livelessons
video instruction from technology experts

Bash Scripting in 4 Hours

Introduction

About your Instructor

- Sander van Vugt
- mail@sandervanvugt.nl

Poll Question 1

How would you rate your level of Linux exertise?

- none
- limited
- comfortable
- advanced
- expert

Poll Question 2

What is your primary operating system?

- Linux
- Windows
- Mac
- Other

Poll Question 3

Where are you from?

- India
- Asia
- Europe
- Netherlands
- North/Central America
- South America
- Africa
- Australia/Pacific

Agenda

- Getting Familiar with Bash
- Creating your First Shell Scripts
- Working with Variables and Arguments
- Transforming Input
- Using If then else
- Using Conditionals and Loops

Using Course Resources

- All demo scripts in the course are available at https://github.com/sandervanvugt/bash-scripting
  - **git clone https://github.com/sandervanvugt/bash-scripting**
- Course command history is available at https://pastebin.com/u/sandervanvugt

livelessons
video instruction from technology experts

1: Getting Familiar with Bash

Using Variables

Understanding Variables

- To have programs (including shell scripts) work with site-specific data, variables are used
- Using variables allows an operating system to keep program code generic, and separated from site-specific information
- The Linux operating system itself comes with variables
- Use **env** to print a list of these environment variables
- Best practice: to write efficient shell scripts, use variables a lot

Using Variables

- To define a variable, use **key=value**
- By default, variables will be available in the current shell only
- To define a variable fore the current shell and all subshells, use **export key=value**
- To make sure a variable is automatically set, put it in one of the Bash startup files
- Refer to a variable using **echo $key**
- To avoid ambiguity, use **echo ${key}**; compare **echo ${key}1** with **echo $key1**

livelessons
video instruction from technology experts

1: Getting Familiar with Bash

Understanding exit Codes

Understanding exit codes

- After execution, a command generates an exit code
- The last exit code generated can be requested using **echo $?**
- If 0, the command was executed successfully
- If 1, there was a generic error
- The developer of a program can decide to code other exit codes as well
- In shell scripts, this is done by using **exit *n*** in case an error condition occurs

livelessons
video instruction from technology experts

1: Getting Familiar with Bash

Alternative Shells

Understanding Linux Shells

- Bash is Bourne Again Shell, a remake of the original Bourne shell that was invented in the early 1970's
- Bash is the default shell on most Linux distributions
- Another common shell is Zsh, which is used as the default shell on MacOS and recent Ubuntu
- And yet another common shell is Dash, which is used frequently in Debian environments
- While writing shell scripts, Bash is the standard and it's very easy to make Bash work, even if you're in a non-bash shell

livelessons
video instruction from technology experts

1: Getting Familiar with Bash

What is a Shell Script

What is a Shell Script?

- A shell script is a computer program designed to run in a shell
- Scripts can be written in different scripting languages
- Typical functions are file manipulation, program executing and printing text

What's the use of Shell Scripts

- Shell scripts are strong in manipulating data
- You can use them for instance to filter ranges, change file names, change data on a large scale easily
- Shell scripts are a perfect solution to do so, as shell scripts are easy to develop, and run from the leading Linux operating system
- For that reason, shell scripts are commonly used in data science and other professional environments

Understanding Bash Script Compatibility

- Every shell comes with a specific feature set
- Bash features might not exist in other shells
- To ensure a Bash script is interpreted the right way, it should start with **#!/bin/bash**
- The only requirement for this to work, is that the Bash shell must be installed on the current operating system

livelessons
video instruction from technology experts

2: Creating your First Shell Script

Choosing an Editor

Choosing an Editor

- To write shell scripts, you need an editor
- This editor must offer syntax highlighting
- Apart from syntax highlighting there are no specific requirements
- Any common editor offers syntax highlighting for shell scripting
- Common Linux editors are **vim**, **nano** and **gedit**
- There are no IDEs for Bash Scripts

livelessons
video instruction from technology experts

2: Creating your First Shell Script

Core Bash Script Ingredients

Core Bash Scripting Ingredients

- Best practice: make sure your scripts always include the following
- Shebang: the indicator of the shell used to run the script code on the first line of the script:
  - **#!/bin/bash** or **#!/usr/bin/env bash** to identify Bash as the script interpreter
- Comment to explain what the script is doing
- White lines to increase readability
- Different block of code to easily distinguish between parts of the script

About the Script Name

- Script names are arbitrary
- Extensions are not required but may be convenient for scripts users from non-Linux operating systems

About the Script Location

- Because of Linux $PATH restrictions, scripts cannot be executed from the current directory
- Consider using **~/bin** to store scripts for personal use
- Consider using **/usr/local/bin** for scripts that should be available for all users

livelessons
video instruction from technology experts

2: Creating your First Shell Script

Running the Scripts

Running Scripts

- To run a script as a separate program, you need to set the execute permission: **chmod +x myscript**
- If the directory containing the script is not in the $PATH, run it using **./myscript**
- Scripts can also be started as an argument to the shell, in which case no execute permission and $PATH is needed: **bash myscript**
- Best practice: to avoid confusion, include a shebang on the first line of the script and put it in the $PATH so that it can run as an individual program

DEMO

- hello-world

livelessons
video instruction from technology experts

2: Creating your First Shell Script

Using Bash Internals versus External Commands

Using Internal versus External Commands

- Internal commands are a part of the Bash shell and for that reason are faster
- External commands can be any executable that exists on disk
- Using external commands may fail on other systems because of dependencies: it might not be installed
- Using internal commands is faster
- But external commands may provide functionality that is not offered by internal commands
- Best practice: if there's a choice, go for the internal command unless you have a good reason not to

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

About Terminology

About Terminology

- An *argument* is anything that can be put behind the name of a command or script
  - **ls -l /etc** has 2 arguments
- An *option* is an argument that changes the behavior of the command or script, and its functionality is programmed into the command
  - In **ls -l /etc**, **-l** is used as an option
- A *positional parameter* is another word for an argument
- A *variable* is a key with a name that can refer to a specific value
- While being handled, all are further threated as *variables*

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Quoting

Understanding Quoting Options

- Escaping is the solution to take away special meaning from characters
- To apply escaping, use quotes
- Double quotes are used to avoid interpretation of spaces
  - **echo "my value"**
- Single quotes are used to avoid interpretation of anything
  - **echo the current '$SHELL' is $SHELL**
- Backslash is used to avoid interpretation of the next character
  - **echo the current \$SHELL is $SHELL**

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Defining and Using Variables

Understanding Variable

- A local variables works in the current shell only
- An environment variable is an operating system setting that is set while booting
- Arrays are special multi-valued variables covered later

Bash Variable Data Types

- Bash variables don't use data types
- Variables can contain a number, character or a string of characters
- **declare** can be used to set specific variable attributes
  - **declare -r ANSWER=yes** sets $ANSWER as a readonly variable
  - **declare [-a|-A] MYARRAY** is used to define an indexed or associative array (treated later)
- Using **declare** is NOT required

Defining Variables

- Defining a variable can be easy: **key=value**
- Variables are not case sensitive
  - Environment variables by default are written in uppercase
  - Local variables can be written in any case
- After defining, a variable is available in the current shell only
- To make variables available in subshell also, use **export key=value**
- To clear variable contents, use **key=**
- Use **env** to get access to environment variables

Using Variables

- To use the current value assigned to a variable, put a $ in front of the variable name
- To better deal with variables that have a space in their value, it is recommended though not mandatory to put the variable name between { }
  - **color=red**
  - **echo $color**
  - **echo ${color}**

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Defining Variables with **read**

Understanding **read**

- When **read** is used, shell script execution will stop to read user input
- The user input is stored in the variable that is provided as an argument to **read**

```
echo enter a value
read value
echo you have entered $value
```

- If **read** is used without further argument, the user input will be ignored
- This can be useful for "Press Enter to continue" structures

```
echo press enter to continue
read
echo continuing...
```

Demo

- readit

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Understanding Variables and Subshells

Understanding Variables and subshells

- If not used with **export**, variables are available in the current shell only
- As separation of data types is good practice in programming, it is common to define variables in separate files and use **include** to import them in the current shell
- This is done using either of two methods:
  - **include myvars**
  - **. myvars**
- In many Linux distributions, include files for services are stored in /etc/sysconfig

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Handing Script Arguments

Understanding Script Arguments

- Scripts can be started with arguments to provide specific values while executing the script
- *Arguments* are also referred to as *Positional Parameters*
- The first argument is stored in $1, the second argument is stored in $2 and so on
- By default, a maximum of nine arguments can be defined this way
- When using curly braces, more than nine arguments can be provided

Referring to Script Arguments

- Script arguments can be addressed individually
- To address all arguments, use **$@** or **$***
- Without quotes, **$@** and **$*** are identical
- With quotes, **$@** expands to properly quoted arguments, and **$*** makes all arguments into a single argument

Demo

- script3
- script4

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Using Command Substitution

Understanding Command Substitution

- Command substitution is used to work with the result of a command, instead of a static value that is provided
- Use this to refer to values that change frequently
  - **today=$(date +%d-$m-%y)**
  - **mykernel=$(uname -r)**
- Command substitution can be done in two ways that are not fundamentally different:
  - **today=$(date +%d-%m-%y)**
  - **today=`date +%d-%m-%y`**
- best practice: for readability purposes, use **$(...)**

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Using here Documents

Understanding Here Documents

- A *here document* is used as I/O redirection to feed a command list to an interactive program or command
- A here document is used as a scripted alternative that can be provided by input redirection
- Here documents are useful, as all the code that needs to be processed is a part of the script, and there is no need to process any external commands

Here Document Example

```
#!/bin/bash

lftp localhost <<ENDSESSION
ls
put /etc/hosts
ls
quit
ENDSESSION
```

- demo: setupftp

```
#!/bin/bash

# due to the location in the course where this script is used it lacks the proper conditional checks
sudo yum install -y vsftpd lftp
sudo sed -i -e 's/anonymous_enable=NO/anonymous_enable=YES/' /etc/vsftpd/vsftpd.conf

# again this is not pretty but I don't want to show anything that hasn't been covered yet
sudo systemctl disable --now vsftpd
sudo systemctl enable --now vsftpd
sudo cp /etc/hosts /var/ftp/pub/

# fetching a file wit a here document
lftp localhost <<ENDSESSION
ls
cd pub
get hosts
ls
quit
ENDSESSION
```

livelessons
video instruction from technology experts

3: Working with Variables and Arguments

Using Functions

Understanding Functions

- A function is a small block of reusable code that can be called from the script by referring to its name
- Using functions is convenient when blocks of code are needed repeatedly
- Functions can be defined in two ways:
  - ```
    function_name () {
        commands
    }
    ```
  - ```
    function function_name {
        commands
    }
    ```

Using Function Arguments

- Functions can work with arguments, which have a local scope within the function

```
#!/bin/bash
hello () {
  echo hello $1
}
hello bob
```

- Function arguments are not affected by passing positional parameters to a script while executing it

livelessons
video instruction from technology experts

4: Transforming Input

Working with Parameter Substitution

Understanding Parameter Substitution

- Parameter substitution can be used to deal with missing parameters
- Use it to set a default, or to display a message in case missing parameters were found

NTS: demo

- psubst1

- If parameter is not set, use default (does NOT set it)
  - **echo ${username:-$(whoami)}** uses result of whoami is $username is not set
  - **filename=${1:-$DEFAULT_FILENAME}** uses value of $DEFAULT_FILENAME if $filename is not set
- If parameter is not set, set to default
  - **echo ${username:=$(whoami)}** uses result of whoami is $username is not set
  - **filename=${1:=$DEFAULT_FILENAME}** uses value of $DEFAULT_FILENAME if $filename is not set
- If parameter is not set, print error_msg and exit script with exit status 1
  - **echo ${myvar:?error_msg}**

NTS: demo

- psubst2
- psubst3
- psubst4

livelessons
video instruction from technology experts

4: Transforming Input

Using Pattern Matching Operators

Understanding Pattern matching Operators

- The purpose of a pattern matching operator is to clean up a string
- **${1#}** prints the string length of $1
- **${1#pattern}** removes the shortest match of pattern from the front end of $1
- **${1##pattern}** removes the longest match of pattern from the front end of $1
- **${1%pattern}** removes the shortest match of pattern from the back end of $1
- **${1%%pattern}** removes the longest match of pattern from the back end of $1

Understanding Variable string replacement

- **${var/pattern/replacement}** is used to replace a pattern inside a variable
- **${var//pattern/replacement}** performs a global replacement
- **${var/#pattern/replacement}** will only replace if the variable start with pattern
- **${var/%pattern/replacement}** will only replace if the variable ends with pattern

NTS

- script6
- script7
- pm

livelessons
video instruction from technology experts

4: Transforming Input

Patterns and Extended Globbing

Patterns and Extended Globbing

- Extended globbing can be used to analyze file patterns in a smart way
- See **patternglob** in the course Git repository
- See **removepattern** in the course Git repository

livelessons
video instruction from technology experts

4: Transforming Input

Calculating

Understanding Bash Calculations

- Bash offers different solutions for calculation with integers:
  - **let expression**
  - **expr expression**
  - **$((expression))**
- Of these 3, the $((...)) method is preferred
- All work with the following operators
  - +
  - -
  - *
  - /
  - %

Bash Calculation Examples

- **let** is a bash internal

```
let a=1+2
echo $a
let a++
echo $a
```

- **expr** is an external command that can be used in scripts, not used much anymore
- **$((...))** allows you to put the calculation between parenthesis

```
echo $(( 2 * 3 ))
```

Advanced Calculation Tools

- **bc** is an advanced calculation tool that allows you to work with decimals
  - **bc** is typically used in pipes: **echo "12/5" | bc**
  - to print decimals in the result, use **-l: echo "12/5" | bc -l**
  - **bc** also offers access to built-in mathematical functions: **echo "sqrt(1000)" | bc -l**
- **factor** decomposes an integer into prime factors
  - **factor 399**

Demo

- countdown

livelessons
video instruction from technology experts

5: Using **if** and **if ... then ... else**

Using **test**

Using **test**

- **test** is the foundation of many **if** statements
- **test** is an external command that allows you to perform different types of test
  - Expression tests: test can evaluate the binary outcome of an expression, which is a logical test by itself
  - String tests: allow you to evaluate if a string is present or absent, and compare one string to another
  - Integer tests: allow you to compare integers, which includes operations like bigger than, smaller than
  - File tests: allow you to test all kind of properties of files
- **test** is typically used in conditional statements
- **test *condition*** can also be written as **[*condition*]**

livelessons
video instruction from technology experts

5: Using **if** and **if ... then ... else**

Using Simple **if** Statements

Using Simple **if** Statements

- **if** is used to verify that a condition is true

```
if true
then
        echo command executed successfully
fi
```

- The condition is a command that returned an exit code 0, or a test that completed successfully:

```
if [ -f /etc/hosts ]; then echo file exists; fi
```

livelessons
video instruction from technology experts

5: Using **if** and **if ... then ... else**

Using Logical Tests

Using || and &&

- Logical test are if then else tests written in a different way
- Logical AND: the structure is **a && b**, which results in **b** being executed when **a** is successful
- Logical OR: the structure is **a || b**, which results in **b** being executed only if **a** is not successful
- Logical operators can be embedded in **if** statements to test multiple conditions
  - **if [-d $1] && [-x $1]; then echo $1 is a directory and has execute; fi**

Rewriting if statements to logical test

- **if [-f /etc/hosts]**
  then
  echo file exists
  fi
- **[-f /etc/hosts] && file exists**

livelessons
video instruction from technology experts

5: Using **if** and **if ... then ... else**

Testing with **[[…]]**

Understanding [[*condition*]]

- A regular test is written as **[*condition*]**
- The enhanced version is written as **[[*condition*]]**
- **[[*condition*]]** is a Bash internal, and offers features not offered by **test**
- Because it is a Bash internal, you may not find it in other shells
- Because of the lack of compatibility, many scripters prefer using **test** instead

[[*condition*]] examples

- **[[$VAR1 = yes && $VAR2 = red]]** is using a conditional statement within the test
- **[[1 < 2]]** tests if 1 is smaller than 2
- **[[-e $b]]** will test is $b exists. If $b is a file that contains spaces, using [[ ]] won't require you to use quotes
- **[[$var = img* && ($var = *.png || $var = *.jpg]] && echo $var starts with img and ends with .jpg or .png**

livelessons
video instruction from technology experts

5: Using **if** and **if ... then ... else**

Using **if ... then ... else**

Extending **if** statements with **else**

- **else** can be used as an extension to **if** statements to perform an action if the first condition is not true
- Notice that instead of using **else**, independent statements can be used in some cases
- See the differences between **else1** and **else2** in the course Git repo

livelessons
video instruction from technology experts

5: Using **if** and **if ... then ... else**

Using **if ... then ... else** with **elif**

Understanding **elif**

- **elif** can be added to the if ... then ... else statement to add a second condition

```
if [ -d $1 ]; then
        echo $1 is a directory
elif [ -f $1 ]; then
        echo $1 is a file
else
        echo $1 is an unknown entity
fi
```

demo

- numcheck

livelessons
video instruction from technology experts

6: Using Conditionals and Loops

Applying Conditionals and Loops

Understanding Conditionals and Loops

- **if ... then ... else** is used to execute commands if a specific condition is true
  - **if [-z $1]; then echo hello; fi**
- **for** is used to execute a command on a range of items
  - **for i in "$@"; do echo $i; done**
- **while** is used to run a command as long as a condition is true
  - **while true; do true; done**
- **until** is used to run a command as long as a condition is not true
  - **until who | grep $1; do echo $1 is not logged in; done**
- **case** is used to run a command if a specific situation is true

livelessons
video instruction from technology experts

6: Using Conditionals and Loops

Using **for**

Using **for**

- **for** is used to iterate over a range of items
- This is useful for handling a range of arguments, or a series of files

- for

livelessons
video instruction from technology experts

6: Using Conditionals and Loops

Using **case**

Using **case**

- **case** is used to check a specific number of cases
- It is useful to provide scripts that work with certain specific arguments
- It has been used a lot in legacy Linux init scripts
- Notice that **case** is case sensitive, consider using **tr** to convert all to a specific case

demo

- yum install -y bash-scripting
- case

livelessons
video instruction from technology experts

6: Using Conditionals and Loops

Using **while** and **until**

Using **while** and **until**

- **while** and **until** are used to run a command based on the exit status of an expression
- **while** will run the command as long as the expression is true
- **until** will run the command as long as the expression is not true

- script10

livelessons
video instruction from technology experts

6: Using Conditionals and Loops

Using **break** and **continue**

Understanding **break** and **continue**

- **break** is used to leave a loop straight away
- Using **break** is useful if an exceptional situation arises
- **continue** is used to stop running through this iteration and begin the next iteration
- Using **continue** is useful if a situation was encountered that makes it impossible to proceed

- backout
- convert