

> [Linux Reviews](#) > [Beginners: Learn Linux](#) >

Bourne Shell Reference

found at Br. David Carlson, O.S.B. pages, cis.stvincent.edu/carlson/cs330/unix/bshellref - Converted to txt2tags format by xiando

This file contains short tables of commonly used items in this shell. In most cases the information applies to both the Bourne shell (sh) and the newer bash shell.

1. [Tests](#)
 - ◆ [1.1. Checking files](#)
 - ◆ [1.2. Checking strings](#)
 - ◆ [1.3. Checking numbers](#)
 - ◆ [1.4. Boolean operators](#)
 2. [I/O Redirection](#)
 3. [Shell Built-in Variables](#)
 4. [Pattern Matching](#)
 5. [Quoting](#)
 6. [Grouping](#)
 7. [Case statement](#)
 8. [Shell Arithmetic](#)
 9. [Order of Interpretation](#)
 10. [Other Shell Features](#)
-

1. Tests

Tests (for ifs and loops) are done with `[]` or with the `test` command.

1.1. Checking files

- `-r file` Check if file is readable.
- `-w file` Check if file is writable.
- `-x file` Check if we have execute access to file.
- `-f file` Check if file is an ordinary file (as opposed to a directory, a device special file, etc.)
- `-s file` Check if file has size greater than 0.
- `-d file` Check if file is a directory.
- `-e file` Check if file exists. Is true even if file is a directory.

Example:

```
if [ -s file ]
then
    such and such
fi
```

1.2. Checking strings

`s1 = s2` Check if `s1` equals `s2`. `s1 != s2` Check if `s1` is not equal to `s2`. `-z s1` Check if `s1` has size 0. `-n s1` Check if `s1` has nonzero size. `s1` Check if `s1` is not the empty string.

Example:

```
if [ $myvar = "hello" ]
then
    echo "We have a match"
fi
```

1.3. Checking numbers

Note that a shell variable could contain a string that represents a number. If you want to check the numerical value use one of the following:

<code>n1 -eq n2</code>	Check to see if <code>n1</code> equals <code>n2</code> .
<code>n1 -ne n2</code>	Check to see if <code>n1</code> is not equal to <code>n2</code> .
<code>n1 -lt n2</code>	Check to see if <code>n1</code> < <code>n2</code> .
<code>n1 -le n2</code>	Check to see if <code>n1</code> <= <code>n2</code> .
<code>n1 -gt n2</code>	Check to see if <code>n1</code> > <code>n2</code> .
<code>n1 -ge n2</code>	Check to see if <code>n1</code> >= <code>n2</code> .

Example:

```
if [ $# -gt 1 ]
then
    echo "ERROR: should have 0 or 1 command-line parameters"
fi
```

1.4. Boolean operators

<code>!</code>	<code>not</code>
<code>-a</code>	<code>and</code>
<code>-o</code>	<code>or</code>

Example:

```
if [ $num -lt 10 -o $num -gt 100 ]
then
    echo "Number $num is out of range"
elif [ ! -w $filename ]
then
    echo "Cannot write to $filename"
fi
```

Note that ifs can be nested. For example:

```
if [ $myvar = "y" ]
then
  echo "Enter count of number of items"
  read num
  if [ $num -le 0 ]
  then
    echo "Invalid count of $num was given"
  else
    ... do whatever ...
  fi
fi
```

The above example also illustrates the use of read to read a string from the keyboard and place it into a shell variable. Also note that most UNIX commands return a true (nonzero) or false (0) in the shell variable status to indicate whether they succeeded or not. This return value can be checked. At the command line echo \$status. In a shell script use something like this:

```
if grep -q shell bshellref
then
  echo "true"
else
  echo "false"
fi
```

Note that -q is the quiet version of grep. It just checks whether it is true that the string shell occurs in the file bshellref. It does not print the matching lines like grep would otherwise do.

2. I/O Redirection

pgm > file	Output of pgm is redirected to file.
pgm < file	Program pgm reads its input from file.
pgm >> file	Output of pgm is appended to file.
pgm1 pgm2	Output of pgm1 is piped into pgm2 as the input to pgm2.
n > file	Output from stream with descriptor n redirected to file.
n >> file	Output from stream with descriptor n appended to file.
n >& m	Merge output from stream n with stream m.
n <& m	Merge input from stream n with stream m.
<< tag	Standard input comes from here through next tag at start of line.

Note that file descriptor 0 is normally standard input, 1 is standard output, and 2 is standard error output.

3. Shell Built-in Variables

\$0	Name of this shell script itself.
\$1	Value of first command line parameter (similarly \$2, \$3, etc)
\$#	In a shell script, the number of command line parameters.
\$*	All of the command line parameters.
\$-	Options given to the shell.
\$?	Return the exit status of the last command.
\$\$	Process id of script (really id of the shell running the script)

4. Pattern Matching

<code>*</code>	Matches 0 or more characters.
<code>?</code>	Matches 1 character.
<code>[AaBbCc]</code>	Example: matches any 1 char from the list.
<code>[^RGB]</code>	Example: matches any 1 char not in the list.
<code>[a-g]</code>	Example: matches any 1 char from this range.

5. Quoting

<code>\c</code>	Take character <code>c</code> literally.
<code>`cmd`</code>	Run <code>cmd</code> and replace it in the line of code with its output.
<code>"whatever"</code>	Take <code>whatever</code> literally, after first interpreting <code>\$</code> , <code>`...`</code> , <code>\</code>
<code>'whatever'</code>	Take <code>whatever</code> absolutely literally.

Example:

```
match=`ls *.bak`           #Puts names of .bak files into shell variable match.
echo \*                    #Echos * to screen, not all filename as in: echo *
echo '$1$2hello'          #Writes literally $1$2hello on screen.
echo "$1$2hello"          #Writes value of parameters 1 and 2 and string hello.
```

6. Grouping

Parentheses may be used for grouping, but must be preceded by backslashes since parentheses normally have a different meaning to the shell (namely to run a command or commands in a subshell). For example, you might use:

```
if test \( -r $file1 -a -r $file2 \) -o \( -r $1 -a -r $2 \)
then
    do whatever
fi
```

7. Case statement

Here is an example that looks for a match with one of the characters a, b, c. If `$1` fails to match these, it always matches the `*` case. A case statement can also use more advanced pattern matching.

```
case "$1" in
  a) cmd1 ;;
  b) cmd2 ;;
  c) cmd3 ;;
  *) cmd4 ;;
esac
```

8. Shell Arithmetic

Bourne Shell Reference (Linux Reviews)

In the original Bourne shell arithmetic is done using the `expr` command as in: `result=`expr $1 + 2``
`result2=`expr $2 + $1 / 2`` `result=`expr $2 * 5`` (note the `\` on the `*` symbol)

With bash, an expression is normally enclosed using `[]` and can use the following operators, in order of precedence: `*` / `%` (times, divide, remainder)

1. `-` (add, subtract) `<` `>` `<=` `>=` (the obvious comparison operators) `==` `!=` (equal to, not equal to) `&&` (logical and)

(logical or)

`=` (assignment) Arithmetic is done using long integers.

Example:

```
result=${$1 + 3}
```

In this example we take the value of the first parameter, add 3, and place the sum into `result`.

9. Order of Interpretation

The bash shell carries out its various types of interpretation for each line in the following order:

brace expansion	(see a reference book)
<code>~</code> expansion	(for login ids)
parameters	(such as <code>\$1</code>)
variables	(such as <code>\$var</code>)
command substitution	(Example: <code>match=`grep DNS *`</code>)
arithmetic	(from left to right)
word splitting	
pathname expansion	(using <code>*</code> , <code>?</code> , and <code>[abc]</code>)

10. Other Shell Features

<code>\$var</code>	Value of shell variable <code>var</code> .
<code>\${var}abc</code>	Example: value of shell variable <code>var</code> with string <code>abc</code> appended.
<code>#</code>	At start of line, indicates a comment.
<code>var=value</code>	Assign the string value to shell variable <code>var</code> .
<code>cmd1 && cmd2</code>	Run <code>cmd1</code> , then if <code>cmd1</code> successful run <code>cmd2</code> , otherwise skip.
<code>cmd1 cmd2</code>	Run <code>cmd1</code> , then if <code>cmd1</code> not successful run <code>cmd2</code> , otherwise skip.
<code>cmd1; cmd2</code>	Do <code>cmd1</code> and then <code>cmd2</code> .
<code>cmd1 & cmd2</code>	Do <code>cmd1</code> , start <code>cmd2</code> without waiting for <code>cmd1</code> to finish.
<code>(cmds)</code>	Run <code>cmds</code> (commands) in a subshell.

See a good reference book for information on traps, signals, exporting of variables, functions, `eval`, `source`, etc.

- <http://cis.stvincent.edu/carlson/cs330/unix/bshellref>

> [Linux Reviews](#) > [Beginners: Learn Linux](#) >
Bourne Shell Reference