CST8177

Scripting 2: What?

What can you do in a script?

1. A script can run any program or script, anything that's executable.

   If it's a command-line program (CLI) the script can interact with the program or act as a filter to it for the keyboard user. If it's a graphical application (GUI), the script can set its start-up arguments but cannot interact with it or mediate its interaction with the user.

   In either case, the script can examine its exit value and report on that, to the screen or to the system log as appropriate.

   The script can also intercept failures of the program, restart it, shut-down gracefully, or ask the keyboard user what to do.

What can you do in a script?

2. A script can read input, examine it, and make a decision about what to do based on it.

   The input can be from any text file (binary files are impractical) that the user has at least read access to. Or the input can be from **`stdin`**, directly from the keyboard user.

   There are, as is usual in Linux/Unix, several ways to actually get the input, and selection of the method may make a difference to the operation of the script. There is a read service built-into the bash shell which is often used. Shell (**`sh`**) is often more complicated to use, but it's still possible.

What can you do in a script?

3. A script has many ways to write output.

   It can write to any text file (again, binary is impractical) using redirection either from another command or program or the **`echo`** and **`printf`** statements. This includes, of course, **`stdout`** and **`stderr`** as long as the script is running in the foreground.

   A script may also write to the system log files using the **`logger`** command. This is most useful for a script running in the background or as a daemon, when there is no access to any sort of console or terminal.

   Many regular commands also have a way to enable or disable normal output. As an example, grep has the **`-q`**, **`--quiet`**, and **`--silent`** options for **`stdout`**, as well as the **`-s`** or **`--no-messages`** options for most of **`stderr`**. For some commands and situations, redirecting output to **`/dev/null`** must be used.

What can you do in a script?

4. A script can calculate numbers using normal BEDMAS priorities (brackets, exponentiation, division, multiplication, addition, subtraction) following the rules for integer numbers *.

   You can use both constants and variables, including system variables and command-line arguments.

   You can manipulate strings and sub-strings with simple parameters, converting freely between strings and numbers. You can combine these with your own configuration files (and those of other programs) and use other techniques to set default values.

   * Integer arithmetic has no fractional values. Any fractional portion created is simply discarded at once. Thus **`( ( 1 / 2  ) * 2 )`** gives a result of **`0`**, not **`1`**.

What can you do in a script?

5. A script can control the sequence of execution of its own statements.

   It can compare numbers or strings, test file and directory status, search for strings and regexps, using the result to decide which statements to execute and which to skip.

   A script can process a group of commands repeatedly, in one of several varieties of loops. It can do this by a numeric count, by the result of another command, or by a condition or event.

   It can handle different cases for values, using different statements for each individual case.

   A script can even trap signals and choose how to proceed, ignoring any signal except SIGKILL if that's what is required.

What can you do in a script?

6. A script can decide when to terminate itself; it does not have to run through to the end.

   The termination can be early, in response to an error, a failure, or a detected condition. The termination can be at the end of processing input from a file or from the keyboard, or from a list of command-line arguments. It can even decide to run through to the end, if that is what's required.

   The termination normally has an exit value that can be tested by subsequent statements, just as a script can test the termination status of any command or program.

Sample Script

```
#! /bin/bash
# create variables with initial values
declare n=$#
declare count=0
# show how many command arguments there are
echo Number of arguments: $n
# for each command-line argument, repeat
while (( $count <= $n ))
do
   # show the arg number and its |value|
   eval "echo arg $count '|'\${$count}'|'"
   # count each argument processed
   let "count += 1"
# end loop
done
echo all arguments evaluated
exit 0
```