fruits[1]} # Pear
```

The entire array can be expanded by using `*` or `@` in place of the numeric index:

```
echo ${fruits[*]} # Apple Pear Plum
echo ${fruits[@]} # Apple Pear Plum
```

There is an important (and subtle) difference between the two lines above: consider an array element containing whitespace:

```
fruits[0]=Apple
fruits[1]="Desert fig"
fruits[2]=Plum
```

We want to print each element of the array on a separate line, so we try to use the `printf` builtin:

```
printf "+ %s\n" ${fruits[*]}
# + Apple
# + Desert
# + fig
# + Plum
```

Why were `Desert` and `fig` printed on separate lines? Let's try to use quoting:

```
printf "+ %s\n" "${fruits[*]}"
# + Apple Desert fig Plum
```

Now everything is on one line — that's not what we wanted! Here's where `${fruits[@]}` comes into play:

```
printf "+ %s\n" "${fruits[@]}"
# + Apple
# + Desert fig
# + Plum
```

Within double quotes, `${fruits[@]}` expands to a separate argument for each element in the array; whitespace in the array elements is preserved.

Array slice

Besides, we can extract a slice of array using the *slice* operators:

```
echo ${fruits[@]:0:2} # Apple Desert fig
```

In the example above, `${fruits[@]}` expands to the entire contents of the array, and `:0:2` extracts the slice of length 2, that starts at index 0.

Adding elements into an array

Adding elements into an array is quite simple too. Compound assignments are specially useful in this case. We can use them like this:

```
fruits=(Orange "${fruits[@]}" Banana Cherry)
echo ${fruits[@]} # Orange Apple Desert fig Plum Banana Cherry
```

The example above, `${fruits[@]}` expands to the entire contents of the array and substitutes it into the compound assignment, then assigns the new value into the `fruits` array mutating its original value.

Deleting elements from an array

To delete an element from an array, use the `unset` command:

```
unset fruits[0]
echo ${fruits[@]} # Apple Desert fig Plum Banana Cherry
```

Streams, pipes and lists

Bash has powerful tools for working with other programs and their outputs. Using streams we can send the output of a program into another program or file and thereby write logs or whatever we want.

Pipes give us opportunity to create conveyors and control the execution of commands.

It is paramount we understand how to use this powerful and sophisticated tool.

Streams

Bash receives input and sends output as sequences or **streams** of characters. These streams may be redirected into files or one into another.

There are three descriptors:

Code	Descriptor	Description
`0`	`stdin`	The standard input.
`1`	`stdout`	The standard output.
`2`	`stderr`	The errors output.

Redirection makes it possible to control where the output of a command goes to, and where the input of a command comes from. For redirecting streams these operators are used:

Operator	Description
`>`	Redirecting output
`&>`	Redirecting output and error output
`&>>`	Appending redirected output and error output
`<`	Redirecting input
`<<`	Here documents syntax
`<<<`	Here strings

Here are few examples of using redirections:

```
# output of ls will be written to list.txt
ls -l > list.txt

# append output to list.txt
ls -a >> list.txt

# all errors will be written to errors.txt
grep da * 2> errors.txt

# read from errors.txt
less < errors.txt
```

Pipes

We could redirect standard streams not only in files, but also to other programs. Pipes let us use the output of a program as the input of another.

In the example below, `command1` sends its output to `command2`, which then passes it on to the input of `command3`:

```
command1 | command2 | command3
```

Constructions like this are called **pipelines**.

In practice, this can be used to process data through several programs. For example, here the output of `ls -l` is sent to the `grep` program, which prints only files with a `.md` extension, and this output is finally sent to the `less` program:

```
ls -l | grep .md$ | less
```

The exit status of a pipeline is normally the exit status of the last command in the pipeline. The shell will not return a status until all the commands in the pipeline have completed. If you want your pipelines to be considered a failure if any of the commands in the pipeline fail, you should set the pipefail option with:

```
set -o pipefail
```

Lists of commands

A **list of commands** is a sequence of one or more pipelines separated by `;`, `&`, `&&` or `||` operator.

If a command is terminated by the control operator `&`, the shell executes the command asynchronously in a subshell. In other words, this command will be executed in the background.

Commands separated by a `;` are executed sequentially: one after another. The shell waits for the finish of each command.

```
# command2 will be executed after command1
command1 ; command2

# which is the same as
command1
command2
```

Lists separated by `&&` and `||` are called *AND* and *OR* lists, respectively.

The *AND-list* looks like this:

```
# command2 will be executed if, and only if, command1 finishes successfully (returns 0 exit status)
command1 && command2
```

The *OR-list* has the form:

```
# command2 will be executed if, and only if, command1 finishes unsuccessfully (returns code of error)
command1 || command2
```

The return code of an *AND* or *OR* list is the exit status of the last executed command.

Conditional statements

Like in other languages, Bash conditionals let us decide to perform an action or not. The result is determined by evaluating an expression, which should be enclosed in `[[ ]]`.

Conditional expression may contain `&&` and `||` operators, which are *AND* and *OR* accordingly. Besides this, there many other handy expressions.

There are two different conditional statements: `if` statement and `case` statement.

Primary and combining expressions

Expressions enclosed inside `[[ ]]` (or `[ ]` for `sh`) are called **test commands** or **primaries**. These expressions help us to indicate results of a conditional. In the tables below, we are using `[ ]`, because it works for `sh` too. Here is an answer about the difference between double and single square brackets in bash.

Working with the file system:

Primary	Meaning
`[ -e FILE ]`	True if `FILE` exists.
`[ -f FILE ]`	True if `FILE` exists and is a regular file.
`[ -d FILE ]`	True if `FILE` exists and is a directory.
`[ -s FILE ]`	True if `FILE` exists and not empty (size more than 0).
`[ -r FILE ]`	True if `FILE` exists and is readable.
`[ -w FILE ]`	True if `FILE` exists and is writable.
`[ -x FILE ]`	True if `FILE` exists and is executable.
`[ -L FILE ]`	True if `FILE` exists and is symbolic link.
`[ FILE1 -nt FILE2 ]`	FILE1 is newer than FILE2.
`[ FILE1 -ot FILE2 ]`	FILE1 is older than FILE2.

Working with strings:

Primary	Meaning
`[ -z STR ]`	`STR` is empty (the length is zero).
`[ -n STR ]`	`STR` is not empty (the length is non-zero).
`[ STR1 == STR2 ]`	`STR1` and `STR2` are equal.
`[ STR1 != STR2 ]`	`STR1` and `STR2` are not equal.

Sometimes `if..else` statements are not enough to do what we want to do. In this case we shouldn't forget about the existence of `if..elif..else` statements, which always come in handy.

Look at the example below:

```
if [[ `uname` == "Adam" ]]; then
  echo "Do not eat an apple!"
elif [[ `uname` == "Eva" ]]; then
  echo "Do not take an apple!"
else
  echo "Apples are delicious!"
fi
```

Using a `case` statement

If you are confronted with a couple of different possible actions to take, then using a `case` statement may be more useful than nested `if` statements. For more complex conditions use `case` like below:

```
case "$extension" in
  "jpg"|"jpeg")
    echo "It's image with jpeg extension."
  ;;
  "png")
    echo "It's image with png extension."
  ;;
  "gif")
    echo "Oh, it's a giphy!"
  ;;
  *)
    echo "Woops! It's not image!"
  ;;
esac
```

Each case is an expression matching a pattern. The `|` sign is used for separating multiple patterns, and the `)` operator terminates a pattern list. The commands for the first match are executed. `*` is the pattern for anything else that doesn't match the defined patterns. Each block of commands should be divided with the `;;` operator.

Loops

Here we won't be surprised. As in any programming language, a loop in bash is a block of code that iterates as long as the control conditional is true.

There are four types of loops in Bash: `for`, `while`, `until` and `select`.

`for` loop

The `for` is very similar to its sibling in C. It looks like this:

```
for arg in elem1 elem2 ... elemN
do
  # statements
done
```

During each pass through the loop, `arg` takes on the value from `elem1` to `elemN`. Values may also be wildcards or brace expansions.

Also, we can write `for` loop in one line, but in this case there needs to be a semicolon before `do`, like below:

```
for i in {1..5}; do echo $i; done
```

By the way, if `for..in..do` seems a little bit weird to you, you can also write `for` in C-like style such as:

```
for (( i = 0; i < 10; i++ )); do
  echo $i
done
```

`for` is handy when we want to do the same operation over each file in a directory. For example, if we need to move all `.bash` files into the `script` folder and then give them execute permissions, our script would look like this:

```
#!/bin/bash

for FILE in $HOME/*.bash; do
  mv "$FILE" "${HOME}/scripts"
  chmod +x "${HOME}/scripts/${FILE}"
done
```

`while` loop

The `while` loop tests a condition and loops over a sequence of commands so long as that condition is *true*. A condition is nothing more than a primary as used in `if..then` conditions. So a `while` loop looks like this:

```
while [[ condition ]]
do
  # statements
done
```

Just like in the case of the `for` loop, if we want to write `do` and condition in the same line, then we must use a semicolon before `do`.

A working example might look like this:

```
#!/bin/bash

# Squares of numbers from 0 through 9
x=0
while [[ $x -lt 10 ]]; do # value of x is less than 10
  echo $(( x * x ))
  x=$(( x + 1 )) # increase x
done
```

`until` loop

The `until` loop is the exact opposite of the `while` loop. Like a `while` it checks a test condition, but it keeps looping as long as this condition is *false*:

```
until [[ condition ]]; do
  #statements
done
```

`select` loop

The `select` loop helps us to organize a user menu. It has almost the same syntax as the `for` loop:

```
select answer in elem1 elem2 ... elemN
do
  # statements
done
```

The `select` prints all `elem1..elemN` on the screen with their sequence numbers, after that it prompts the user. Usually it looks like `$?` (`PS3` variable). The answer will be saved in `answer`. If `answer` is the number between `1..N`, then `statements` will execute and `select` will go to the next iteration — that's because we should use the `break` statement.

A working example might look like this:

```
#!/bin/bash

PS3="Choose the package manager: "
select ITEM in bower npm gem pip
do
  echo -n "Enter the package name: " && read PACKAGE
  case $ITEM in
    bower) bower install $PACKAGE ;;
    npm)   npm   install $PACKAGE ;;
    gem)   gem   install $PACKAGE ;;
    pip)   pip   install $PACKAGE ;;
  esac
  break # avoid infinite loop
done
```

This example, asks the user what package manager {s,he} would like to use. Then, it will ask what package we want to install and finally proceed to install it.

If we run this, we will get:

```
$ ./my_script
1) bower
2) npm
3) gem
4) pip
Choose the package manager: 2
Enter the package name: bash-handbook
<installing bash-handbook>
```

Loop control

There are situations when we need to stop a loop before its normal ending or step over an iteration. In these cases, we can use the shell built-in `break` and `continue` statements. Both of these work with every kind of loop.

The `break` statement is used to exit the current loop before its ending. We have already met with it.

The `continue` statement steps over one iteration. We can use it as such:

```
for (( i = 0; i < 10; i++ )); do
  if [[ $(( i % 2 )) -eq 0 ]]; then continue; fi
  echo $i
done
```

If we run the example above, it will print all odd numbers from 0 through 9.

Functions

In scripts we have the ability to define and call functions. As in any programming language, functions in bash are chunks of code, but there are differences.

In bash, functions are a sequence of commands grouped under a single name, that is the *name* of the function. Calling a function is the same as calling any other program, you just write the name and the function will be *invoked*.

We can declare our own function this way:

```
my_func () {
  # statements
}

my_func # call my_func
```

We must declare functions before we can invoke them.

Functions can take on arguments and return a result — exit code. Arguments, within functions, are treated in the same manner as arguments given to the script in non-interactive mode — using positional parameters. A result code can be *returned* using the `return` command.

Below is a function that takes a name and returns `0`, indicating successful execution.

```
# function with params
greeting () {
  if [[ -n $1 ]]; then
    echo "Hello, $1!"
  else
    echo "Hello, unknown!"
  fi
  return 0
}

greeting Denys  # Hello, Denys!
greeting        # Hello, unknown!
```

We already discussed exit codes. The `return` command without any arguments returns the exit code of the last executed command. Above, `return 0` will return a successful exit code. `0`.

Debugging

The shell gives us tools for debugging scripts. If we want to run a script in debug mode, we use a special option in our script's shebang:

```
#!/bin/bash options
```

These options are settings that change shell behavior. The following table is a list of options which might be useful to you:

Short	Name	Description
`-f`	noglob	Disable filename expansion (globbing).
`-i`	interactive	Script runs in *interactive* mode.
`-n`	noexec	Read commands, but don't execute them (syntax check).
	pipefail	Make pipelines fail if any commands fail, not just if the final command fail.
`-t`	—	Exit after first command.
`-v`	verbose	Print each command to `stderr` before executing it.
`-x`	xtrace	Print each command and its expanded arguments to `stderr` before executing it.

For example, we have script with `-x` option such as:

```bash
#!/bin/bash -x

for (( i = 0; i < 3; i++ )); do
  echo $i
done
```

This will print the value of the variables to `stdout` along with other useful information:

```
$ ./my_script
+ (( i = 0 ))
+ (( i < 3 ))
+ echo 0
0
+ (( i++  ))
+ (( i < 3 ))
+ echo 1
1
+ (( i++  ))
+ (( i < 3 ))
+ echo 2
2
+ (( i++  ))
+ (( i < 3 ))
```

Reading User Input

The user can enter data into shell variables using `read` commands.

`read`

This command reads input from stdin into variables

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars]
     [-p prompt] [-t timeout] [-u fd] [variable1 ...] [variable2 ...]
```

If no variable names are provided, the user input is stored in the variable `$REPLY` by default.

```
#!/bin/bash

read         #Waits for user input
echo $REPLY  #Prints the text
```

Short	Name	Description
`-a`	array	Store the words in an indexed array named `$array`
`-e`		Read data from terminal character by character till the delimiter is reached
`-d`	delimiter	Set the delimiting character to delimiter specified By default, newline('\n') is the delimiter
`-n`	nchars	Stop reading when **n** characters or delimiter is read
`-N`	nchars	Stops reading only when **n** charaters or EOF is read, **ignores delimiter**
`-p`	prompt	Prints prompt string on console
`-i`	interactive	Prints placeholder text which user can modify Used in conjunction with `-e`
`-r`	raw input	Disable shell interpretation of special charaters like $ and *
`-s`	silent	Disable echo of characters read onto terminal
`-t`	timout	Waits for certain amount of time before exitting
`-u`	file descriptor	Reads input from file descriptor specified

```
#!/bin/bash
read -p 'Enter your name: ' name
echo Hello, "$name"!

read -sp 'Enter Password: ' password
if [ "$password" == "1234" ]; then #Space around '[' is required
  echo -e "\nSuccess!"
else
  echo -e "\nTry Again!"
fi
```