

Learn X in Y minutes

[Share this page](#)Select theme: light dark

Where X=bash

Get the code: [LearnBash.sh](#)

Bash is a name of the unix shell, which was also distributed as the shell for the GNU operating system and as the default shell on most Linux distros. Nearly all examples below can be a part of a shell script or executed directly in the shell.

[Read more here.](#)

```
#!/usr/bin/env bash
# First line of the script is the shebang which tells the
# system how to execute
# the script: https://en.wikipedia.org/wiki/Shebang_(Unix)
# As you already figured, comments start with #. Shebang is
# also a comment.

# Simple hello world example:
echo "Hello world!" # => Hello world!

# Each command starts on a new line, or after a semicolon:
echo "This is the first command"; echo "This is the second
command"
# => This is the first command
# => This is the second command

# Declaring a variable looks like this:
variable="Some string"

# But not like this:
variable = "Some string" # => returns error "variable:
command not found"
# Bash will decide that `variable` is a command it must
```

```
execute and give an error  
# because it can't be found.
```

```
# Nor like this:
```

```
variable= "Some string" # => returns error: "Some string:  
command not found"  
# Bash will decide that "Some string" is a command it must  
execute and give an  
# error because it can't be found. In this case the  
"variable=" part is seen  
# as a variable assignment valid only for the scope of the  
"Some string"  
# command.
```

```
# Using the variable:
```

```
echo "$variable" # => Some string  
echo '$variable' # => $variable  
# When you use a variable itself – assign it, export it, or  
else – you write  
# its name without $. If you want to use the variable's  
value, you should use $.  
# Note that ' (single quote) won't expand the variables!  
# You can write variable without surrounding quotes but it's  
not recommended.
```

```
# Parameter expansion ${...}:
```

```
echo "${variable}" # => Some string  
# This is a simple usage of parameter expansion such as two  
examples above.  
# Parameter expansion gets a value from a variable.  
# It "expands" or prints the value.  
# During the expansion time the value or parameter can be  
modified.  
# Below are other modifications that add onto this expansion.
```

```
# String substitution in variables:
```

```
echo "${variable/Some/A}" # => A string  
# This will substitute the first occurrence of "Some" with  
"A".
```

```
# Substring from a variable:
```

```
length=7
echo "${variable:0:length}" # => Some st
# This will return only the first 7 characters of the value
echo "${variable: -5}" # => tring
# This will return the last 5 characters (note the space
before -5).
# The space before minus is mandatory here.

# String length:
echo "${#variable}" # => 11

# Indirect expansion:
other_variable="variable"
echo "${!other_variable}" # => Some string
# This will expand the value of `other_variable`.

# The default value for variable:
echo "${foo:-"DefaultValueIfFooIsMissingOrEmpty"}"
# => DefaultValueIfFooIsMissingOrEmpty
# This works for null (foo=) and empty string (foo=""); zero
(foo=0) returns 0.
# Note that it only returns default value and doesn't change
variable value.

# Declare an array with 6 elements:
array=(one two three four five six)
# Print the first element:
echo "${array[0]}" # => "one"
# Print all elements:
echo "${array[@]}" # => "one two three four five six"
# Print the number of elements:
echo "${#array[@]}" # => "6"
# Print the number of characters in third element
echo "${#array[2]}" # => "5"
# Print 2 elements starting from fourth:
echo "${array[@]:3:2}" # => "four five"
# Print all elements each of them on new line.
for item in "${array[@]}; do
    echo "$item"
done
```

```
# Built-in variables:
# There are some useful built-in variables, like:
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
```

echo "All arguments passed to script: \$@"

echo "Script's arguments separated into different variables: \$1 \$2..."


```
# Brace Expansion {...}
# used to generate arbitrary strings:
echo {1..10} # => 1 2 3 4 5 6 7 8 9 10
echo {a..z} # => a b c d e f g h i j k l m n o p q r s t u v
w x y z
# This will output the range from the start value to the end
value.
# Note that you can't use variables here:
from=1
to=10
echo {$from..$to} # => {$from..$to}
```



```
# Now that we know how to echo and use variables,
# let's learn some of the other basics of Bash!
```



```
# Our current directory is available through the command
`pwd`.
# `pwd` stands for "print working directory".
# We can also use the built-in variable `$PWD`.
# Observe that the following are equivalent:
echo "I'm in $(pwd)" # execs `pwd` and interpolates output
echo "I'm in $PWD" # interpolates the variable
```



```
# If you get too much output in your terminal, or from a
script, the command
# `clear` clears your screen:
clear
# Ctrl-L also works for clearing output.
```



```
# Reading a value from input:
echo "What's your name?"
read name
```

```
# Note that we didn't need to declare a new variable.
echo "Hello, $name!"

# We have the usual if structure.
# Condition is true if the value of $name is not equal to the
current user's login username:
if [[ "$name" != "$USER" ]]; then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# To use && and || with if statements, you need multiple
pairs of square brackets:
read age
if [[ "$name" == "Steve" ]] && [[ "$age" -eq 15 ]]; then
    echo "This will run if $name is Steve AND $age is 15."
fi

if [[ "$name" == "Daniya" ]] || [[ "$name" == "Zach" ]]; then
    echo "This will run if $name is Daniya OR Zach."
fi

# There are other comparison operators for numbers listed
below:
# -ne - not equal
# -lt - less than
# -gt - greater than
# -le - less than or equal to
# -ge - greater than or equal to

# There is also the `=~` operator, which tests a string
against the Regex pattern:
email=me@example.com
if [[ "$email" =~ [a-z]+@[a-z]{2,}\.(com|net|org) ]]
then
    echo "Valid email!"
fi

# There is also conditional execution
echo "Always executed" || echo "Only executed if first
command fails"
```

```
# => Always executed
echo "Always executed" && echo "Only executed if first
command does NOT fail"
# => Always executed
# => Only executed if first command does NOT fail

# A single ampersand & after a command runs it in the
background. A background command's
# output is printed to the terminal, but it cannot read from
the input.
sleep 30 &
# List background jobs
jobs # => [1]+  Running                  sleep 30 &
# Bring the background job to the foreground
fg
# Ctrl-C to kill the process, or Ctrl-Z to pause it
# Resume a background process after it has been paused with
Ctrl-Z
bg
# Kill job number 2
kill %2
# %1, %2, etc. can be used for fg and bg as well

# Redefine command `ping` as alias to send only 5 packets
alias ping='ping -c 5'
# Escape the alias and use command with this name instead
\ping 192.168.1.1
# Print all aliases
alias -p

# Expressions are denoted with the following format:
echo $(( 10 + 5 )) # => 15

# Unlike other programming languages, bash is a shell so it
works in the context
# of a current directory. You can list files and directories
in the current
# directory with the ls command:
ls # Lists the files and subdirectories contained in the
current directory
```

```
# This command has options that control its execution:
ls -l # Lists every file and directory on a separate line
ls -t # Sorts the directory contents by last-modified date
(descending)
ls -R # Recursively `ls` this directory and all of its
subdirectories

# Results (stdout) of the previous command can be passed as
input (stdin) to the next command
# using a pipe |. Commands chained in this way are called a
"pipeline", and are run concurrently.
# The `grep` command filters the input with provided
patterns.
# That's how we can list .txt files in the current directory:
ls -l | grep "\.txt"

# Use `cat` to print files to stdout:
cat file.txt

# We can also read the file using `cat`:
Contents=$(cat file.txt)
# "\n" prints a new line character
# "-e" to interpret the newline escape characters as escape
characters
echo -e "START OF FILE\n$Contents\nEND OF FILE"
# => START OF FILE
# => [contents of file.txt]
# => END OF FILE

# Use `cp` to copy files or directories from one place to
another.
# `cp` creates NEW versions of the sources,
# so editing the copy won't affect the original (and vice
versa).
# Note that it will overwrite the destination if it already
exists.
cp srcFile.txt clone.txt
cp -r srcDirectory/ dst/ # recursively copy

# Look into `scp` or `sftp` if you plan on exchanging files
between computers.
```

```
# `scp` behaves very similarly to `cp`.
```

```
# `sftp` is more interactive.
```

```
# Use `mv` to move files or directories from one place to another.
```

```
# `mv` is similar to `cp`, but it deletes the source.
```

```
# `mv` is also useful for renaming files!
```

```
mv source3.txt dst.txt # sorry, 133t hackers...
```

```
# Since bash works in the context of a current directory, you might want to
```

```
# run your command in some other directory. We have cd for changing location:
```

```
cd ~      # change to home directory
```

```
cd        # also goes to home directory
```

```
cd ..     # go up one directory
```

```
          # (^^say, from /home/username/Downloads to /home/username)
```

```
cd /home/username/Documents # change to specified directory
```

```
cd ~/Documents/..          # now in home directory (if ~/Documents exists)
```

```
cd -      # change to last directory
```

```
# => /home/username/Documents
```

```
# Use subshells to work across directories
```

```
(echo "First, I'm here: $PWD") && (cd someDir; echo "Then, I'm here: $PWD")
```

```
pwd # still in first directory
```

```
# Use `mkdir` to create new directories.
```

```
mkdir myNewDir
```

```
# The `-p` flag causes new intermediate directories to be created as necessary.
```

```
mkdir -p myNewDir/with/intermediate/directories
```

```
# if the intermediate directories didn't already exist, running the above
```

```
# command without the `-p` flag would return an error
```

```
# You can redirect command input and output (stdin, stdout, and stderr)
```

```
# using "redirection operators". Unlike a pipe, which passes
```



```
output to a command,
# a redirection operator has a command's input come from a
file or stream, or
# sends its output to a file or stream.

# Read from stdin until ^EOF$ and overwrite hello.py with the
lines
# between "EOF" (which are called a "here document"):
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF
# Variables will be expanded if the first "EOF" is not quoted

# Run the hello.py Python script with various stdin, stdout,
and
# stderr redirections:
python hello.py < "input.in" # pass input.in as input to the
script

python hello.py > "output.out" # redirect output from the
script to output.out

python hello.py 2> "error.err" # redirect error output to
error.err

python hello.py > "output-and-error.log" 2>&1
# redirect both output and errors to output-and-error.log
# &1 means file descriptor 1 (stdout), so 2>&1 redirects
stderr (2) to the current
# destination of stdout (1), which has been redirected to
output-and-error.log.

python hello.py > /dev/null 2>&1
# redirect all output and errors to the black hole,
/dev/null, i.e., no output
```

```
# The output error will overwrite the file if it exists,
# if you want to append instead, use ">>":
python hello.py >> "output.out" 2>> "error.err"

# Overwrite output.out, append to error.err, and count lines:
info bash 'Basic Shell Features' 'Redirections' > output.out
2>> error.err
wc -l output.out error.err

# Run a command and print its file descriptor (e.g.
/dev/fd/123)
# see: man fd
echo <(echo "#helloworld")

# Overwrite output.out with "#helloworld":
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null

# Cleanup temporary files verbosely (add '-i' for
interactive)
# WARNING: `rm` commands cannot be undone
rm -v output.out error.err output-and-error.log
rm -r tempDir/ # recursively delete
# You can install the `trash-cli` Python package to have
`trash`
# which puts files in the system trash and doesn't delete
them directly
# see https://pypi.org/project/trash-cli/ if you want to be
careful

# Commands can be substituted within other commands using $(
):
# The following command displays the number of files and
directories in the
# current directory.
echo "There are $(ls | wc -l) items here."

# The same can be done using backticks `` but they can't be
```

```

nested -
# the preferred way is to use $( ).
echo "There are `ls | wc -l` items here."

# Bash uses a `case` statement that works similarly to switch
in Java and C++:
case "$Variable" in
    # List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";; # match everything
esac

# `for` loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done
# => 1
# => 2
# => 3

# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done
# => 1
# => 2
# => 3

# They can also be used to act on files..
# This will run the command `cat` on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command

```

```
# This will `cat` the output from `ls`.
for Output in $(ls)
do
    cat "$Output"
done

# Bash can also accept patterns, like this to `cat`
# all the Markdown files in current directory
for Output in ./*.markdown
do
    cat "$Output"
done

# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done
# => loop body here...

# You can also define functions
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    returnValue=0    # Variable values can be returned
    return $returnValue
}
# Call the function `foo` with two arguments, arg1 and arg2:
foo arg1 arg2
# => Arguments work just like script arguments: arg1 arg2
# => And: arg1 arg2...
# => This is a function
# Return values can be obtained with $?
resultValue=$?
# More than 9 arguments are also possible by using braces,
e.g. ${10}, ${11}, ...
```

```
# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
}
# Call the function `bar` with no arguments:
bar # => Another way to declare functions!

# Calling your function
foo "My name is" $Name

# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt

# prints first 10 lines of file.txt
head -n 10 file.txt

# print file.txt's lines in sorted order
sort file.txt

# report or omit repeated lines, with -d it reports them
uniq -d file.txt

# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt

# replaces every occurrence of 'okay' with 'great' in
file.txt
# (regex compatible)
sed -i 's/okay/great/g' file.txt
# be aware that this -i flag means that file.txt will be
changed
# -i or --in-place erase the input file (use --in-
place=.backup to keep a back-up)

# print to stdout all lines of file.txt which match some
regex
# The example prints lines which begin with "foo" and end in
"bar"
```

```
grep "^foo.*bar$" file.txt
```

```
# pass the option "-c" to instead print the number of lines  
matching the regex
```

```
grep -c "^foo.*bar$" file.txt
```

```
# Other useful options are:
```

```
grep -r "^foo.*bar$" someDir/ # recursively `grep`
```

```
grep -n "^foo.*bar$" file.txt # give line numbers
```

```
grep -rI "^foo.*bar$" someDir/ # recursively `grep`, but  
ignore binary files
```

```
# perform the same initial search, but filter out the lines  
containing "baz"
```

```
grep "^foo.*bar$" file.txt | grep -v "baz"
```

```
# if you literally want to search for the string,
```

```
# and not the regex, use `fgrep` (or `grep -F`)
```

```
fgrep "foobar" file.txt
```

```
# The `trap` command allows you to execute a command whenever  
your script
```

```
# receives a signal. Here, `trap` will execute `rm` if it  
receives any of the
```

```
# three listed signals.
```

```
trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM
```

```
# `sudo` is used to perform commands as the superuser
```

```
# usually it will ask interactively the password of superuser
```

```
NAME1=$(whoami)
```

```
NAME2=$(sudo whoami)
```

```
echo "Was $NAME1, then became more powerful $NAME2"
```

```
# Read Bash shell built-ins documentation with the bash
```

```
`help` built-in:
```

```
help
```

```
help help
```

```
help for
```

```
help return
```

```
help source
```

```
help .
```

```
# Read Bash manpage documentation with `man`
apropos bash
man 1 bash
man bash

# Read info documentation with `info` (`?` for help)
apropos info | grep '^info.*('
man info
info info
info 5 info

# Read bash info documentation:
info bash
info bash 'Bash Features'
info bash 6
info --apropos bash
```

Got a suggestion? A correction, perhaps? [Open an Issue](#) on the Github Repo, or make a [pull request](#) yourself!

Originally contributed by Max Yankov, and updated by [58 contributor\(s\)](#).



© 2023 [Max Yankov](#), [Darren Lin](#), [Alexandre Medeiros](#), [Denis Arh](#), [akirahirose](#), [Anton Strömkvist](#), [Rahil Momin](#), [Gregory Kielian](#), [Etan Reisner](#), [Jonathan Wang](#), [Leo Rudberg](#), [Betsy Lorton](#), [John Detter](#), [Harry Mumford-Turner](#), [Martin Nicholson](#), [Mark Grimwood](#), [Emily Grace Seville](#)

Translated by: [Dimitri Kokkonis](#)