The classic test command

```
test <EXPRESSION>

[ <EXPRESSION> ]
```

General syntax

This command allows you to do various tests and sets its exit code to 0 (*TRUE*) or 1 (*FALSE*) whenever such a test succeeds or not. Using this exit code, it's possible to let Bash react on the result of such a test, here by using the command in an if-statement:

```
#!/bin/bash
# test if /etc/passwd exists

if test -e /etc/passwd; then
  echo "Alright man..." >&2
else
  echo "Yuck! Where is it??" >&2
  exit 1
fi
```

The syntax of the test command is relatively easy. Usually it's the command name " `test` " followed by a test type (here " `-e` " for "file exists") followed by test-type-specific values (here the filename to check, " `/etc/passwd` ").

There's a second standardized command that does exactly the same: the command " `[` " - the difference just is that it's called " `[` " and the last argument to the command must be a " `]` ": It forms " **`[ <EXPRESSION> ]`** "

Let's rewrite the above example to use it:

```
#!/bin/bash
# test if /etc/passwd exists

if [ -e /etc/passwd ]; then
  echo "Alright man..." >&2
else
  echo "Yuck! Where is it??" >&2
  exit 1
fi
```

One might **think** now that these "[" and "]" belong to the syntax of Bash's if-clause: **No they don't! It's a simple, ordinary command, still!**

Another thing you have to remember is that if the test command wants one parameter for a test, you have to give it one parameter. Let's check for some of your music files:

```
#!/bin/bash

mymusic="/data/music/Van Halen/Van Halen - Right Now.mp3"

if [ -e "$mymusic" ]; then
  echo "Let's rock" >&2
else
  echo "No music today, sorry..." >&2
  exit 1
fi
```

As you definitely noted, the filename contains spaces. Since we call a normal ordinary command ("test" or "[") the shell will word-split the expansion of the variable `mymusic` : You need to quote it when you don't want the `test` -command to complain about too many arguments for this test-type! If you didn't understand it, please read the article about words...

Please also note that the file-tests want **one filename** to test. Don't give a glob (filename-wildcards) as it can expand to many filenames ⇒ **too many arguments!**

Another common mistake is to provide too **few** arguments:

```
[ "$mystring"!="test" ]
```

This provides exactly **one** test-argument to the command. With one parameter, it defaults to the `-n` test: It tests if a provided string is empty (`FALSE`) or not (`TRUE`) - due to the lack of **spaces to separate the arguments** the shown command always ends `TRUE` !

Well, I addressed several basic rules, now let's see what the test-command can do for you. The Bash test-types can be split into several sections: **file tests**, **string tests**, **arithmetic tests**, **misc tests**. Below, the tests marked with ❗ are non-standard tests (i.e. not in SUS/POSIX/etc..).

File tests

This section probably holds the most tests, I'll list them in some logical order. Since Bash 4.1, all tests related to permissions respect ACLs, if the underlying filesystem/OS () supports them.

Operator syntax	Description
-a <FILE>	True if <FILE> exists. ❗ (not recommended, may collide with `-a` for `AND` , see below)
-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**.

Operator syntax	Description
-c <FILE>	True, if <FILE> exists and is a **character special** file.
-b <FILE>	True, if <FILE> exists and is a **block special** file.
-p <FILE>	True, if <FILE> exists and is a **named pipe** (FIFO).
-S <FILE>	True, if <FILE> exists and is a **socket** file.
-L <FILE>	True, if <FILE> exists and is a **symbolic link**.
-h <FILE>	True, if <FILE> exists and is a **symbolic link**.
-g <FILE>	True, if <FILE> exists and has **sgid bit** set.
-u <FILE>	True, if <FILE> exists and has **suid bit** set.
-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**.
-s <FILE>	True, if <FILE> exists and has size bigger than 0 (**not empty**).
-t <fd>	True, if file descriptor <fd> is open and refers to a terminal.
<FILE1> **-nt** <FILE2>	True, if <FILE1> is **newer than** <FILE2> (mtime). ❗
<FILE1> **-ot** <FILE2>	True, if <FILE1> is **older than** <FILE2> (mtime). ❗
<FILE1> **-ef** <FILE2>	True, if <FILE1> and <FILE2> refer to the **same device and inode numbers**. ❗

String tests

Operator syntax	Description
-z <STRING>	True, if <STRING> is **empty**.
-n <STRING>	True, if <STRING> is **not empty** (this is the default operation).
<STRING1> **=** <STRING2>	True, if the strings are **equal**.
<STRING1> **!=** <STRING2>	True, if the strings are **not equal**.
<STRING1> **<** <STRING2>	True if <STRING1> sorts **before** <STRING2> lexicographically (pure ASCII (), not current locale!). Remember to escape! Use `\<`
<STRING1> **>** <STRING2>	True if <STRING1> sorts **after** <STRING2> lexicographically (pure ASCII (), not current locale!). Remember to escape! Use `\>`

Arithmetic tests

Operator syntax	**Description**
<INTEGER1> **-eq** <INTEGER2>	True, if the integers are **equal**.
<INTEGER1> **-ne** <INTEGER2>	True, if the integers are **NOT equal**.
<INTEGER1> **-le** <INTEGER2>	True, if the first integer is **less than or equal** second one.
<INTEGER1> **-ge** <INTEGER2>	True, if the first integer is **greater than or equal** second one.
<INTEGER1> **-lt** <INTEGER2>	True, if the first integer is **less than** second one.
<INTEGER1> **-gt** <INTEGER2>	True, if the first integer is **greater than** second one.

Misc syntax

Operator syntax	**Description**
<TEST1> **-a** <TEST2>	True, if <TEST1> **and** <TEST2> are true (AND). Note that `-a` also may be used as a file test (see above)
<TEST1> **-o** <TEST2>	True, if either <TEST1> **or** <TEST2> is true (OR).
! <TEST>	True, if <TEST> is **false** (NOT).
(<TEST> **)**	Group a test (for precedence). **Attention:** In normal shell-usage, the "(" and ")" must be escaped; use "\(" and "\)"!
-o <OPTION_NAME>	True, if the shell option <OPTION_NAME> is set.
-v <VARIABLENAME>	True if the variable <VARIABLENAME> has been set. Use `var[n]` for array elements.
-R <VARIABLENAME>	True if the variable <VARIABLENAME> has been set and is a nameref variable (since 4.3-alpha)

Number of Arguments Rules

The `test` builtin, especially hidden under its `[` name, may seem simple but is in fact **causing a lot of trouble sometimes**. One of the difficulty is that the behaviour of `test` not only depends on its arguments but also on the **number of its arguments**.

Here are the rules taken from the manual (**Note:** This is for the command `test` , for `[` the number of arguments is calculated without the final `]` , for example `[ ]` follows the "zero arguments" rule):

- **0 arguments**
  - The expression is false.
- **1 argument**
  - The expression is true if, and only if, the argument is not null
- **2 arguments**
  - If the first argument is `!` (exclamation mark), the expression is true if, and only if, the second argument is null
  - If the first argument is one of the unary conditional operators listed above under the syntax rules, the expression is true if the unary test is true
  - If the first argument is not a valid unary conditional operator, the expression is false
- **3 arguments**
  - If the second argument is one of the binary conditional operators listed above under the syntax rules, the result of the expression is the result of the binary test using the first and third arguments as operands
  - If the first argument is `!` , the value is the negation of the two-argument test using the second and third arguments
  - If the first argument is exactly `(` and the third argument is exactly `)` , the result is the one-argument test of the second argument. Otherwise, the expression is false. The `-a` and `-o` operators are considered binary operators in this case (**Attention:** This means the operator `-a` is not a file operator in this case!)
- **4 arguments**
  - If the first argument is `!` , the result is the negation of the three-argument expression composed of the remaining arguments. Otherwise, the expression is parsed and evaluated according to precedence using the rules listed above
- **5 or more arguments**
  - The expression is parsed and evaluated according to precedence using the rules listed above

These rules may seem complex, but it's not so bad in practice. Knowing them might help you to explain some of the "unexplicable" behaviours you might encounter:

```
var=""
if [ -n $var ]; then echo "var is not empty"; fi
```

This code prints "var is not empty", even though `-n something` is supposed to be true if `$var` is not empty - **why?**

Here, as `$var` is **not quoted**, word splitting occurs and `$var` results in actually nothing (Bash removes it from the command's argument list!). So the test is in fact `[ -n ]` **and falls into the "one argument" rule**, the only argument is "-n" which is not null and so the test returns true. The solution, as usual, is to **quote the parameter expansion**: `[ -n "$var" ]` so that the test has always 2 arguments, even if the second one is the null string.

These rules also explain why, for instance, -a and -o can have several meanings.

AND and OR

The Prefered Way

The way often recommended to logically connect several tests with AND and OR is to use **several single test commands** and to **combine** them with the shell `&&` and `||` **list control operators**.

See this:

```
if [ -n "$var"] && [ -e "$var"]; then
   echo "\$var is not null and a file named $var exists!"
fi
```

The return status of AND and OR lists is the exit status of the last command executed in the list

- With `command1 && command2`, `command2` is executed if, and only if, `command1` returns an exit status of zero (true)
- With `command1 || command2`, `command2` is executed if, and only if, `command1` returns a non-zero exit status (false)

The other way: -a and -o

The logical operators AND and OR for the test-command itself are `-a` and `-o`, thus:

```
if [ -n "$var" -a -e "$var" ] ; then
   echo "\$var is not null and a file named $var exists"
fi
```

They are **not** `&&` or `||`:

```
$ if [ -n "/tmp" && -d "/tmp"]; then echo true; fi # DOES NOT WORK
bash: [: missing `]'
```

You might find the error message confusing, `[` does not find the required final `]`, because as seen above `&&` is used to write a **list of commands**. The `if` statement actually **sees two commands**:

- `[ -n "/tmp"`
- `-d "/tmp" ]`

…which **must** fail.

Why you should avoid using -a and -o

If portability is a concern

POSIX®/SUSv3 does **not** specify the behaviour of `test` in cases where there are more than 4 arguments. If you write a script that might not be executed by Bash, the behaviour might be different! [1)]

If you want the cut behaviour

Let's say, we want to check the following two things (AND):

1. if a string is null (empty)
2. if a command produced an output

Let's see:

```
if [ -z "false" -a -z "$(echo I am executed >&2)" ] ; then ...
```

⇒ The arguments are all expanded **before** `test` runs, thus the echo-command **is executed**.

```
if [ -z "false" ] && [ -z "$(echo I am not executed >&2)" ]; then...
```

⇒ Due to the nature of the `&&` list operator, the second test-command runs only if the first test-command returns true, our echo-command **is not executed**.

Note: In my opinion, `-a` and `-o` are also less readable `[pgas]`

Precedence and Parenthesis

Take care if you convert your scripts from using `-a` and `-o` to use the list way (`&&` and `||`):

- in the test-command rules, `-a` has **precedence over** `-o`
- in the shell grammar rules, `&&` and `||` have **equal precedence**

That means, **you can get different results**, depending on the manner of use:

```
$ if [ "true" ] || [ -e /does/not/exist ] && [ -e /does/not/exist ];
then echo true; else echo false; fi
false

$ if [ "true" -o -e /does/not/exist -a -e /does/not/exist ]; then  ec
ho true; else echo false;fi
true
```

As a result you have to think about it a little or add precedence control (parenthesis).

For `&&` and `||` parenthesis means (shell-ly) grouping the commands, and since `( … )` introduces a subshell we will use `{ … }` instead:

```
$ if  [ "true" ] || { [ -e /does/not/exist ]  && [ -e /does/not/exist
] ;} ; then echo true; else echo false; fi
true
```

For the test command, the precedence parenthesis are, as well, `( )`, but you need to escape or quote them, so that the shell doesn't try to interpret them:

```
$ if [ \( "true" -o -e /does/not/exist \) -a -e /does/not/exist ]; th
en  echo true; else echo false; fi
false

# equivalent, but less readable IMHO:
$ if [ '(' "true" -o -e /does/not/exist ')' -a -e /does/not/exist ];
then  echo true; else echo false; fi
false
```

NOT

As for AND and OR, there are 2 ways to negate a test with the shell keyword `!` or passing `!` as an argument to `test`.

Here `!` negates the exit status of the command `test` which is 0 (true), and the else part is executed:

```
if ! [ -d '/tmp' ]; then echo "/tmp doesn't exists"; else echo "/tmp
exists"; fi
```

Here the `test` command itself exits with status 1 (false) and the else is also executed:

```
if  [ ! -d '/tmp' ]; then echo "/tmp doesn't exists"; else echo "/tmp
exists"; fi
```

Unlike for AND and OR, both methods for NOT have an identical behaviour, at least for doing one single test.

Pitfalls summarized

In this section you will get all the mentioned (and maybe more) possible pitfalls and problems in a summary.

General

Here's the copy of a mail on bug-bash list. A user asking a question about using the test command in Bash, **he's talking about a problem, which you may have already had yourself**:

```
From: (PROTECTED)
Subject: -d option not working. . .?
Date: Tue, 11 Sep 2007 21:51:59 -0400
To: bug-bash@gnu.org

Hi All,

I've got a script that I'm trying to set up, but it keeps telling me
that  "[-d command not found".  Can someone please explain what is
wrong with this?:




#!/bin/sh

for i in $*
do
{
        if  [-d $i]
        then
                echo "$i is a directory! Yay!"
        else
                echo "$i is not a directory!"
        fi
}
done



Regards
```

See the problem regarding the used test-command (the other potential problems are not of interest here)?

```
[-d $i]
```

He simply didn't know that `test` or `[` is a normal, simple command. Well, here's the answer he got. I quote it here, because it's a well written text that addresses most of the common issues with the "classic" test command:

```
From: Bob Proulx (EMAIL PROTECTED)
Subject: Re: -d option not working. . .?
Date: Wed, 12 Sep 2007 10:32:35 -0600
To: bug-bash@gnu.org

> (QUOTED TEXT WAS REMOVED)

The shell is first and foremost a way to launch other commands.  The
syntax is simply "if" followed by a command-list, (e.g. if /some/foo;
or even if cmd1; cmd2; cmd3; then).  Plus the '( ... )' syntax is
already taken by the use of starting a subshell.

As I recall in the original shell language the file test operator was
not built-in.  It was provided by the standalone '/bin/test' command.
The result was effectively this:

  if /bin/test -d somedir

Although the full path /bin/test was never used.  I showed it that wa
y
here for emphasis that following the 'if' statement is a command lis
t.
Normally it would simply have been:

  if test -d somedir

Of course that is fine and for the best portability that style is
still the recommended way today to use the test command.  But many
people find that it looks different from other programming languages.
To make the test operator (note I mention the test operator and not
the shell language, this is a localized change not affecting the
language as a whole) look more like other programming languages the
'test' program was coded to ignore the last argument if it was a ']'.
Then a copy of the test program could be used as the '[' program.

  ...modify /bin/test to ignore ']' as last argument...
  cp /bin/test /bin/[

This allows:

  if [ -d somedir ]

Doesn't that look more normal?  People liked it and it caught on.  It
was so popular that both 'test' and '[' are now shell built-ins.  The
y
don't launch an external '/bin/test' program anymore.  But they *used
*
to launch external programs.  Therefore argument parsing is the same
as if they still did launch an external program.  This affects
argument parsing.

  it test -f *.txt
  test: too many arguments

Oops.  I have twenty .txt files and so test got one -f followed by th
```

```
e
first file followed by the remaining files.  (e.g. test -f 1.txt 2.tx
t
3.txt 4.txt)

  if test -d $file
  test: argument expected

Oops.  I meant to set file.

  file=/path/some/file
  if test -d $file

If variables such as that are not set then they wlll be expanded by
the shell before passing them to the (possibly external) command and
disappear entirely.  This is why test arguments should always be quot
ed.

  if test -d "$file"
  if [ -d "$file" ]

Actually today test is defined that if only one argument is given as
in this case "test FOO" then then test returns true if the argument i
s
non-zero in text length.  Because "-d" is non-zero length "test -d" i
s
true.  The number of arguments affects how test parses the args.  Thi
s
avoids a case where depending upon the data may look like a test
operator.

  DATA="something"
  if test "$DATA"         # true, $DATA is non-zero length

  DATA=""
  if test "$DATA"         # false, $DATA is zero length

But the problem case is how should test handle an argument that looks
like an operator?  This used to generate errors but now because it is
only one argument is defined to be the same as test -n $DATA.

  DATA="-d"
  if test "$DATA"         # true, $DATA is non-zero length
  if test -d              # true, same as previous case.

Because test and [ are possibly external commands all of the parts of
them are chosen to avoid shell metacharacters.  The Fortran operator
naming was well known at the time (e.g. .gt., .eq., etc.) and was
pressed into service for the shell test operator too.  Comming from
Fortran using -gt, -eq, etc. looked very normal.

Incorrect use generating unlikely to be intended results:

  if test 5 > 2    # true, "5" is non-zero length, creates file named
"2"
```

```
Intended use:

  if test 5 -gt 2  # true (and no shell meta characters needing quoti
ng)

Then much later, sometime in the mid 1980's, the Korn sh decided to
improve upon this situation.  A new test operator was introduced.
This one was always a shell built-in and therefore could act upon the
shell arguments directly.  This is '[[' which is a shell keyword.
(Keyword, metacharacters, builtins, all are different.)  Because the
shell processes [[ internally all arguments are known and do not need
to be quoted.

  if [[ -d $file ]]  # okay
  if [[ 5 > 2 ]]     # okay

I am sure that I am remembering a detail wrong but hopefully this is
useful as a gentle introduction and interesting anyway.

Bob
```

I hope this text protects you a bit from stepping from one pitfall into the next.

I find it very interesting and informative, that's why I quoted it here. Many thanks, Bob, also for the permission to copy the text here!

Code examples

Snipplets

Some code snipplets follow, different ways of shell reaction is used.

- **check if a variable is defined/non-NULL**
  - `test "$MYVAR"`
  - `[ "$MYVAR" ]`
  - **Note:** There are possibilities to make a difference if a variable is *undefined* or *NULL* - see Parameter Expansion - Using an alternate value
- **check if a directory exists, if not, create it**
  - `test ! -d /home/user/foo && mkdir /home/user/foo`
  - `[ ! -d /home/user/foo ] && mkdir /home/user/foo`
  - `if [ ! -d /home/user/foo ]; then mkdir /home/user/foo; fi`
- **check if minimum one parameter was given, and that one is "Hello"**
  - `test $# -ge 1 -a "$1" = "Hello" || exit 1`
  - `[ $# -ge 1 ] && [ "$1" = "Hello" ] || exit 1` (see lists description)

Listing directories

Using a for-loop to iterate through all entries of a directory, if an entry is a directory (`[ -d "$fn" ]`), print its name:

```
for fn in *; do
  [ -d "$fn" ] && echo "$fn"
done
```

See also

- Internal: conditional expression (aka "the new test command")
- Internal: the if-clause

[1] <rant>Of course, one can wonder what is the use of including the parenthesis in the specification without defining the behaviour with more than 4 arguments or how usefull are the examples with 7 or 9 arguments attached to the specification.</rant>

Discussion

PC Pete, 2012/10/06 00:06 ()

Thanks for an interesting and helpful explanation of the sources and requirements of the test operators. Even after 20 years, I'm still learning!

What I'd like to know is how to avoid one of the most common pitfalls of the file and directory tests (-f and -d in particular). This is the strange behaviour when you test a hidden file, or a file starting with '.' (not just a file that isn't readable by the permissions applied to it).

In this case, although the file can be listed and passed as an argument to both test types ("if [[-d" and "if test -d"), both tests fail when passed a 'dot' file.

So far, all the workarounds I've seen are quite cumbersome, and detract from "nice" shell scripting. Can you help with an example, or explain why these tests apparently 'fail' the way they do, and what we can do to get them to work with all files?

Thanks!

Jan Schampera, 2012/10/06 08:30 ()

Hi Pete,

can you explain more what you mean by failing with dot-files? A dot-file is "hidden" by a convention to not display it (ls, file explorers, …), not technically hidden (i.e. there is no "hidden" flag).

For the permissions thing, it's relatively easy to explain. Just give me an example of what's unclear.

Pete, 2012/10/06 14:16 (), 2012/10/13 09:31 ()

Thanks, Jan, I appreciate the help!

OK, here's the actual example of the problem I'm seeing.

I have a directory in my home directory. In that subdirectory is only one item, a directory called ".git", which contains a number of files and folders, which I want to search (without using find). This search is part of a general search, but for some reason it never seemed to search the .git folder!

I wrote a little recursive-function script that cd's into each directory, then for each file in the directory, I use the following exact code (minus some printing functions that only get executed outside the folder test):

```
#!/bin/bash

shopt -s dotglob

...

function RecurseDirs
{
    oldIFS=$IFS
    IFS=$'\n'
    for f in "$@"
    do
        if [[ -e "$PWD/$FilePattern" ]]; then
            ## go do stuff to let me know we found something...
e.g. FoundFile
        fi
        echo "Test : checking if $f is a directory..."
        if [[ -d "${f}" ]]; then
            echo Looking in "$f"...
            cd "${f}"
            RecurseDirs $(ls -A1 ".")
            cd ..
        fi
    done
    IFS=$oldIFS
}

...

RecurseDirs $(ls -A1 "$StartPath")
```

I set $StartPath to, say, '.' to begin with, and kick the script off in my home directory.

It works as expected for all the folders it finds - but when it gets to the folder containing the .git folder, although the first echo command echoes the dot folder (so it's not being hidden by the ls options or anything, that's why I set the dotglob shell option), the -d test always fails for that folder, and the second echo command never executes, even though it's an actual folder, is readable, and so on.

This happens for any ".folder" - except, if I test by using -d '.foldername' on the command line, it works!

I'm sure this is something really silly I'm misunderstanding, but I'll be darned if I can figure it out. Any ideas, or suggestions?

I thought it might be the use of the "." as the parameter to ls in the function call… but removing it had no effect on this issue, and I want to be able to extend the code and use that as another parameter later on. That's as much as I could figure out might be causing the issue.

BTW, I also get very strange errors in some folders with this script, such as "ls : option 'A' is invalid", I'm unsure if they are related, but I can't find any information in any of the shell docs about these error messages or the dot folder problem. Most frustrating… But one thing at a time!

Any help is very much appreciated! It's driving me nuts, it's a good thing not many of the files I want to find are beneath .folders!

Jan Schampera, 2012/10/13 09:41 ()

For a directory structure like

```
sub1/
sub1/subsub1
sub1/subsub2
sub2/
sub2/subsub1
sub2/subsub2
sub2/.subsub3
sub2/.subsub4
sub2/.subsub4/subsubsub1
sub2/subsub5
sub2/subsub6
```

it works here:

```
bonsai@core:~/tests/firerecurs$ ./script
Test : checking if script is a directory...
Test : checking if sub1 is a directory...
Looking in sub1...
Test : checking if subsub1 is a directory...
Looking in subsub1...
Test : checking if subsub2 is a directory...
Looking in subsub2...
Test : checking if sub2 is a directory...
Looking in sub2...
Test : checking if subsub1 is a directory...
Looking in subsub1...
Test : checking if subsub2 is a directory...
Looking in subsub2...
Test : checking if .subsub3 is a directory...
Looking in .subsub3...
Test : checking if .subsub4 is a directory...
Looking in .subsub4...
Test : checking if subsubsub1 is a directory...
Looking in subsubsub1...
Test : checking if subsub5 is a directory...
Looking in subsub5...
Test : checking if subsub6 is a directory...
Looking in subsub6...
```