Beginner Mistakes

Here are some typical traps:

Script execution

Your perfect Bash script executes with syntax errors

If you write Bash scripts with Bash specific syntax and features, run them with Bash, and run them with Bash in native mode.

Wrong:

- no shebang
  - the interpreter used depends on the OS () implementation and current shell
  - **can** be run by calling bash with the script name as an argument, e.g. `bash myscript`
- `#!/bin/sh` shebang
  - depends on what `/bin/sh` actually is, for a Bash it means compatiblity mode, **not** native mode

See also:

- Bash startup mode: SH mode
- Bash run mode: POSIX mode

Your script named "test" doesn't execute

Give it another name. The executable `test` already exists.

In Bash it's a builtin. With other shells, it might be an executable file. Either way, it's bad name choice!

Workaround: You can call it using the pathname:

```
/home/user/bin/test
```

Globbing

Brace expansion is not globbing

The following command line is not related to globbing (filename expansion):

```
# YOU EXPECT
# -i1.vob -i2.vob -i3.vob ....

echo -i{*.vob,}

# YOU GET
# -i*.vob -i
```

Why? The brace expansion is simple text substitution. All possible text formed by the prefix, the postfix and the braces themselves are generated. In the example, these are only two: `-i*.vob` and `-i`. The filename expansion happens **after** that, so there is a chance that `-i*.vob` is expanded to a filename - if you have files like `-ihello.vob`. But it definitely doesn't do what you expected.

Please see:

- Brace expansion

Test-command

- `if [ $foo ]` …
- `if [-d $dir]` …
- …

Please see:

- The classic test command - pitfalls

Variables

Setting variables

The Dollar-Sign

There is no `$` (dollar-sign) when you reference the **name** of a variable! Bash is not PHP!

```
# THIS IS WRONG!
$myvar="Hello world!"
```

A variable name preceeded with a dollar-sign always means that the variable gets **expanded**. In the example above, it might expand to nothing (because it wasn't set), effectively resulting in…

```
="Hello world!"
```

…which **definitely is wrong**!

When you need the **name** of a variable, you write **only the name**, for example

- (as shown above) to set variables: `picture=/usr/share/images/foo.png`
- to name variables to be used by the `read` builtin command: `read picture`
- to name variables to be unset: `unset picture`

When you need the **content** of a variable, you prefix its name with **a dollar-sign**, like

- echo "The used picture is: $picture"

Whitespace

Putting spaces on either or both sides of the equal-sign (=) when assigning a value to a variable **will** fail.

```
# INCORRECT 1
example = Hello

# INCORRECT 2
example= Hello

# INCORRECT 3
example =Hello
```

The only valid form is **no spaces between the variable name and assigned value**:

```
# CORRECT 1
example=Hello

# CORRECT 2
example=" Hello"
```

Expanding (using) variables

A typical beginner's trap is quoting.

As noted above, when you want to **expand** a variable i.e. "get the content", the variable name needs to be prefixed with a dollar-sign. But, since Bash knows various ways to quote and does word-splitting, the result isn't always the same.

Let's define an example variable containing text with spaces:

```
example="Hello world"
```

Used form	result	number of words
`$example`	`Hello world`	2
`"$example"`	`Hello world`	1
`\$example`	`$example`	1
`'$example'`	`$example`	1

If you use parameter expansion, you **must** use the **name** (`PATH`) of the referenced variables/parameters. i.e. **not** (`$PATH`):

```
# WRONG!
echo "The first character of PATH is ${$PATH:0:1}"

# CORRECT
echo "The first character of PATH is ${PATH:0:1}"
```

Note that if you are using variables in arithmetic expressions, then the bare **name** is allowed:

```
((a=$a+7))         # Add 7 to a
((a = a + 7))      # Add 7 to a.  Identical to the previous command.
((a += 7))         # Add 7 to a.  Identical to the previous command.

a=$((a+7))         # POSIX-compatible version of previous code.
```

Please see:

- Words...
- Quotes and escaping
- Word splitting
- Parameter expansion

Exporting

Exporting a variable means giving **newly created** (child-)processes a copy of that variable. It does **not** copy a variable created in a child process back to the parent process. The following example does **not** work, since the variable `hello` is set in a child process (the process you execute to start that script `./script.sh`):

```
$ cat script.sh
export hello=world

$ ./script.sh
$ echo $hello
$
```

Exporting is one-way. The direction is from parent process to child process, not the reverse. The above example **will** work, when you don't execute the script, but include ("source") it:

```
$ source ./script.sh
$ echo $hello
world
$
```

In this case, the export command is of no use.

Please see:

- Bash and the process tree

Exit codes

Reacting to exit codes

If you just want to react to an exit code, regardless of its specific value, you **don't need** to use `$?` in a test command like this:

```
grep ^root: /etc/passwd >/dev/null 2>&1

if [ $? -ne 0 ]; then
  echo "root was not found - check the pub at the corner"
fi
```

This can be simplified to:

```
if ! grep ^root: /etc/passwd >/dev/null 2>&1; then
  echo "root was not found - check the pub at the corner"
fi
```

Or, simpler yet:

```
grep ^root: /etc/passwd >/dev/null 2>&1 || echo "root was not found -
check the pub at the corner"
```

If you need the specific value of `$?`, there's no other choice. But if you need only a "true/false" exit indication, there's no need for `$?`.

See also:

- Exit codes

Output vs. Return Value

It's important to remember the different ways to run a child command, and whether you want the output, the return value, or neither.

When you want to run a command (or a pipeline) and save (or print) the **output**, whether as a string or an array, you use Bash's `$(command)` syntax:

```
$(ls -l /tmp)
newvariable=$(printf "foo")
```

When you want to use the **return value** of a command, just use the command, or add ( ) to run a command or pipeline in a subshell:

```
if grep someuser /etc/passwd ; then
    # do something
fi

if ( w | grep someuser | grep sqlplus ) ; then
    # someuser is logged in and running sqlplus
fi
```

Make sure you're using the form you intended:

```
# WRONG!
if $(grep ERROR /var/log/messages) ; then
    # send alerts
fi
```

Please see:

- Bash compound commands
- Command substitution
- Grouping commands in a subshell

Discussion

U.Lickert, 2015/09/24 19:37 ()

Reacting to exit codes

You may use the specific value, and do much more by enclosing a group of commands in { }. (and still a 1-liner not pushing the rest of code out of sight)

Note the ';' after the last command, it is **necessary**

```
grep ^root: /etc/passwd >/dev/null 2>&1 || { rc=$?; echo "Tha
t search returned a '$rc' to me. Maybe time for the pub."; re
turn $rc; }
```

Bash Hackers Wiki