

# Contents

<b>1</b>	<b>Bash Quick Guide</b>	<b>1</b>
<b>2</b>	<b>Commands and Arguments</b>	<b>2</b>
2.1	Preventing Word Splitting . . . . .	2
2.2	The Importance of Spaces . . . . .	3
2.3	Scripts . . . . .	3
<b>3</b>	<b>Variables and Parameters</b>	<b>3</b>
3.1	Special Parameters . . . . .	4
3.2	Environment Variables . . . . .	5
3.3	Ambiguous Names . . . . .	5
<b>4</b>	<b>Globbering</b>	<b>6</b>
<b>5</b>	<b>Expansion</b>	<b>6</b>
5.1	Expansions and Quotes . . . . .	6
5.2	Expansion Order . . . . .	7
5.3	Brace Expansion . . . . .	7
5.4	Tilde Expansion . . . . .	8
5.5	Parameter and Variable Expansion . . . . .	8
5.6	Command Substitution . . . . .	8
5.7	Arithmetic Expansion . . . . .	8
5.8	Globbering . . . . .	8
<b>6</b>	<b>Tests and Conditionals</b>	<b>8</b>
6.1	Control Operators (&& and   ) . . . . .	9
6.2	Grouping Commands . . . . .	9
6.3	Conditional Blocks (if and []) . . . . .	9
6.4	Conditional Loops (while, until and for) . . . . .	10
6.5	Choices (case and select) . . . . .	11
<b>7</b>	<b>Input and Output</b>	<b>12</b>
7.1	Command-line Arguments . . . . .	12
7.2	File Descriptors . . . . .	13
7.3	Redirection . . . . .	14
7.4	Pipes . . . . .	15
<b>8</b>	<b>Compound Commands</b>	<b>15</b>
8.1	Subshells . . . . .	15
8.2	Command Grouping . . . . .	15
8.3	Arithmetic Evaluation . . . . .	16
<b>9</b>	<b>Functions</b>	<b>16</b>
<b>10</b>	<b>Useful Commands</b>	<b>17</b>

## 1 Bash Quick Guide

Bash is an acronym for Bourne Again Shell. It is based on the Bourne shell and is mostly compatible with its features.

Shells are command interpreters. They are applications that provide users with the ability to give commands to their operating system interactively, or to execute batches of commands quickly. In no way are they required for the execution of programs; they are merely a layer between system function calls and the user.

Think of a shell as a way for you to speak to your system. Your system doesn't need it for most of its work, but it is an excellent interface between you and what your system can offer. It allows you to perform basic math, run basic

tests and execute applications. More importantly, it allows you to combine these operations and connect applications to each other to perform complex and automated tasks.

Bash is not your operating system. It is not your window manager. It is not your terminal (but it often runs inside your terminal). It does not control your mouse or keyboard. It does not configure your system, activate your screen-saver, or open your files. It's important to understand that bash is only an interface for you to execute statements (using bash syntax), either at the interactive bash prompt or via bash scripts. The things that *actually happen* are usually caused by other programs.

This guide is based on the bash guide in GreyCat's wiki<sup>1</sup> and aims to be more concise, while still being accurate. It was produced specifically for the Bash Workshop by TheAlternative.ch<sup>2</sup>.

It is published under the CC by-nc-sa 4.0 license<sup>3</sup>.

## 2 Commands and Arguments

Bash reads commands from its input, which can be either a file or your terminal. In general, each line is interpreted as a command followed by its arguments.

```
ls
touch file1 file2 file3
ls -l
rm file1 file2 file3
```

The first word is always the command that is executed. All subsequent words are given to that command as argument.

Note that *options*, such as the `-l` option in the example above, are not treated specially by bash. They are arguments like any other. It is up to the program (`ls` in the above case) to treat it as an option.

Words are delimited by whitespace (spaces or tabs). It does not matter how many spaces there are between two words. For example, try

```
echo Hello      World
```

The process of splitting a line of text into words is called *word splitting*. It is vital to be aware of it, especially when you come across expansions later on.

### 2.1 Preventing Word Splitting

Sometimes, you will want to pass arguments to commands that contain whitespace. To do so, you can use quotes:

```
touch "Filename with spaces"
```

This command creates a single file named *Filename with spaces*. The text within double quotes is protected from word splitting and hence treated as a single word.

Note that you can also use single quotes:

```
touch 'Another filename with spaces'
```

There is, however, an important difference between the two:

- Double quotes prevent **word splitting**
- Single quotes prevent **word splitting and expansion**

When you use single quotes, the quoted text will never be changed by bash. With double quotes, expansion will still happen. This doesn't make a difference in the above example, but as soon as you e.g. use variables, it becomes important.

In general, it is considered good practice to use single quotes whenever possible, and double quotes only when expansion is desired. In that sense, the last example above can be considered "more correct".

---

<sup>1</sup><http://mywiki.woledge.org/BashGuide>

<sup>2</sup>[www.thealternative.ch](http://www.thealternative.ch)

<sup>3</sup><http://creativecommons.org/licenses/by-nc-sa/4.0/>

## 2.2 The Importance of Spaces

Bash contains various keywords and built-ins that aren't immediately recognizable as commands, such as the new test command:

```
[[ -f file ]]
```

The above code tests whether a file named “file” exists in the current directory. Just like every line of bash code, it consists of a command followed by its arguments. Here, the command is `[[`, while the arguments are `-f`, `file` and `]]`.

Many programmers of other languages would write the above command like so:

```
[[ -f file]]
```

This, though, is wrong: Bash will look for a command named `[[ -f`, which doesn't exist, and issue an error message. This kind of mistake is very common for beginners. It is advisable to always use spaces after any kind of brackets in bash, even though there are cases where they are not necessary.

## 2.3 Scripts

You have probably interacted with bash through a terminal before. You would see a bash prompt, and you would issue one command after another.

Bash scripts are basically a sequence of commands stored in a file. They are read and processed in order, one after the other.

Making a script is easy. Begin by making a new file, and put this on the first line:

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

This line is called an *interpreter directive*, or more commonly, a *hashbang* or *shebang*. Your operating system uses it to determine how this file can be run. In this case, the file is to be run using `bash`, which is stored in the `/bin/` directory.

After the shebang, you can add any command that you could also use in your terminal. For example, you could add

```
echo 'Hello World'
```

and then save the file as “myscript”

You can now run the file from the terminal by typing

```
bash myscript
```

Here, you explicitly called `bash` and made it execute the script. `bash` is used as the command, while `myscript` is an argument. However, it's also possible to use `myscript` as a command directly.

To do so, you must first make it executable:

```
chmod +x myscript
```

Now that you have permission to execute this script directly, you can type

```
./myscript
```

to run it.

The `./` is required to tell bash that the executable is located in the current directory, rather than the system directory. We will come back to this in the chapter on Variables.

## 3 Variables and Parameters

Variables and parameters can be used to store strings and retrieve them later. *Variables* are the ones you create yourself, while *special parameters* are pre-set by bash. *Parameters* actually refers to both, but is often used synonymously to special parameters.

To store a string in a variable, we use the *assignment syntax*:

```
varname=vardata
```

This sets the variable `varname` to contain the string `vardata`.

Note that you cannot use spaces around the `=` sign. With the spaces, bash would assume `varname` to be a command and then pass `=` and `vardata` as arguments.

To access the string that is now stored in the variable `varname`, we have to use *parameter expansion*. This is the most common kind of expansion: A variable is replaced with its content.

If you want to print the variable's value, you can type

```
echo $varname
```

The `$` indicates that you want to use expansion on `varname`, meaning it is replaced by its content. Note that expansion happens before the command is run. Here's what happens step-by-step:

- Bash uses variable expansion, changing `echo $varname` to `echo vardata`
- Then, bash runs `echo` with `vardata` as its parameter.

The most important thing here is that **variable expansion happens before wordsplitting**. That means, if you have defined a variable like this:

```
myfile='bad song.mp3'
```

and then run the command

```
rm $myfile
```

bash will expand this to

```
rm bad song.mp3
```

Only now, word splitting occurs, and bash will call `rm` with two arguments: `bad` and `song.mp3`. If you now had a file called `song.mp3` in your current directory, that one would be deleted instead.

To prevent this from happening, you can use double quotes:

```
rm "$myfile"
```

This will be expanded to

```
rm "bad song.mp3"
```

which is what we want. In this case, you have to use double quotes, as single quotes would prevent expansion from happening altogether.

Not quoting variable and parameter expansions is a very common mistake even among advanced bash programmers. It can cause bugs that are hard to find and can be very dangerous. **Always quote your variable expansions.**

You can also use variable expansions inside the variable assignment itself. Consider this example:

```
myvariable='blah'
myvariable="$myvariable blah"
echo "$myvariable"
```

What will the output of this script be?

First, the variable `myvariable` will get the value `blah`. Then, `myvariable` is assigned to again, which overwrites its former content. The assignment contains a variable expansion, `"$myvariable blah"`. This is expanded to `"blah blah"`, and that is going to be the new value of `myvariable`. So the last command is expanded to `echo "blah blah"`, and the output of the script is `blah blah`.

### 3.1 Special Parameters

*Special parameters* are variables that are set by bash itself. Most of those variables can't be written to and they contain useful information.

Parameter Name	Usage	Description
0	"\$0"	Contains the name of the current script
1 2 3 etc.	"\$1" etc.	Contains the arguments that were passed to the current script. The number indicates the position of that argument (first, second...). These parameters are also called positional parameters.
*	"\$@"	Contains all the positional parameters. Double quoted, it expands to a single word containing them all.
@	"\$@"	Contains all the positional parameters. Double quoted, it expands to <b>several words, where each word is one parameter</b> . This is special syntax, it behaves differently from “normal” expansion in quotes. It retains the arguments exactly as they were passed to the script.
#	"\$#"	Contains the number of parameters that were passed to the script
?	"\$?"	Contains the exit code of the last executed command

## 3.2 Environment Variables

*Environment variables* are special variables that are already set when you start bash. In fact, these variables aren't specific to bash - they are available in every program that runs on your system and they can affect your system's behaviour.

You can use the command `printenv` in your terminal to display all the environment variables you have currently set. Most of them contain some system configuration, like the variable `LANG`, which designates your preferred language. Some variables, like `TERM`, `BROWSER` or `SHELL`, designate your preferred default programs (terminal, web browser and shell, respectively). These may not be set on all systems).

Some of these variables can be useful in your scripts. For example, the variable `RANDOM` gives you a different random number every time you read it.

Another important environment variable is `PATH`. It contains a bunch of file paths, separated by colons. These paths designate where your system will look for executables when you type a command. For example, if you type `grep` in your terminal, your system will search for an executable called `grep` in the directories designated in your `PATH` variable. As soon as it finds one, it will execute that. If it doesn't find it, you will get a “command not found” error message.

You can modify your environment variables, if you want. The guideline here is to only mess with those variables of which you know what they do, otherwise you might break something.

The place to modify these variables is your `~/.bash_profile` file. This file contains some bash code that is executed whenever you log in. For example, you could add the following line:

```
export BROWSER="firefox"
```

This would set your default browser to firefox. Note that on some systems, there are other settings—for example in your Desktop Environment—which can override these environment variables. You'll have to test whether this works.

Note the `export` keyword. This is a bash builtin that takes a variable definition as its argument and puts it in your *environment*. If you omit this, your new variable will just be an ordinary variable, rather than an environment variable.

## 3.3 Ambiguous Names

Say you have a variable called `name` that is declared as follows:

```
name='bat'
```

Now, you want to use this variable in order to print *batman*:

```
echo "$nameman"
```

If you try this, you will notice that it doesn't work—that is because bash will now look for a variable called `nameman`, which doesn't exist. Here's what you can do instead:

```
echo "${name}man"
```

The curly braces tell bash where the variable name ends. This allows you to add more characters at the end of a variable's content.

## 4 Globbing

*Globs* are an important bash concept—mostly for their incredible convenience. They are patterns that can be used to match filenames or other strings.

Globs are composed of normal characters and metacharacters. Metacharacters are characters that have a special meaning. These are the metacharacters that can be used in globs:

- `*`: Matches any string, including the empty string (i.e. nothing)
- `?`: Matches any single character
- `[...]`: Matches any one of the characters enclosed in the brackets

Bash sees the glob, for example `a*`. It expands this glob, by looking in the current directory and matching it against all files there. Any filenames that match the glob are gathered up and sorted, and then the list of filenames is used in place of the glob. So if you have three files `a`, `b` and `albert` in the current directory, the glob is expanded to `a albert`.

A glob always has to match the entire filename. That means `a*` will match `at` but not `bat`.

Note that globbing is special in that it happens *after word splitting*. This means you never need to worry about spaces in filenames when you use globbing, and quoting globs is not necessary. In fact, quotes will prevent globbing from happening.

## 5 Expansion

We've already seen *parameter and variable expansion*, but that's not the only kind of expansion that happens in bash. In this chapter, we'll look at all kinds of expansion that aren't covered elsewhere.

### 5.1 Expansions and Quotes

You already know that it is important to quote parameter and variable expansions, but we also told you that quoting globs—which are, in fact, just another form of expansion—is not necessary. So, which expansions need to be quoted?

The rule of thumb here is as follows:

- Always quote **parameter expansion**, **command substitution** and **arithmetic expansion**
- Never quote **brace expansion**, **tilde expansion** and **globs**

The handy thing here is: All the expansions that require quoting have a `$` in their syntax. Parameter expansion is simply a `$` followed by a parameter name. Command substitution starts with a `$(`, and arithmetic expansion starts with `$(`.

So, the rule of thumb breaks down to the following: **If there's a dollar, you probably need quotes.**

Now, what if you want to use two kinds of expansion in the same line, but one requires quotes and the other doesn't? Consider the following script:

```
prefix='my picture'
rm ~/pictures/$prefix*
```

Here, we use tilde expansion, parameter expansion and globbing in order to remove all files that start with `my picture` in the folder `/home/username/pictures/`. But because quotes prevent tilde expansion and globbing, we cannot quote the entire expression. This means that the parameter expansion, too, goes unquoted—and this is fatal, because our variable contains a space. So what should we do?

The important thing to realize here is that quoting simply prevents word splitting, but it does not actually designate something as a single string. So we can do the following:

```
prefix='my picture'
rm ~/pictures/"$prefix"
```

Only the parameter expansion is quoted, so it is protected from word splitting. But that does not automatically separate it from the rest of the string. Note that there are no spaces between "\$prefix" and ~/pictures/. Since word splitting only happens when there are spaces, the entire thing will not be split. Here's what happens, in order:

First, tilde expansion occurs:

```
rm /home/username/pictures/"$prefix"/*
```

Next, parameter expansion:

```
rm /home/username/pictures/"my picture"*
```

At this point, word splitting happens. But since the only space in our argument is in quotes, the argument remains intact.

And last, globbing:

```
rm /home/username/pictures/"my picture"001.jpg /home/username/pictures/"my picture"002.jpg
```

Now, there's one last step that happens which we didn't mention before. It's called *quote removal*. All the quotes that were needed to prevent word splitting are now ignored, which means that the arguments that are finally given to `rm` are:

- /home/username/pictures/my picture001.jpg
- /home/username/pictures/my picture002.jpg

and this is exactly what we wanted.

So, remember: Quotes don't need to be at the beginning or end of an argument, and if you use several kinds of expansion together, you can add quotes in the middle as required.

## 5.2 Expansion Order

All the kinds of expansion happen in a certain order. The order is as follows:

- Brace expansion
- Tilde expansion
- Parameter and variable expansion
- Command substitution
- Arithmetic expansion
- Word splitting
- Globbing

## 5.3 Brace Expansion

*Brace expansions* are often used in conjunction with globs, but they also have other uses. They always expand to all possible permutations of their contents. Here's an example:

```
$ echo th{e,a}n
then than
$ echo {1..9}
1 2 3 4 5 6 7 8 9
$ echo {0,1}{0..9}
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
```

Brace expansions are replaced by a list of words. They are often used in conjunction with globs to match specific files but not others. For example, if you want to delete pictures from your pictures folder with filenames IMG020.jpg through IMG039.jpg, you could use the following pattern:

```
rm IMG0{2,3}*.jpg
```

Note that we don't use quotes here. Quotes prevent brace expansion.

Brace expansion happens before globbing, so in the above example, the braces are expanded to

```
rm IMG02*.jpg IMG03*.jpg
```

We end up with two glob patterns, the first matches IMG020.jpg through IMG029.jpg, and the second matches IMG030.jpg through IMG039.jpg.

## 5.4 Tilde Expansion

You have probably already seen and used the tilde in the terminal. It is a shortcut to your home directory:

```
cd ~/files
```

This will be expanded to `cd /home/username/files`. Note that tilde expansion only happens outside of quotes, so the following won't work:

```
cd "~/files"
```

## 5.5 Parameter and Variable Expansion

*Parameter and variable expansion* is explained in the chapter on Variables and Parameters.

An important sidenote here is that Parameter expansion and Variable expansion often refer to the same thing. The official name as per the bash manual is *Parameter expansion*, but *Variable expansion* is often used instead as it is less misleading.

## 5.6 Command Substitution

*Command substitution* is a way of using a command's output inside your script. For example, let's say you want to find out where your script is executed, and then move a file to that location.

```
echo 'I am at '  
pwd
```

This script will simply print the directory it is called from. But you want to move a file to that directory, which means you need to use it as an argument to the `mv` command. For that, you need command substitution:

```
mv ~/myfile "$ ( pwd )" 
```

The `$()` introduces a command substitution. It works similarly to variable expansion, but instead of putting a variable's content, it puts a command's output in that place. In this example, `pwd` is run, and the output is (for example) `/home/username/scripts/`. Then, the entire command is expanded to

```
mv ~/myfile "/home/username/scripts/"
```

Note that with this kind of expansion, quotes are important. The path returned by `pwd` might have contained a space, in which case we need the argument to be properly quoted.

## 5.7 Arithmetic Expansion

*Arithmetic expansion* is explained in the chapter on arithmetic evaluation.

## 5.8 Globbing

*Globbing* is so important, it has a chapter of its own.

# 6 Tests and Conditionals

Every command you run in your terminal or shell script has a return value. That value is a number, and by convention, a return value of 0 means *success*, while any other number indicates an error.

You usually don't see the return value after running a command. What you do see is the command's *output*. If you want to know a command's return value, you can read it by issuing

```
echo "$?"
```

immediately after running that command.

While you often don't need to know a command's return value, it can be useful to construct conditionals and thereby achieve advanced logic.



## 6.1 Control Operators (&& and ||)

Control operators can be used to make a command's execution depend on another command's success. This concept is called *conditional execution*:

```
mkdir folder && cd folder
```

In the above example, the `&&` operator is used to connect two commands `mkdir folder` and `cd folder`. Using this connection, bash will first execute `mkdir folder`, and then execute `cd folder` *only if the first command was successful*.

```
mkdir folder || echo 'Error: could not create folder'
```

In this example, the `||` operator is used to connect the commands. Here, the second command is executed *only if the first command failed*.

It is good practice to make your own scripts return an error (i.e. something other than 0) whenever something goes wrong. To do that, you can use this construct:

```
mkdir folder || exit 1
```

The `exit` command immediately stops the execution of your script and makes it return the number you specified as an argument. In this example, your script will attempt to create a folder. If that goes wrong, it will immediately stop and return 1.

Control operators can be written more legibly by spreading them across multiple lines:

```
mkdir folder \  
|| exit 1
```

The backslash at the end of the first line makes bash ignore that line break and treat both lines as a single command plus arguments.

## 6.2 Grouping Commands

Now, what if you want to execute multiple commands if the first one fails? Going from the example above, when `mkdir folder` fails, you might want to print an error message *and* return 1.

This can be done by enclosing these two commands in single curly braces:

```
mkdir folder || { echo 'Could not create folder'; exit 1 }
```

The two commands in curly braces are treated as an unit, and if the first command fails, both will be executed.

Note that you need to include spaces between the curly braces and the commands. If there were no spaces, bash would look for a command named `{echo` and fail to find it.

There's a semicolon separating the two commands. The semicolon has the same function as a line break: it makes bash consider both parts as individual commands-plus-arguments.

The example above could be rewritten as follows:

```
mkdir folder || {  
    echo 'Could not create folder'  
    exit 1  
}
```

Here, no semicolon is required, because there is a line break between the two statements. Line breaks and semicolons can be used interchangeably.

## 6.3 Conditional Blocks (if and [])

`if` is a shell keyword. It first executes a command and looks at its return value. If it was 0 (success), it executes one list of commands, and if it was something else (failure), it executes another.

It looks like this:

```
if true
then
    echo 'It was true'
else
    echo 'It was false'
fi
```

`true` is a bash builtin command that does nothing and returns 0. `if` will run that command, see that it returned 0, and then execute the commands between the `then` and the `else`.

In many programming languages, operators such as `>`, `<` or `==` exist and can be used to compare values. In bash, operators don't exist. But since comparing values is so common, there's a special command that can do it:

```
[[ a = b ]]
```

`[[` is a command that takes as its arguments a comparison. The last argument has to be a `]]`. It is made to look like a comparison in double brackets, but it is, in fact, a command like any other. It is also called the *new test command*. (The *old test command*, often simply called *test*, exists as well, so be careful not to confuse them).

For that reason, the spaces are absolutely needed. You cannot write this:

```
[[a=b]]
```

This will make bash look for a command called `[[a=b]]`, which doesn't exist.

`[[` does not only support comparing strings. For example, `[[ -f file ]]` will test whether a file named “file” exists. Here's a list of the most common tests you can use with `[[`:

- `-e FILE`: True if file exists.
- `-f FILE`: True if file is a regular file.
- `-d FILE`: True if file is a directory.
- String operators:
  - \* `-z STRING`: True if the string is empty (its length is zero).
  - \* `-n STRING`: True if the string is not empty (its length is not zero).
  - \* `STRING = STRING`: True if the string matches the glob pattern (if you quote the glob pattern, the strings have to match exactly).
  - \* `STRING != STRING`: True if the string does not match the glob pattern (if you quote the glob pattern, the strings just have to be different).
  - \* `STRING < STRING`: True if the first string sorts before the second.
  - \* `STRING > STRING`: True if the first string sorts after the second.
- `EXPR -a EXPR`: True if both expressions are true (logical AND).
- `EXPR -o EXPR`: True if either expression is true (logical OR).
- `! EXPR`: Inverts the result of the expression (logical NOT).
- `EXPR && EXPR`: Much like the `-a` operator of test, but does not evaluate the second expression if the first already turns out to be false.
- `EXPR || EXPR`: Much like the `-o` operator of test, but does not evaluate the second expression if the first already turns out to be true.
- Numeric operators:
  - `INT -eq INT`: True if both integers are equal.
  - `INT -ne INT`: True if the integers are not equal.
  - `INT -lt INT`: True if the first integer is less than the second.
  - `INT -gt INT`: True if the first integer is greater than the second.
  - `INT -le INT`: True if the first integer is less than or equal to the second.
  - `INT -ge INT`: True if the first integer is greater than or equal to the second.

You might occasionally come across something like this:

```
[ a = b ]
```

Here, we use single brackets instead of double brackets. This is, in fact, an entirely different command, the `[` command or *old test command*. It has the same purpose—comparing things—but the `[[` command is newer, has more features, and is easier to use. We strongly recommend using `[[` over `[`.

## 6.4 Conditional Loops (while, until and for)

Loops can be used to repeat a list of commands multiple times. In bash, there are `while` loops and `for` loops.

While loops look like this:

```
while true
do
    echo 'Infinite loop'
done
```

The `while` keyword will execute the `true` command, and if that returns 0, it executes all commands between the `do` and `done`. After that, it starts over, until the `true` command returns 1 (which it never does, which is why this loop will run indefinitely).

The above example might not be immediately useful, but you could also do something like this:

```
while ping -c 1 -W 1 www.google.com
do
    echo 'Google still works!'
done
```

There's also a variation of the `while` loop, called `until`. It works similarly, except it only runs its command list when the first command *fails*:

```
until ping -c 1 -W 1 www.google.com
do
    echo 'Google isn\'t working!'
done
```

`for` loops can be used to iterate over a list of strings:

```
for var in 1 2 3
do
    echo "$var"
done
```

After the `for`, you specify a variable name. After the `in`, you list all the strings you want to iterate over.

The loop works by setting the variable you specified to all the values from the list in turn, and then executing the command list for each of them.

This is especially useful in combination with globs or brace expansions:

```
echo 'This is a list of all my files starting with f:'
for var in f*
do
    echo "$var"
done

echo 'And now I will count from 1 to 100:'
for var in {1..100}
do
    echo "$var"
done
```

## 6.5 Choices (case and select)

Sometimes, you want your script to behave differently depending on the content of a variable. This could be implemented by taking a different branch of an `if` statement for each state:

```
if [[ "$LANG" = 'en' ]]
then
    echo 'Hello!'
elif [[ "$LANG" = 'de' ]]
then
    echo 'Guten Tag!'
elif [[ "$LANG" = 'it' ]]
then
    echo 'Ciao!'
else
    echo 'I do not speak your language.'
fi
```

This is quite cumbersome to write. At the same time, constructs like this are very common. For that reason, `bash` provides a keyword to simplify it:

```
case "$LANG" in
    en)
        echo 'Hello!'
    ;;
```

```

de)
    echo 'Guten Tag!'
    ;;
it)
    echo 'Ciao!'
    ;;
*)
    echo 'I do not speak your language.'
    ;;
esac

```

Each choice of the case statement consists of a string or glob pattern, a `)`, a list of commands that is to be executed if the string matches the pattern, and two semicolons to denote the end of a list of commands.

The string after the keyword `case` is matched against each glob pattern in order. The list of commands after the first match is executed. After that, execution continues after the `esac`.

Since the string is matched against glob patterns, we can use `*` in the end to catch anything that didn't match before.

Another construct of choice is the `select` construct. It looks and works similarly to a loop, but it also presents the user with a predefined choice. You are encouraged to try running this example yourself:

```

echo 'Which one of these does not belong in the group?'
select choice in Apples Pears Crisps Lemons Kiwis
do
    if [[ "$choice" = Crisps ]]
    then
        echo 'Correct! Crisps are not fruit.'
        break
    fi
    echo 'Wrong answer. Try again.'
done

```

The syntax of the `select` construct is very similar to `for` loops. The difference is that instead of setting the variable (`choice` in this example) to each value in turn, the `select` construct lets the user choose which value is used next. This also means that a `select` construct can run indefinitely, because the user can keep selecting new choices. To avoid being trapped in it, we have to explicitly use `break`. `break` is a builtin command that makes bash jump out of the current `do` block. Execution will continue after the `done`. `break` also works in `for` and `while` loops.

As you can see in the example above, we used an `if` command inside a `select` command. All of these conditional constructs (`if`, `for`, `while`, `case` and `select`) can be nested indefinitely.

## 7 Input and Output

Input and output in bash is very flexible and, consequentially, complex. We will only look at the most widely used components.

### 7.1 Command-line Arguments

For many bash scripts, the first input we care about are the arguments given to it via the command line. As we saw in the chapter on Parameters, these arguments are contained in some *special parameters*. These are called *positional parameters*. The first parameter is referred to with `$1`, the second with `$2`, and so on. After number 9, you have to enclose the numbers in curly braces: `${10}`, `${11}` and so on.

In addition to referring to them one at a time, you may also refer to the entire set of positional parameters with the `"$@"` substitution. The double quotes here are **extremely important**. If you don't use the double quotes, each one of the positional parameters will undergo word splitting and globbing. You don't want that. By using the quotes, you tell Bash that you want to preserve each parameter as a separate word.

There are even more ways to deal with parameters. For example, it is very common for commands to accept *options*, which are single letters starting with a `-`. For example, `ls -l` calls the `ls` program with the `-l` option, which makes it output more information. Usually, multiple options can be combined, as in `ls -la`, which is equivalent to `ls -l -a`.

You might want to create your own scripts that accept some options. Bash has the so called `getopts` builtin command to parse passed options.

```
while getopts 'hlf:' opt
do
    case "$opt" in
        h|l|f)
            echo 'available options: -h -l -f [filename]'
            ;;
        f)
            file="$OPTARG"
            ;;
        l)
            list=true
            ;;
        esac
    done
shift "$(( OPTIND - 1 ))"
```

As you can see, we use `getopts` within a while loop. `getopts` will return 0 as long as there are more options remaining and something else if there are no more options. That makes it perfectly suitable for a loop.

`getopts` takes two arguments, the *optstring* and the *variable name*. The *optstring* contains all the letters that are valid options. In our example, these are `h`, `l` and `f`.

The `f` is followed by a colon. This indicates that the `f` option requires an argument. The script could for example be called like so:

```
myscript -f file.txt
```

`getopts` will set the variable that you specified as its second argument to the letter of the option it found first. If the option required an argument, the variable `OPTARG` is set to whatever the argument was. In the example above, a case statement is used to react to the different options.

There's also a special option `?`. Whenever `getopts` finds an option that is not present in the *optstring*, it sets the shell variable (`opt` in the example) to `?`. In the case statement above, that triggers the help message.

After all options are parsed, the remaining arguments are “moved” such that they are now in `$1`, `$2`... even though previously, these positional parameters were occupied by the options.

Also note the line `shift "$(( OPTIND - 1 ))"` at the end. The `shift` builtin can be used to discard command-line arguments. Its argument is a number and designates how many arguments we want to discard.

This is needed because we don't know how many options the user will pass to our script. If there are more positional parameters after all the options, we have no way of knowing at which number they start. Fortunately, `getopts` also sets the shell variable `OPTIND`, in which it stores the index of the option it's going to parse next.

So after parsing all the option, we just discard the first `OPTIND - 1` options, and the remaining arguments now start from `$1` onwards.

## 7.2 File Descriptors

*File descriptors* are the way programs refer to files, or other things that work like files (such as pipes, devices, or terminals). You can think of them as pointers that point to data locations. Through these pointers, programs can write to or read from these locations.

By default, every program has three file descriptors:

- Standard Input (stdin): File Descriptor 0
- Standard Output (stdout): File Descriptor 1
- Standard Error (stderr): File Descriptor 2

When you run a script in the terminal, then `stdin` contains everything you type in that terminal. `stdout` and `stderr` both point to the terminal, and everything that is written to these two is displayed as text in the terminal. `stdout` is where programs send their normal information, and `stderr` is where they send their error messages.

Let's make these definitions a little more concrete. Consider this example:

```
echo 'What is your name?'
read name
echo "Good day, $name. Would you like some tea?"
```

You already know `echo`. It simply prints its argument to *stdout*. Since *stdout* is connected to your terminal, you will see that message there.

`read` is a command that reads one line of text from *stdin* and stores it in a variable, which we specified to be `name`. Because *stdin* is connected to what you type in your terminal, it will let you type a line of text, and as soon as you press enter, that line will be stored in the variable.

So what about *stderr*? Consider this example:

```
ehco 'Hello!'
```

The command `ehco` does not exist. If you run this, bash will print an error message to *stderr*. Because *stderr* is connected to your terminal, you will see that message there.

## 7.3 Redirection

*Redirection* is the most basic form of input/output manipulation in bash. It is used to change the source or destination of *File descriptors*, i.e. connect them to something other than your terminal. For example, you can send a command's output to a file instead.

```
echo 'It was a dark and stormy night. Too dark to write.' > story
```

The `>` operator begins an *output redirection*. It redirects the *stdout* file descriptor of the command to the left, and connects it to a file called “story”. That means if you run this, you will not see the output of `echo`—after all, *stdout* no longer points to your terminal.

Note that `>` will just open the file you specify without checking whether it already exists first. If the file already exists, its contents will be overwritten and you will lose whatever was stored in there before. **Be careful.**

If you don't want to overwrite the existing content of a file, but rather append your output to the end of that file, you can use `>>` instead of `>`.

Now, let's look at *input redirection*. For that, we first introduce a command named `cat`. `cat` is often used to display the contents of a file, like so:

```
cat myfile
```

If `cat` is called without an argument, however, it will simply read from *stdin* and print that directly to *stdout*.

Try the following: Run `cat`, without arguments, in your terminal. Then type some characters and hit enter. Can you figure out what is happening?

*Input redirection* uses the `<` operator. It works as follows:

```
cat < story
```

The `<` operator will take a command's *stdin* file descriptor and point it to a file, “story” in this example. This means `cat` now ignores your terminal and reads from “story” instead. Note that this has the same effect as typing `cat story`.

If you want to redirect *stderr* instead of *stdout*, you can do as follows:

```
ehco 'Hello!' 2> errors
```

If you run this, you won't see any error message, even though the command `ehco` doesn't exist. That's because *stderr* is no longer connected to your terminal, but to a file called “errors” instead.

Now that you know about redirection, there is one subtlety that you have to be aware of: You can't have two file descriptors point to the same file.

If you wanted to log a command's complete output—*stdout* and *stderr*—you might be tempted to do something like this:

```
mycommand > logfile 2> logfile
```

However, this is a **bad** idea. The two file descriptors will now both point to the same file *independently*, which causes them to constantly overwrite each other's text.

If you still want to point both *stdout* and *stderr* to the same file, you can do it like this:

```
mycommand > logfile 2>&1
```

Here, we use the `>&` syntax to duplicate file descriptor 1. In this scenario, we no longer have two file descriptors pointing to one file. Instead, we have only one file descriptor that acts as both `stdout` and `stderr` at the same time.

To help remember the syntax, you can think of `&1` as “where 1 is”, and of the `2>` as “point 2 to”. The whole thing, `2>&1`, then becomes “point 2 to wherever 1 is”. This also makes it clear that `> logfile` has to come *before* `2>&1`: First you point 1 to “logfile”, and only then you can point 2 to where 1 is.

There’s also a quick way to completely get rid of a command’s output using redirections. Consider this:

```
mycommand > /dev/null 2>&1
```

`/dev/null` is a special file in your file system that you can write to, but the things you write to it are not stored. So, `mycommand`’s output is written somewhere where it’s not stored or processed in any way. It is discarded completely.

You could also leave out the `2>&1`. Then, you’d still see error messages, but discard the normal output.

## 7.4 Pipes

Now that you know how to manipulate *file descriptors* to direct output to files, it’s time to learn another type of I/O redirection.

The `|` operator can be used to connect one command’s *stdout* to another command’s *stdin*. Have a look at this:

```
echo 'This is a beautiful day!' | sed 's/beauti/wonder'
```

The `sed` command (“sed” is short for “stream editor”) is a utility that can be used to manipulate text “on the fly”. It reads text from `stdin`, edits it according to some commands, and then prints the result to `stdout`. It is very powerful. Here, we use it to replace “beauti” with “wonder”.

First, the `echo` command writes some text to its `stdout`. The `|` operator connected `echo`’s `stdout` to `sed`’s `stdin`, so everything `echo` sends there is immediately picked up by `sed`. `sed` will then edit the text and print the result to its own `stdout`. `sed`’s `stdout` is still connected to your terminal, so this is what you see.

## 8 Compound Commands

*Compound commands* is a catch-all phrase covering several different concepts. We’ve already seen `if`, `for`, `while`, `case`, `select` and the `{ }` keyword, which all fall into this category. Now we’ll look at a few more.

### 8.1 Subshells

*Subshells* can be used to encapsulate a command’s effect. If a command has undesired side effects, you can execute it in a subshell. Once the subshell command ends, all side effects will be gone.

To execute a command (or several commands) in a subshell, enclose them in parenthesis:

```
(  
    cd /tmp  
    pwd  
)  
pwd
```

The `cd` and the first `pwd` commands are executed in a subshell. All side effects in that subshell won’t affect the second `pwd` command. Changing the current directory is such a side effect—even though we use `cd` to go to the `/tmp` folder, we jump back to our original folder as soon as the subshell ends.

### 8.2 Command Grouping

You can group several commands together by enclosing them in curly braces. This makes `bash` consider them as a unit with regard to pipes, redirections and control flow:

```
{
    echo 'Logfile of my backup'
    rsync -av . /backup
    echo "Backup finished with exit code $?"
} > backup.log 2>&1
```

This redirects stdout and stderr of *all three commands* to a file called backup.log. Note that while this looks similar to subshells, it is not the same. Side effects that happen within the curly braces will still be present outside of them.

## 8.3 Arithmetic Evaluation

So far, we've only been manipulating strings in bash. Sometimes, though, it is also necessary to manipulate numbers. This is done through arithmetic evaluation.

Say you want to add the numbers 5 and 4. You might do something like this:

```
a=5+4
```

However, this will result in the variable `a` containing the string `5+4`, rather than the number 9. Instead, you should do this:

```
(( a=5+4 ))
```

The double parenthesis indicate that something arithmetic is happening. In fact, `((` is a bash keyword, much like `[[`.

`((` can also be used to do arithmetic comparison:

```
if (( 5 > 9 ))
then
    echo '5 is greater than 9'
else
    echo '5 is not greater than 9'
fi
```

It is important not to confuse `((` and `[[`. `[[` is for comparing strings (among other things), while `((` is only for comparing numbers.

There's also *arithmetic substitution*, which works similarly to *command substitution*:

```
echo "There are $(( 60 * 60 * 24 )) seconds in a day."
```

## 9 Functions

Inside a bash script, functions are very handy. They are lists of commands—much like a normal script—except they don't reside in their own file. They do however take arguments, just like scripts.

Functions can be defined like this:

```
sum() {
    echo "$1 + $2 = $(( $1 + $2 ))"
}
```

If you put this in a script file and run it, absolutely nothing will happen. The function `sum` has been defined, but it is never used.

You can use your function like any other command, but you have to define it *before* you use it:

```
sum() {
    echo "$1 + $2 = $(( $1 + $2 ))"
}
sum 1 2
sum 3 9
sum 6283 3141
```

As you can see, you can use the function `sum` multiple times, but you only need to define it once. This is useful in larger scripts, where a certain task has to be performed multiple times. Whenever you catch yourself writing the same or very similar code twice in the same script, you should consider using a function.



## 10 Useful Commands

This chapter provides an overview of useful commands that you can use in your scripts. It is nowhere near complete, and serves only to provide a brief overview. If you want to know more about a specific command, you should read its manpage.

### grep

`grep` can be used to search for a string within a file, or within the output of a command.

```
# searches logfile.txt for lines containing the word error
grep 'error' logfile.txt

# searches the directory 'folder' for files
# containing the word 'analysis'
grep 'analysis' folder/

# searches the output of 'xrandr' for lines that say 'connected'.
# only matches whole words, so 'disconnected' will not match.
xrandr | grep -w 'connected'
```

`grep` returns 0 if it finds something, and returns an error if it doesn't. This makes it useful for conditionals.

### sed

`sed` can be used to edit text “on the fly”. It uses its own scripting language to describe modifications to be made to the text, which makes it extremely powerful. Here, we provide examples for the most common usages of `sed`:

```
# replaces the first occurrence of 'find' in every line by 'replace'
sed 's/find/replace' inputfile

# replaces every occurrence of 'find' in every line by 'replace'
sed 's/find/replace/g' inputfile

# deletes the first occurrence of 'find' in every line
sed 's/find//' inputfile

# deletes every occurrence of 'find' in every line
sed 's/find//g' inputfile

# displays only the 12th line
sed '12q;d' inputfile
```

`sed` is often used in combination with pipes to format the output or get rid of unwanted characters.

### curl and wget

`curl` and `wget` are two commands that can be used to access websites or other content from the web. The difference is that `wget` will simply download the content to a file, while `curl` will output it to the console.

```
curl http://www.thealternative.ch
wget http://files.project21.ch/LinuxDays-Public/16FS-install-guide.pdf
```

### xrandr

`xrandr` can be used to manipulate your video outputs, i.e. enabling and disabling monitors or setting their screen resolution and orientation.

```
# list all available outputs and their status info
xrandr

# enables output HDMI-1
xrandr --output HDMI-1 --auto

# puts output HDMI-1 to the left of output LVDS-1
xrandr --output HDMI-1 --left-of LVDS-1

# disables output LVDS-1
xrandr --output LVDS-1 --off
```

### ImageMagick (convert)

The `convert` command makes it possible to do image processing from the commandline.

```
# scale fullHDWallpaper.jpg to specified resolution
convert fullHDWallpaper.jpg -scale 3200x1800 evenBiggerWallpaper.jpg

# "automagically" adjusts the gamma level of somePicture.jpg
convert somePicture.jpg -auto-gamma someOtherPicture.jpg

# transform image to black and white
convert colorfulPicture.jpg -monochrome blackWhitePicture.jpg
```

It is extremely powerful and has lots of options. A good resource is the official website<sup>4</sup>. It also provides examples for most options.

## notify-send

`notify-send` can be used to display a desktop notification with some custom text:

```
notify-send 'Battery warning' 'Your battery level is below 10%'
```

The first argument is the notification's title, the second is its description.

`notify-send` requires a *notification daemon* to be running, else it won't work. Most desktop environments come with a notification daemon set up and running. If you can't see your notifications, it might be that you don't have such a daemon.

## find

`find` can be used to find files in a directory structure.

```
# finds all files in the current directory and all subdirectories that end
# in .png
find -name '*.png'

# finds all files ending in .tmp and removes them. {} is replaced by the
# file's name when executing the command.
# Note that we don't use globbing here, but instead pass the * to find.
# find will then interpret the * as a wildcard.
find -name '*.tmp' -exec rm '{}'

# finds all files in the directory 'files' and prints their size and path
find 'files/' -printf '%s %p\n'
```

`find` has many options that allow you to perform arbitrary actions on the files it found or pretty-print the output.

## sort

`sort` sorts lines of text files, or lines it reads from *stdin*.

```
sort listOfNames.txt # sorts all lines in listOfNames.txt alphabetically
```

## head and tail

`head` and `tail` can be used to show the beginning or the end of a long stream of text, respectively.

```
# display the last few lines of the dmesg log
dmesg | tail

# display only the first few lines of a very long text file
head verylongtext.txt
```

## jshon

`jshon` is a very simple command-line json parser. It can read data in json format and return specific values.

Its most important options are `-e` and `-a`. `-e` extracts the value of a given key from a json array or object:

```
# Find the object with key "title" from a json object stored in the file "json"
jshon -e 'title' < 'json'
```

The `-a` option maps all remaining options across the currently selected element. It has to be combined with other options, for example the `-e` option.

```
# Find the names of all elements stored in the json object in file "json"
jshon -a -e 'name' < 'json'
```

---

<sup>4</sup><http://www.imagemagick.org>

## shuf

`shuf` randomly permutes the lines of its input, effectively *shuffling* them.

```
# Shuffle the lines of file "playlist"
shuf 'playlist'

# Get a random line from file "quotes"
shuf -n 1 'quotes'
```

## tee

`tee` takes input from *stdin* and writes it to a file:

```
sudo zypper up | tee 'updatelog'
```

Note that this is equivalent to

```
sudo zypper up > 'updatelog'
```

`tee` is useful when you want to write to a file that requires root access. Then you can do the following:

```
echo 'some configuration' | sudo tee '/etc/systemconfig'
```

A normal redirection wouldn't work in this case, as that would open the file as a normal user instead of root. **Please be careful when modifying system files.**

## sleep

`sleep` pauses the execution of your script for a number of seconds. It basically takes a number as an argument, then does nothing for that number of seconds, and then returns. It can be useful if you want to make your script do something in a loop, but want to throttle the speed a little.

```
# prints "Get back to work" once per second.
# Note that 'true' is a command that always returns 0.
while true
do
    echo 'Get back to work!'
    sleep 1
done
```

## mpplayer or mpv

`mplayer` and `mpv` are media players that can be used from the console. `mpv` is based on `mplayer` and a lot more modern, but they can do essentially the same.

```
# play file "music.mp3"
mplayer 'music.mp3'
mpv 'music.mp3'

# play file "alarm.mp3" in an infinite loop
mplayer -loop 0 'alarm.mp3'
mplayer --loop=inf 'alarm.mp3'

# play a youtube video
# this only works with mpv and requires youtube-dl to be installed
mpv 'https://www.youtube.com/watch?v=lAIGbilfpBw'
```

## xdotool

`xdotool` can simulate keypresses. With this command, you can write macros that perform actions for you. It allows you to write scripts that interact with GUI programs.

```
# press ctrl+s (in order to save a document)
xdotool key 'Control+s'

# type "hello"
xdotool type 'hello'
```