Skill Pill: Shell scripting

How to bash the most out of your shell

Guido Klingbeil

Okinawa Institute of Science and Technology
guido.klingbeil@oist.jp

March 23, 2016

Overview

Get the material

All the material, slides, handout, examples, are on GitHub.
Clone the repository:

```
git clone git@github.com:risingape/bash_scripting.git
```

to the machine you are working on.

Motivation

Why shell scripting when we got GUIs for everything?
Is there? Compute π with your file manager.

```
seq -f ’4/%g’ 1 2 99999 | paste -sd-+ | bc -l
```

Gives 3.14157265358979523735.
Taken from: `http://stackoverflow.com/questions/23524661/how-can-i-calculate-pi-using-bash-command`

Motivation

Why would we like to use a shell script?

- to automate tasks such as:
    - software build process (make files),
    - submit jobs to a cluster,
- to scale a task (apply the same command to 1000 files),
- makes it easy to pass arguments to commands.

The shell and the UNIX philosophie

"This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." [Doug McIlroy]

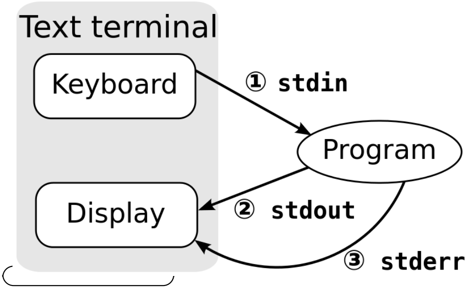
Mike Gancarz's 9 rules:

- Small is beautiful.
- Make each program do one thing well.
- Build a prototype as soon as possible.
- Choose portability over efficiency.
- Store data in flat text files.
- Use software leverage to your advantage.
- Use shell scripts to increase leverage and portability.
- Avoid captive user interfaces.
- Make every program a filter.

Standard streams: stdin, stdout, stderr

There are three standard streams or pre-connected communication channels between a computer program and its environment:

- standard input (`stdin`),
- standard output (`stdout`),
- standard error (`stderr`).



Shorter notation:

1 'represents' `stdout` and 2 `stderr`.

Redirections

- > redirection of output
- < redirect a file into a stream,
- >> appending rdirection of output,
- & denotes a file descriptor (streams are handled like files).

Examples:

- stdout to a file: `ls -l > ls-l.txt`,
- stderr to a file: `grep da * 2> errors.txt`,
- stdout to stderr: `grep da * 1>&2`,
- stderr to stdout: `grep * 2>&1`,
- stdout and stderr to a file:
  `rm -f $(find / -name core) &> /dev/null`,
- a file to stdin and stdout to a file::
  `grep da < input.txt > output.txt`.

File descriptors

Each open file gets assigned a file descriptor. You already know three of them: `stdin`, `stdout`, and `sdterr`. For opening additional files, there remain descriptors 3 to 9.
You can open a file for reading from it <, for writing to it >, or both <>.
To open and close a file we need the help of the `exec` command:

- `exec 3< test.txt` to open the file `test.txt` and attach it to the file descriptor 3,
- `exec 3>&-` to close the file attached to file descriptor 3 again.

You are know able to move araound in your file:

- `read -n 4 < &3` reads 4 characters from file descriptor 3. The next command using `&3` starts at the 5-th character.

Redirections exercise

Commands we will use:

- `exec` invokes a subprocess and replaces the current program in the current process. We will only use it to open and close files,
- `read` reads from a file. The option `-n N` reads `N` characters and places the file descriptor behind the `N`-th character,
- `echo` write to a file at the current position of the file descriptor (Hint: `echo` always print a new line. The option `-n` suppresses this).

Try to:

Write the digits 1 to 0 to a file and replace the fifth digit with the decimal point (find the solution in `redirection.sh`):

```
1234567890 -> 1234.67890
```

Pipes

Pipes | let you use the output of a program as the input of another one. Pipes are a general purpose tool to chain commands, scripts, files, and programs together by making the output of one program the input of the next:

- `ls -l | grep .txt` lists only all your text files.

What is a shell script?

A shell is a user interface to access services of an operating system's. In general, shells use either a command-line interface (CLI) or graphical user interface (GUI).

For today, a shell is a command line interpreter.

Typical tasks done by shell scripts:

- file manipulation, e.g. `cat /dev/urandom > /dev/sda`
- program execution, e.g. `rm -rf /`

Don't try these at home!

With the proper permissions `cat /dev/urandom > /dev/sda` writes (pseudo) random data to your hard drive and `rm -rf /` deletes all your files.

"Hello world"

```
# this is a comment

# The shebang, or hash bang, or ...
# set the shell to BASH - Bourne Again Shell
#!/bin/bash

# the command echo displays a line of text
echo "Hello␣world!"
```

Running the script

- `vi hello_world.sh` type in the above lines and save it,
- `chmod +x hello_world.sh` make the script executable (otherwise you will get `bash: ./hello_world.sh: Permission denied`),
- `./hello_world.sh` run the script.

Exercise: try it yourself!

Variables

There are no data types: A variable in bash can contain a number, a character, or a string of characters.
There is no need to declare a variable, just assigning a value to its reference will create it. The basic syntax is:

```
variable_name=value # no spaces around the
                    # assignment operator
```

To access the value and not the name of a variable prefix it with `$`. To execute a command and assign the output to a variable use `my_files=$(ls)`. To list all the files in the current directory:

```
my_files=$(ls)
echo $my_files
```

The syntax `$(( EXPRESSION ))` evaluates the arithmetic expression `EXPRESSION`. The output of the arithmetic expansion is guaranteed to be one word or a digit.

Special variables

- `$0` the name of the Bash script,
- `$1 - $9` the first 9 arguments to the Bash script,
- `$#` how many arguments were passed to the Bash script,
- `$@` all the arguments supplied to the Bash script,
- `$?` the exit status of the most recently run process,
- `$$` the process ID of the current script,
- `$USER` the username of the user running the script,
- `$HOSTNAME` the hostname of the machine the script is running on,
- `$SECONDS` the number of seconds since the script was started,
- `$RANDOM` returns a random number each time is it referred to,
- `$LINENO` returns the current line number in the Bash script.

Environment variables

There are already lots of global (available in all `shells`) variables defining many aspects of the computing environment. For example:

```
CPU=x86_64
QT_IM_MODULE=ibus
JAVA_BINDIR=/usr/lib64/jvm/java/bin
XDG_SESSION_TYPE=x11
INPUTRC=/home/guido/.inputrc
PWD=/home/guido
XMODIFIERS=@im=ibus
JAVA_HOME=/usr/lib64/jvm/java
LANG=en_US.utf8
GDM_LANG=en_US.utf8
PYTHONSTARTUP=/etc/pythonstart
```

Variable declaration

Variable declaration allows to limit the what kind of values can be assigned:

```
# declare option variablename [=value]
declare -r readonly_variable=1
```

The option could be:

- -r read only variable,
- -i integer variable,
- -a array variable,
- -f for funtions,
- -x declares and export to subsequent commands via the environment.

Try to:

Run the script `./variables.sh` and modify it such that it is either giving no or more error messages.

Floating point arithmetics

Bash is only able to deal with integers. However, we got everything in place to teach it how to deal with floats.

To do this we have to enlist the help of the command line calculator `bc`.

We use `echo` to print a string with our arithmetic operation to `stdout` and pipe it into `bc`. We take the value of the result and assign it to our variable `X`:

```
X=$(echo "scale=10; $RANDOM / 32767.0" | bc)
```

Note: we need to give `bc` the desired precision by setting the `scale` property.

Control structures - conditionals

The basic syntax for a conditional statement is:

```
if [ "$1" = "cool" ]          # the condition to
                              # be tested
then
    echo "Cool␣Beans"         # branch taken if the
                              # condition is true
else
    echo "Not␣Cool␣Beans"     # branch taken if the
                              # condition is false
fi
```

Further branches can be inserted by using the `elif` keyword.

Bash uses its own way for the conditional statements. Today, we will only use numerical comparisons:

- `expr1 -eq expr2` returns true if the expressions are equal,
- `expr1 -ne expr2` returns true if the expressions are not equal,
- `expr1 -gt expr2` returns true if expr1 is greater than expr2,
- `expr1 -ge expr2` returns true if expr1 is greater than or equal to expr2,
- `expr1 -lt expr2` returns true if expr1 is less than expr2,
- `expr1 -le expr2` returns true if expr1 is less than or equal to expr2.

A summary is given here `http://codewiki.wikidot.com/shell-script:if-else`.

Control structures - loops

There are `for`, `while`, and `until` loops.

The `for` loop is a little bit different from other programming languages such as C. Like in Python, it let's you iterate over a series of 'words' within a string:

```
for i in $( ls ); do     # declare a variable i to
                         # take the different values
                         # contained in $( ls )
    echo item: $i        # do something for each
                         # value $i of i
done                     # we are done with the loop
                         # and the variable i may be
                         # re-used.
```

Control structures - loops

Of course, you can also iterate C-style:

```
for i in ‘seq 1 10‘; do
    echo $i
done
```

There are two ”conditional” loops: the `while` loop and the `until` loop:

```
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done

COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

Putting it all together

Try to:

Write a simple script computing an estimation of π using a Monte Carlo method. You can find a simple and concise description of the method here: `http://www.eveandersson.com/pi/monte-carlo-circle`.

The script should do:

- loop for a given number of iterations,
- in each iteration pick 2 random numbers `X` and verb—Y— between 0 and 1,
- check if these coordinates are within the unity circle,
- if so, accept it and increment a success counter,
- compute your estimation of pi.

You will find a solution in `compute_pi.sh`.

DONE