The basics of shell scripting

Script files

A shell script usually resides inside a file. The file can be executable, but you can call a Bash script with that filename as a parameter:

```
bash ./myfile
```

There is **no need to add a boring filename extension** like `.bash` or `.sh`. That is a holdover from UNIX®, where executables are not tagged by the extension, but by **permissions** (filemode). The file name can be any combination of legal filename characters. Adding a proper filename extension is a convention, nothing else.

```
chmod +x ./myfile
```

If the file is executable, and you want to use it by calling only the script name, the shebang must be included in the file.

The Shebang

The in-file specification of the interpreter of that file, for example:

```
#!/bin/bash
echo "Hello world..."
```

This is interpreted by the kernel [1] of your system. In general, if a file is executable, but not an executable (binary) program, and such a line is present, the program specified after `#!` is started with the scriptname and all its arguments. These two characters `#` and `!` must be **the first two bytes** in the file!

You can follow the process by using `echo` as a fake interpreter:

```
#!/bin/echo
```

We don't need a script body here, as the file will never be interpreted and executed by "`echo`". You can see what the Operating System does, it calls "`/bin/echo`" with the name of the executable file and following arguments.

```
$ /home/bash/bin/test testword hello
/home/bash/bin/test testword hello
```

The same way, with `#!/bin/bash` the shell "`/bin/bash`" is called with the script filename as an argument. It's the same as executing "`/bin/bash /home/bash/bin/test testword hello`"

If the interpreter can be specified with arguments and how long it can be is system-specific (see #!-magic (http://www.in-ulm.de/~mascheck/various/shebang/)). When Bash executes a file with a #!/bin/bash shebang, the shebang itself is ignored, since the first character is a hashmark "#", which indicates a comment. The shebang is for the operating system, not for the shell. Programs that don't ignore such lines, may not work as shebang driven interpreters.

Attention:

When the specified interpreter is unavailable or not executable (permissions), you usually get a "`bad interpreter`" error message., If you get nothing and it fails, check the shebang. Older Bash versions will respond with a "`no such file or directory`" error for a nonexistant interpreter specified by the shebang.

Additional note: When you specify `#!/bin/sh` as shebang and that's a link to a Bash, then Bash will run in POSIX® mode! See:

- Bash behaviour.

A common method is to specify a shebang like

```
#!/usr/bin/env bash
```

…which just moves the location of the potential problem to

- the `env` utility must be located in /usr/bin/
- the needed `bash` binary must be located in `PATH`

Which one you need, or whether you think which one is good, or bad, is up to you. There is no bulletproof portable way to specify an interpreter. **It's a common misconception that it solves all problems. Period.**

The standard filedescriptors

Once Initialized, every normal UNIX®-program has *at least 3 open files*:

- **stdin**: standard input
- **stdout**: standard output
- **stderr**: standard error output

Usually, they're all connected to your terminal, stdin as input file (keyboard), stdout and stderr as output files (screen). When calling such a program, the invoking shell can change these filedescriptor connections away from the terminal to any other file (see redirection). Why two different output filedescriptors? It's convention to send error messages and warnings to stderr and only program output to stdout. This enables the user to decide if they want to see nothing, only the data, only the errors, or both - and where they want to see them.

When you write a script:

- always read user-input from `stdin`
- always write diagnostic/error/warning messages to `stderr`

To learn more about the standard filedescriptors, especially about redirection and piping, see:

- An illustrated redirection tutorial

Variable names

It's good practice to use lowercase names for your variables, as shell and system-variable names are usually all in UPPERCASE. However, you should avoid naming your variables any of the following (incomplete list!):

`BASH`	`BASH_ARGC`	`BASH_ARGV`	`BASH_LINENO`	`BASH_SOURCE`	`BA`
`BASH_VERSION`	`COLUMNS`	`DIRSTACK`	`DISPLAY`	`EDITOR`	`EU`
`GROUPS`	`HISTFILE`	`HISTFILESIZE`	`HISTSIZE`	`HOME`	`HO`
`IFS`	`LANG`	`LANGUAGE`	`LC_ALL`	`LINES`	`LO`
`LS_COLORS`	`MACHTYPE`	`MAILCHECK`	`OLDPWD`	`OPTERR`	`OP`
`OSTYPE`	`PATH`	`PIPESTATUS`	`PPID`	`PROMPT_COMMAND`	`PS`
`PS2`	`PS4`	`PS3`	`PWD`	`SHELL`	`SH`
`SHLVL`	`TERM`	`UID`	`USER`	`USERNAME`	`XA`

This list is incomplete. **The safest way is to use all-lowercase variable names.**

Exit codes

Every program you start terminates with an exit code and reports it to the operating system. This exit code can be utilized by Bash. You can show it, you can act on it, you can control script flow with it. The code is a number between 0 and 255. Values from 126 to 255 are reserved for use by the shell directly, or for special purposes, like reporting a termination by a signal:

- **126**: the requested command (file) was found, but can't be executed
- **127**: command (file) not found
- **128**: according to ABS it's used to report an invalid argument to the exit builtin, but I wasn't able to verify that in the source code of Bash (see code 255)
- **128 + N**: the shell was terminated by the signal N
- **255**: wrong argument to the exit builtin (see code 128)

The lower codes 0 to 125 are not reserved and may be used for whatever the program likes to report. A value of 0 means **successful** termination, a value not 0 means **unsuccessful** termination. This behavior (== 0, != 0) is also what Bash reacts to in some flow control statements.

An example of using the exit code of the program `grep` to check if a specific user is present in /etc/passwd:

```
if grep ^root /etc/passwd; then
   echo "The user root was found"
else
   echo "The user root was not found"
fi
```

A common decision making command is " `test` " or its equivalent " `[` ". But note that, when calling test with the name " `[` ", the square brackets are not part of the shell syntax, the left bracket **is** the test command!

```
if [ "$mystring" = "Hello world" ]; then
   echo "Yeah dude, you entered the right words..."
else
   echo "Eeeek - go away..."
fi
```

Read more about the test command

A common exit code check method uses the " `||` " or " `&&` " operators. This lets you execute a command based on whether or not the previous command completed successfully:

```
grep ^root: /etc/passwd >/dev/null || echo "root was not found - chec
k the pub at the corner."
which vi && echo "Your favourite editor is installed."
```

Please, when your script exits on errors, provide a "FALSE" exit code, so others can check the script execution.

Comments

In a larger, or complex script, it's wise to comment the code. Comments can help with debugging or tests. Comments start with the # character (hashmark) and continue to the end of the line:

```
#!/bin/bash
# This is a small script to say something.
echo "Be liberal in what you accept, and conservative in what you sen
d" # say something
```

The first thing was already explained, it's the so-called shebang, for the shell, **only a comment**. The second one is a comment from the beginning of the line, the third comment starts after a valid command. All three syntactically correct.

Block commenting

To temporarily disable complete blocks of code you would normally have to prefix every line of that block with a # (hashmark) to make it a comment. There's a little trick, using the pseudo command `:` (colon) and input redirection. The `:` does nothing, it's a pseudo command, so it does not care about standard input. In the following code example, you want to test mail and logging, but not dump the database, or execute a shutdown:

```
#!/bin/bash
# Write info mails, do some tasks and bring down the system in a safe
way
echo "System halt requested" | mail -s "System halt" netadmin@exampl
e.com
logger -t SYSHALT "System halt requested"

##### The following "code block" is effectively ignored
: <<"SOMEWORD"
/etc/init.d/mydatabase clean_stop
mydatabase_dump /var/db/db1 /mnt/fsrv0/backups/db1
logger -t SYSHALT "System halt: pre-shutdown actions done, now shutti
ng down the system"
shutdown -h NOW
SOMEWORD
##### The ignored codeblock ends here
```

What happened? The `:` pseudo command was given some input by redirection (a here-document) - the pseudo command didn't care about it, effectively, the entire block was ignored.

The here-document-tag was quoted here **to avoid substitutions** in the "commented" text! Check redirection with here-documents for more

Variable scope

In Bash, the scope of user variables is generally *global*. That means, it does **not** matter whether a variable is set in the "main program" or in a "function", the variable is defined everywhere.

Compare the following *equivalent* code snippets:

```
myvariable=test
echo $myvariable
```

```
myfunction() {
  myvariable=test
}

myfunction
echo $myvariable
```

In both cases, the variable `myvariable` is set and accessible from everywhere in that script, both in functions and in the "main program".

Attention: When you set variables in a child process, for example a *subshell*, they will be set there, but you will **never** have access to them outside of that subshell. One way to create a subshell is the pipe. It's all mentioned in a small article about Bash in the processtree!

Local variables

Bash provides ways to make a variable's scope *local* to a function:

- Using the `local` keyword, or
- Using `declare` (which will *detect* when it was called from within a function and make the variable(s) local).

```
myfunc() {
local var=VALUE

# alternative, only when used INSIDE a function
declare var=VALUE

...
}
```

The *local* keyword (or declaring a variable using the `declare` command) tags a variable to be treated *completely local and separate* inside the function where it was declared:

```
foo=external

printvalue() {
local foo=internal

echo $foo
}


# this will print "external"
echo $foo

# this will print "internal"
printvalue

# this will print - again - "external"
echo $foo
```

Environment variables

The environment space is not directly related to the topic about scope, but it's worth mentioning.

Every UNIX® process has a so-called *environment*. Other items, in addition to variables, are saved there, the so-called *environment variables*. When a child process is created (in Bash e.g. by simply executing another program, say `ls` to list files), the whole environment *including the environment variables* is copied to the new process. Reading that from the other side means: **Only variables that are part of the environment are available in the child process.**

A variable can be tagged to be part of the environment using the `export` command:

```
# create a new variable and set it:
# -> This is a normal shell variable, not an environment variable!
myvariable="Hello world."

# make the variable visible to all child processes:
# -> Make it an environment variable: "export" it
export myvariable
```

Remember that the *exported* variable is a **copy**. There is no provision to "copy it back to the parent." See the article about Bash in the process tree!

[1] under specific circumstances, also by the shell itself

Discussion

scripting/basics.txt Last modified: 2019/08/30 09:07 by ersen