

Bash Quick-Reference

The bash man page is very long and detailed, with the result that it can be difficult to find what you're looking for in it.

Variables

There are many shell variables that can come in handy, both interactively and in scripts. We typically write them with a dollar sign in front, because that is how they are referenced (but not set!).

<i>Variable</i>	<i>Contents</i>
\$\$	Current process (shell) PID
#!	PID of last subprocess started in the background
\$?	Return value of the last completed subprocess
\$0	Name with which the shell/script was invoked
\$1, \$2, ...	Positional parameters to the shell/script
\$#	Number of positional parameters to shell/script
@	Positional parameters, expanded as separate words
*	Positional parameters, expanded as a single word
\$HOME	The current user's home directory
\$OLDPWD	The previous working directory (see "cd -")
\$PATH	The directories searched for commands
\$PPID	The PID of the shell/script's parent process
\$PWD	The current working directory
\$RANDOM	A random int in the range [0,32767]
\$EUID	The current effective user numeric ID
\$UID	The current user numeric ID
\$USER	The current username

You set a variable with

```
my_var=123
```

Referencing variables is done with a dollar sign, but there is more than one way to do this. These are equivalent:

```
$HOME
${HOME}
```

Why would we use the longer version? Try the following:

```
for f in *; do echo $f0; done
```

Now try:

```
for f in *; do echo ${f}0; done
```

The braces give us considerable more control, and some extra features, as we'll see.

It is also possible to set variables for a single command. There are two ways to do this:

```
/usr/bin/env a=foo my_command
a=foo my_command
```

Both of these set the variable **a** to the value **foo**, but *only* for the environment seen by **my_command**. This is used frequently to override default variables without changing them:

```
JAVA_HOME=${HOME}/my_java some_java_program
LD_LIBRARY_PATH=${HOME}/build/lib my_c_program
```

By convention, script-local variables are lowercase, and more global variables (like HOME) are uppercase.

Subprocesses typically don't get passed the variables you define. To change this, you need to export the variable:

```
export PATH
```

Parameter Expansion

See the “Parameter Expansion” section of the bash manpage for more details and other expansion options.

<i>Expansion</i>	<i>Effect</i>
<code>\${#var}</code>	The length of <i>var</i>
<code>\${var:-def}</code>	<i>var</i> , if set, otherwise <i>def</i>
<code>\${var:=def}</code>	As above, but <i>var</i> will be set to <i>def</i> if not set
<code>\${var:off}</code>	Substring of <i>var</i> , beginning with character <i>off</i>
<code>\${var:ol}</code>	As above, but at most <i>l</i> characters
<code>\${var/p/s}</code>	Expand <i>var</i> , replace pattern <i>p</i> with string <i>s</i> If <i>s</i> not provided, remove the pattern # and % perform prefix and suffix matches

We can use this in a script along with variable overriding for handling script inputs symbolically, rather than positionally:

```
${target:=foo.txt}
grep foobar ${target}
```

We would call this (assuming it's called “myscript”):

```
target=/etc/hosts myscript
```

Quoting

How a language handles single- vs double-quotes varies quite a bit. Python treats them equivalently, while C only allows a single character between single-quotes. Bash works a bit differently: single-quoted strings do not have parameters expanded, while double-quoted strings do. For example, compare:

```
echo '${HOME}'
echo "${HOME}"
```

You will most often want double-quotes, but single-quotes are very useful when preparing input to another program. For example:

```
find . -name '*.txt'
```

which is equivalent to

```
find . -name \*.txt
```

Command Execution

There are multiple ways to do this. Consider an executable `foo`:

<i>Invocation</i>	<i>Effect</i>
<code>foo</code>	foo is run as a subprocess normally
<code>foo &</code>	foo is run in the background, as execution continues
<code>. foo</code>	foo is run <i>in the current shell</i>
<code>(foo)</code>	a subshell is started, and foo is run as a subproc of it
<code>`foo`</code>	foo is run as a subprocess, and its STDOUT is returned
<code>\$(foo)</code>	Same as the above

The last two are equivalent, but the `$()` form is preferable, because it is clearer and can be nested.

The dot-execution is useful for snippets of bash code which set environment variables or define functions. Think of it like an “include” statement.

The advantage of running in a subshell is that it doesn’t impact the current shell. Here’s an example:

```
for d in * # "*" expands to the contents of the current directory
do
    if [ -d $d ] # Test that $d is a directory
    then
        (cd $d; git pull origin master)
    fi
done
```

If we ran in the parent shell, we would have to make this longer:

```
cwd=$(pwd)
for d in * # "*" expands to the contents of the current directory
do
    if [ -d $d ] # Test that $d is a directory
    then
        cd $d
        git pull origin master
        cd $cwd
    fi
done
```

We could use `..` instead of `$cwd`, but then we’d have to worry about `cd $d` failing. With a subshell, we don’t need to worry about this at all.

Working with Positional Parameters

Sometimes, `$*` or `$@` are good enough:

```
foo $* # Pass all parameters to this other command
for a in $*; do ... # Loop over the parameters
```

If the parameters have different meanings, we can do the following:

```
a=$1
b=$2
```

We can make this more robust, with defaults:

```
a=${1:-foo}
b=${2:-bar}
```

We can also use `shift`, which pops the first positional parameter:

```
a=$1
shift
b=$1
shift
```

or, more compactly:

```
a=$1; shift
b=$1; shift
```

The advantage of the `$1; shift` form is that we can add more positional parameters without having to keep count. We'll see other uses later.

Mathematical Expressions

Many mathematical operations can be put in `$(())`. This will only perform integer math, however. Here's an example:

```
total=0
for thing in $*
do
    total=$(( ${total} + ${#thing} ))
done
echo ${total}
```

This will sum the lengths of the positional parameters, and print the result to STDOUT.

For floating-point math, we have to use other options (such as `awk`).

File Descriptors

There are three automatic file descriptors (in addition to any files your program opens):

<i>File</i>	<i>Descriptor</i>	<i>Meaning</i>
STDIN	0	Standard input, reading from the terminal
STDOUT	1	Standard output, writing to the terminal
STDERR	2	Standard error, writing to the terminal

The normal thing for a program to do is read from STDIN and write to STDOUT. Many programs will also write to STDERR, but if all you have is the terminal, it's hard to tell STDOUT from STDERR. However, the shell still knows, and lets us treat these differently. Here's an example. First, try:

```
grep bash /etc/*
```

Now, try this:

```
grep bash /etc/* 2>/dev/null
```

The second form told the shell that STDERR (2) should be redirected to (`>`) the special file `/dev/null`. We could also do:

```
grep bash /etc/* 2>/dev/null >bash_in_etc
```

You shouldn't see any output now, but take a look at the new file `bash_in_etc`. If unspecified, output redirection applies to STDOUT.

We can also merge STDOUT and STDERR:

```
grep bash /etc/* 2>&1
```

Here, we’ve specified the redirection target as `&1`, which means “whatever file descriptor 1 points to”.

To append, instead of overwriting, we can use `>>` instead of `>`.

We can also redirect STDIN, by using `<`:

```
wc -l <bash_in_etc # This counts the number of lines in the file
```

Here Documents

There’s a special form of input redirection, using `<<`:

```
wc <<EOFWC
this
is
a
test
EOFWC
```

The string “EOFWC” is arbitrary, but “EOF” is fairly common. We can also pass a single string as STDIN with `<<<`:

```
wc <<<"this is a test"
```

Pipelines

The Unix philosophy is that a given tool should do one thing, and if you have to do multiple things, you should compose different tools. Pipelines are the shell’s way to do this. In short:

```
foo | bar | baz
```

takes STDOUT from `foo`, redirects that to the STDIN of `bar`, and redirects `bar`’s STDOUT to `baz`’s STDIN.

You should become comfortable with this pattern, because it is one of the keys to creating powerful scripts. We can also combine with other things we’ve seen:

```
echo "grep produced $(grep bash /etc/* 2>&1 >/dev/null | wc -l) errors"
```

Control Flow

Bash has an `if/then/elif/else/fi` construction. The minimal version is `if/then/fi`, as in:

```
if $foo
then
    echo "foo"
fi
```

The full form would be:

```
if $foo
then
    echo "foo"
elif $bar
then
    echo "bar"
else
```

```

    echo "baz"
fi

```

The `if` statement uses command return codes, so you can put a command in the test, or use the `test` command (usually written `[]`):

```

grep foo /etc/hosts
have_foo=$?
grep localhost /etc/hosts
have_local=$?
if [ 0 -eq ${have_foo} ]
then
    echo "We have foo"
elif [ 0 -eq ${have_local} ]
then
    echo "We have localhost"
fi

```

See the `test` manpage for details; there are many tests you can perform, and the manpage is fairly compact.

We can also construct loops in bash, as we’ve already seen briefly. There are “for” loops and “while” loops, and they behave as you’d expect. Both have the format:

```

<for or while>
do
    # ...
done

```

A “for” statement looks like

```

for loop_var in <sequence>

```

Sequence can be something like “*”, or “a \$b \$c”, or “(ls /etc)”. If you want to iterate over numbers, you can do something like

```

for loop_var in $(seq 10)
do
    echo "foo${loop_var}"
done

```

The `while` statement takes a conditional, much like `if`. We can loop indefinitely with it:

```

while true
do
    # ...
    if $condition
    then
        break
    fi
done

```

Finally, bash has a `case` statement:

```

case ${switch_var} in
    foo) echo "foo";;
    bar|baz) echo "bar"; echo "baz";;
    *) echo "default";;
esac

```

Let’s combine this for command-line argument parsing:

```

a="foo"
b="bar"
c=""
while [ $# -gt 0 ]
do
    case $1 in
        -a) shift; a=$1; shift;;
        -b) shift; b=$1; shift;;
        *) c="$c $1"; shift;;
    esac
done

```

Functions

Defining a function:

```

function my_func {
    local a=$1
    echo $a
}

```

Functions begin with the **function** keyword, then a name, and then the body of the function, in curled braces. The **local** keyword defines a variable in the function's scope. If not used, the variable will be defined in the global scope, and hence visible outside of the function. Positional parameters are redefined for the function's scope.

Once defined, the function behaves like any other command:

```
my_func "hello"
```

You can define particularly useful functions in your `~/bashrc`, which will be executed (using `.`) whenever you start a new shell.

Aliases

Common one-liners are often nice to put into simple aliases, which are defined like:

```
alias ls='ls -FC'
```

Here are some aliases I find useful:

```

alias ls='ls -FC'
alias la='ls -A'
alias ll='ls -l'
alias ltr='ls -ltr'
alias lsd='ls -lsd'

```

These are defined in my `~/bashrc`. There's also a file called either `~/bash_profile` or `~/profile`, which is only run for a login shell (that is, only once when you first log in). When you modify one of these files, make sure you re-dot them

```

. ~/.bashrc
. ~/.bash_profile
. ~/.profile

```

Scripting

Some principles I live by:

1. If I'm going to do something more than once, I script it.
2. If I do something once, there's a good chance I'm going to have to do it again.

Most shells, bash included, treat executable text files specially. Given no other information, they run them as scripts for the current shell. *Do not assume that file extensions mean anything.* At the very least, bash does not care about the name of your file, it only cares about the content. Naming a file `foo.py` does not mean bash will treat it as a python file, for instance.

To run the script correctly, there's an easy way and a hard way. The hard way is to call the appropriate interpreter explicitly:

```
bash my_shell_script.sh
python my_python_script.py
```

The easy way is to use a convention called *shebang* (short for “hash-bang”). The shell will look at the first line of an executable ASCII file. If that line begins with a shebang, the arguments to that provide the program with which to run the script:

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

You can even provide options:

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

For python, there's a better way to call it:

```
#!/usr/bin/env python
```

What does this do? We're not actually invoking python directly. Instead, we invoke `env`, which passes the parent environment. In particular, this means it's also using the parent shell's `$PATH` variable to determine how to find python. This has a couple of advantages:

- The location of python may vary from installation to installation, but `env`'s location is always predictable.
- If you use python virtual environments, this will pick up your `virtualenv` python.

As a final note, make sure your scripts are executable! See “`chmod`” in the filesystems quick-ref for details, but 99 times out of 100, you will want to run `chmod a+x` on your script. `git` keeps track of file permissions in addition to the contents.