INFN
Istituto Nazionale di Fisica Nucleare

Lecture 1: bash scripting

Physical Sensors and Systems for Biomedical Imaging - AI4ST

Ian Postuma

10 October 2023

Tecnologo @ INFN-PV - ian.postuma@pv.infn.it

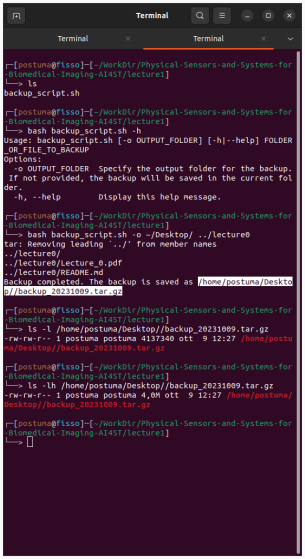
BASH

THE BOURNE-AGAIN SHELL

Bash scripting is used to automate tasks on Linux systems, such as:

- Managing files and directories
- Processing data
- Running programs
- Creating custom workflows

Bash scripts are written in **plain text files** and can be executed using the Bash shell. To write a Bash script, you simply create a new text file and start typing your commands. Once you are finished, you can save the file with a .sh extension. **To run the script, you can simply type bash followed by the name of the script**.



BASH Variables

In Bash, you can declare variables without specifying a data type. Variables can store various types of data, such as strings, numbers, or arrays. To assign a value to a variable, you use the = operator with no spaces on either side. For example: **variable_name=value**. To access the value stored in a variable, you prefix the variable name with a $ symbol, like this: **$variable_name**. Variable names are case-sensitive and can consist of letters, numbers, and underscores but must start with a letter.

```
name="John"
age=30
echo "$name is $age years old"
```

Control flow structures

Bash supports various control flow structures, including **if**, **else**, **elif**, **for**, **while**, and **case**. if statements are used for conditional branching. for and while loops allow you to perform repetitive tasks.

```
if [ condition1 ]; then
    # code to execute if the condition1 is true
elif [ condition2 ]; then
    # code to execute if the condition2 is true
else
    # code to execute if all the conditions are false
fi
```

functions

Functions in Bash are defined using the function keyword or simply by providing a name and parentheses. Functions can take parameters and return values.

```
function greet {
    echo "Hello, $1!"
}
```

using the function:

```
greet "Alice"
```

Input and Output

Bash scripts can read input from users or files and display output. The **read** command is used to read user input.

```
echo "What's your name?"
read name
echo $name
```

Standard input (stdin) can be redirected using **<**, and standard output (stdout) can be redirected using **>**.

```
echo "Hello, world!" > output.txt
```

Command-line arguments

In Bash scripting, you can use command-line arguments to pass information to your script when you run it. These arguments allow you to **customize the behavior of your script without modifying its code**. Command-line arguments are accessed in your Bash script using special variables:

- **$0**: The name of the script itself.
- **$1**, **$2**, **$3**, ...: Represent the first, second, third, and so on, arguments passed to the script.
- **$#**: represents the number of arguments passed to the script.
- **$@** represents all arguments as a list.

Command-line arguments

For example, if you run your script with

```
./myscript.sh arg1 arg2
```

then:

- $0 will be ./myscript.sh.
- $1 will be arg1.
- $2 will be arg2.

Simple backup script:

The **#!** at the beginning of a Bash script is called a **shebang** or hashbang. It's a special line used to indicate the interpreter that should be used to execute the script. In this case, **#!/bin/bash specifies that the script should be run using the Bash shell**.

This shebang line is essential for making a script executable and specifying which interpreter should be used to interpret the script's commands. Without it, the script might be executed by a different shell or interpreter depending on the system's default settings, which could lead to unexpected behavior.

To **make the script executable** use:

```
chmod +x backup_script.sh
```

$ backup_script.sh

```bash
#!/bin/bash

# Function to display help message
display_help() {
    echo "Usage: $0 [-o OUTPUT_FOLDER] [-h|--help] FOLDER_OR_FILE_TO_BACKUP"
    echo "Options:"
    echo "  -o OUTPUT_FOLDER  Specify the output folder for the backup. If not provided, the backup will be saved in the current folder."
    echo "  -h, --help        Display this help message."
    exit 1
}

# Initialize variables with default values
output_folder="."
date_today=$(date +%Y%m%d)
backup_name="backup_$date_today.tar.gz"

# Parse command-line options
while [[ $# -gt 0 ]]; do
    case $1 in
        -o)
            shift
            output_folder="$1"
            shift
            ;;
        -h|--help)
            display_help
            ;;
        *)
            source_path="$1"
            shift
            ;;
    esac
done
```

Simple backup script:

The **display_help()** function in the Bash script is responsible for displaying a help message to the user when they invoke the script with the -h option. Here's what the display_help() function does:

- It prints usage instructions and a **brief explanation** of the script's options and parameters.
- It uses echo statements to display this information in a **user-friendly format**.
- It exits the script with a status code of 1 to indicate that there was an error in the script's usage or that the user requested help. **Exiting with a non-zero status code is a common practice to signal that something went wrong** or that the script should not proceed with its regular execution.

Check the exit status of a command or script with:

```
echo $?
```

$ backup_script.sh

```bash
#!/bin/bash

# Function to display help message
display_help() {
    echo "Usage: $0 [-o OUTPUT_FOLDER] [-h|--help] FOLDER_OR_FILE_TO_BACKUP"
    echo "Options:"
    echo "  -o OUTPUT_FOLDER  Specify the output folder for the backup. If not provided, the backup will be saved in the current folder."
    echo "  -h, --help        Display this help message."
    exit 1
}

# Initialize variables with default values
output_folder="."
date_today=$(date +%Y%m%d)
backup_name="backup_$date_today.tar.gz"

# Parse command-line options
while [[ $# -gt 0 ]]; do
    case $1 in
        -o)
            shift
            output_folder="$1"
            shift
            ;;
        -h|--help)
            display_help
            ;;
        *)
            source_path="$1"
            shift
            ;;
    esac
done
```

Simple backup script:

```
output_folder="."
```

This line initializes a variable named output_folder and assigns it the value ". ". In this context, ". " represents the current directory. This variable is used to store the path to the folder where the backup will be saved. By default, it's set to the current directory.

```
date_today=$(date +%Y%m%d)
```

This line initializes a variable named date_today by running the date command with a specific format. The date command with %Y%m%d format retrieves the current date in the "YearMonthDay" format (e.g., 20231010 for October 10, 2023). The result is assigned to the date_today variable, representing today's date.

```
backup_name="backup_$date_today.tar.gz"
```

This line initializes a variable named backup_name. It combines the string "backup_" with the value of the date_today variable (which is today's date in the format specified earlier) and appends .tar.gz to create a backup file name in the format "backup_YYYYMMDD.tar.gz." This variable stores the name of the backup file that will be created.

$ backup_script.sh

```
#!/bin/bash

# Function to display help message
display_help() {
    echo "Usage: $0 [-o OUTPUT_FOLDER] [-h|--help] FOLDER_OR_FILE_TO_BACKUP"
    echo "Options:"
    echo "  -o OUTPUT_FOLDER  Specify the output folder for the backup. If not provided, the backup will be saved in the current folder."
    echo "  -h, --help        Display this help message."
    exit 1
}

# Initialize variables with default values
output_folder="."
date_today=$(date +%Y%m%d)
backup_name="backup_$date_today.tar.gz"

# Parse command-line options
while [[ $# -gt 0 ]]; do
    case $1 in
        -o)
            shift
            output_folder="$1"
            shift
            ;;
        -h|--help)
            display_help
            ;;
        *)
            source_path="$1"
            shift
            ;;
    esac
done
```

Simple backup script:

The while loop iterates through the command-line arguments, and the case statement is used to process and assign values to variables based on the provided options, such as -o for specifying an output folder or -h for displaying help.

`$#`

Is the number of command line arguments.

`-gt`

Means greater than.

`$1`

Current command line argument.

`shift`

Shifts to the next argument.

`;;`

symbol indicating the end of each case.

`*)`

wildcard pattern that matches anything.

$ backup_script.sh

```bash
#!/bin/bash

# Function to display help message
display_help() {
    echo "Usage: $0 [-o OUTPUT_FOLDER] [-h|--help] FOLDER_OR_FILE_TO_BACKUP"
    echo "Options:"
    echo "  -o OUTPUT_FOLDER  Specify the output folder for the backup. If not provided, the backup will be saved in the current folder."
    echo "  -h, --help        Display this help message."
    exit 1
}

# Initialize variables with default values
output_folder="."
date_today=$(date +%Y%m%d)
backup_name="backup_$date_today.tar.gz"

# Parse command-line options
while [[ $# -gt 0 ]]; do
    case $1 in
        -o)
            shift
            output_folder="$1"
            shift
            ;;
        -h|--help)
            display_help
            ;;
        *)
            source_path="$1"
            shift
            ;;
    esac
done
```

Simple backup script:

`-z "$source_path$"`

Check if the source_path variable if defined.

`! -e "$source_path$"`

Checks if the source_path is not a path or directory.

`backup_path='$output_folder/$backup_name'`

Combines the output_folder with the backup_name and stores the output in the backup_path variable.

`tar -czvf ...`

Compresses the file or folder the user wants to compress and it saves the results to the output folder.

```
34
35  # Check if a source path is provided
36  if [ -z "$source_path" ]; then
37      echo "Error: Please provide a folder or file to backup."
38      display_help
39  fi
40
41  # Check if the source path exists
42  if [ ! -e "$source_path" ]; then
43      echo "Error: The specified source folder or file does not exist."
44      exit 1
45  fi
46
47  # Create the backup
48  backup_path="$output_folder/$backup_name"
49  tar -czvf "$backup_path" "$source_path"
50
51  echo "Backup completed. The backup is saved as $backup_path"
52
```

Image Manipulation with ImageMagick

ImageMagick is a command-line tool for manipulating images. It provides a wide range of functions for image editing and conversion, such as:

Resize: To resize an image, you can use the convert command followed by the input image, desired dimensions, and the output image name.

```
convert input.jpg -resize 800x600 output.jpg
```

Crop: Cropping can be achieved using the convert command with the -crop option. Specify the dimensions and position for cropping.

```
convert input.jpg -crop 400x400+100+100 output.jpg
```

Convert to Different Formats: You can convert an image to a different format using the convert command with the desired output format.

```
convert input.png output.jpg
```

ImageMagick also supports applying convolutional filters to images. Filters can be used for various image enhancements and effects. Here's an example of applying a Gaussian blur:

```
convert input.jpg -gaussian-blur 0x5 output.jpg
```

You can explore various filters and their effects in the ImageMagick documentation.

Image Manipulation with ImageMagick

A more advance use of ImageMagick can be done by creating a custom kernel (filter) in a text file and then use the **-convolve** option to apply this kernel to an image.

```
# simple edge detection
1 0 1
0 1 0
1 0 1
```

use the **convert** command along with the **-convolve** option to apply the custom kernel to an image.

```
convert input.jpg -define convolve:scale='!' -convolve 'cat custom_kernel.txt' output.jpg
```