Introduction to Shell Scripts

Prof. Dr. Peter Braun

15. Oktober 2024

1 Titlepage

Introduction to Shell Scripts

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

2 Goals of this Unit

Goals of this Unit

- You learn the concept of shell scripts
- You learn how to make a script executable
- You learn how to pass arguments to a script
- You learn how to use arguments in the script

Welcome to this unit, in which we would like to give you a brief introduction to programming shell scripts. We assume you are already familiar with a Linux system, know how to use the command line, and learned some basic Linux commands. With shell scripts, you can now write your commands without using programming languages such as C or other higher programming languages. In this unit, we will first introduce you to the basic concepts of shell scripts. You will learn some first examples, and we will show you how to execute a shell script from the command line. We will then show you how to pass arguments or options to a shell script and how you can evaluate these arguments. This is the first of three units that introduce the programming of gel scripts. The other two units will deal with control structures and variables.

3 Use Cases of Shell Scripts

Use Cases of Shell Scripts

- **Automatise working with the shell**
- Every sequence of commands should be a script
- System administration, e.g. install and configure

As already mentioned, we use shell scripts to program new Linux commands, which we can then use on the command line like the existing commands. And we do this whenever we want to automate our work with the command line. In practice, executing several commands in succession is often necessary, even for simple tasks. And this happens again and again. As a computer scientist, you should quickly realize that you should automate this mindless typing of commands. And that's exactly what you write a shell script for. Many shell scripts are very simple and summarize a handful of other Linux commands in a certain way. However, very complicated shell scripts use common concepts from programming languages, such as loops and branches. Of course, you could also work directly with higher programming languages instead of shell scripts. Python is, for example, a good language for this. However, these programming languages are often not available in a minimal setup. It would be possible, but it is not recommended.

4 Introduction Bash

Introduction to Bash

- By scripts we create new complex commands
- Bash is a Turing-complete programming language
- Control structures like loops and conditions
- Variables to save data temporarily
- Communication with the user
- You **should not** use Bash for complex problems

If you have already familiarised yourself with the operation of a Linux computer, then you probably already know the term BASH. On the one hand, this term stands for the name of a so-called shell, i.e., a program that receives and executes commands from the user. On the other hand, Bash is also the name of a programming language we want to use to develop shell scripts. This programming language is Turing-complete, which means that any problem that can be solved with a computer can also be solved with a Bash program. Once you have gained more experience in shell programming, you can test this yourself by developing a simulator for a Turing machine. It is typical for such Turing-complete programming languages that they offer control structures such as loops and branching. There is also a concept of variables in Bash, and there is, of course, the possibility to communicate with the end user, i.e., to receive input and produce screen output. Of course, reading and creating new files is also possible. Even if solving all computable problems with a shell program is feasible, writing complex programs in the Bash programming language is not necessarily recommended. Bash scripts are typically rather slow, much slower than programs written directly in C. The programming language is also not necessarily easy to read, so the programs should remain small.

5 Finding a Name for the Script

Finding a Name for the Script

- Select a meaningful and unique name
- Check for programs with same name by `which`
- `which` searches for programs with given name
- Example: `which ls`

So, let's start right away with our first script. Firstly, we need to define a name for the script. This name should be meaningful, and, above all, it should be unique. To check whether there is already another program or script on the computer with the same name, you can use the program "which." This program searches the entire computer for other programs with the specified name. If a program already exists under this name, the corresponding path to the program appears on the screen. If no program was found, no output will appear. As I said, you should always choose a name for your script that still needs to be created on the system.

6 First Script: textttuserinfo

First Script: `userinfo`

- We select `userinfo` as name for our first script
- Start vi: `vi userinfo.sh`
- Input the following text:

```
echo "I am ..."
whoami
echo "Today is ..."
date
```

- Make script executable: `chmod u+x userinfo.sh`
- Execute script in the terminal: `./userinfo.sh`

We have now decided on "userinfo" for our first script. With this script, we want to output the currently logged-in user's name and today's date. There is no deeper meaning behind this, but it simply serves as a demonstration for a very simple shell script. We start the editor like VI and enter the name of our shell script as an argument. Please note that we append the suffix "SH" to "userinfo". "SH" stands for Shell Script. This suffix is not necessary, but it is a good style. Now type in the text you see next to me and exit the editor with the command: Q W. When you are back on the command line, you have to make the new shell script executable with the Linux command Change mod and the arguments as shown next to me. The first argument, "U plus X," means that the owner of this file, i.e., you, should be authorized to execute it. You can run your script in the command line next to me as specified. You may wonder why you must enter the dot and the slash before the name. When the user enters a command in the command line, the shell only searches for this command in certain directories. The list of directories searched by the shell is stored in the PATH environment variable. Usually, the current directory is not included in this PATH variable. Without the dot and the slash, the shell could not find our script. We, therefore, specify the hard path here. And now, let's take a look at what happens live.

7 Screencast 1

Screencast 1

- Zeige die letzte Folie live am Rechner

Let's have a look at the first script. So we start the editor. And now we copy the script from the slide. We start with the echo command, I am, then the who am I command. And the next line, today is. And finally the date command. Then we save this. And next we have to change the permissions of this file. So at the moment we are not allowed to execute the file. You can see there is no X in the set of permissions. And to change the permissions we use the command change mode or change mod. And we set the permission for us as user and owner of the file. And we add the X permission to execute this file. We double check and now you can see in the first group of three permissions RWX there is now the X set. So we can execute the file. And that's the output. That's how we defined this in the script. And next we want to improve the line feed after I am and after today is there should be no line feed. And we have to find out how can we modify the echo command. So we open the main page of echo. And I think we had this in the last unit already or in one of the last units. So it's the first option dash N by which we can skip the new line character. So we start VI again. And we add now this dash N option before we define the text as argument here. And there is a dot missing at the end of the string. So we save this and we start the script again and now it looks like it should look like.

8 She-Bang Line

She-Bang Line

- First line defines the program used for execution
- The line begins with `#!` followed by the program
- `#!/bin/bash` defines to use Bash
- `#!/usr/bin/env bash` is used to ensure portability
- `#!/usr/bin/python` defines to use Python

Congratulations, you have now written your first small shell script. Let's briefly ask ourselves what has just happened here. You have called the script in the shell, but how did it know that it is a shell script and not a binary program that you have previously written in the C programming language or a Python script? In general, binary programs and scripts differ in the first bytes of the corresponding file. The shell can at least easily distinguish between these two categories. And the shell now assumes that if the command is not a binary program, it must be a shell script. The shell cannot automatically determine whether the text file contains Python code, so it calls the Python interpreter accordingly. To give the shell some help here, there is the convention of the so-called she-bang line. This line begins with a hash symbol followed by an exclamation mark and then the full path to the program to run. For shell scripts, it is strongly recommended that you fill this first line as in the first example next to me. The path to the program for executing such shell scripts is /bin/bash. If you plan to run your script on any Linux or even Unix variants, the second version of this she-bang line is recommended. The program "ENV" for Environment starts the program specified as an argument. This program can be located anywhere in the directory tree. If you do not want to write a shell script but a Python program, you can use the third option in the She-Bang line to tell the shell the path to your Python interpreter. Doing this allows you to start a Python program without manually calling the interpreter.

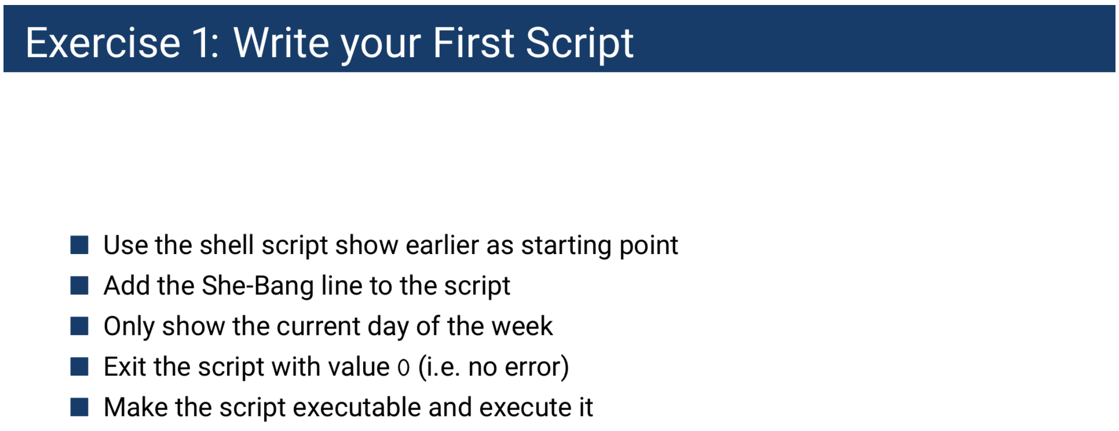
9 Good Practise for Shell Scripts

Good Practise for Shell Scripts

- Meaningful and unique name
- She-Bang line defines program used for execution
- Write comments in your shell script
- Terminate the script with an appropriate `exit` code

Before you write your first shell script in the first exercise, here are a few general tips on programming shell scripts. As already mentioned, you should always take care to find a unique and meaningful name. As shown, you should reserve the first line in your shell script for the so-called She-bang line. This is not necessary, but it is good style. As soon as the shell scripts become even more complex, start writing comments in your script. Comments always start with the hash sign. If you have already gained some experience in software development with programming languages such as Java or C++, you may have already come into contact with the principle of clean code. The idea behind this is that you write source code that is easy to read and requires few or no comments. Shell scripts are different. Shell scripts cannot be programmed to be readable. They are cryptic and that's how it is. You typically use other Linux commands in shell scripts, and these, in turn, expect arguments or options that are sometimes extremely cryptic for historical reasons. You should explain such parts with a comment so that another software developer understands what you mean here. Last, it would be best to get into the habit of always ending a shell script with an EXIT code. You may already know about EXIT codes from the normal Linux commands. An EXIT code of zero means that everything was OK. An EXIT code greater than zero indicates an error. The command for setting the Exit code is simply EXIT with a number as an argument.

10 Exercise 1: Write your First Script



Exercise 1: Write your First Script

- Use the shell script show earlier as starting point
- Add the She-Bang line to the script
- Only show the current day of the week
- Exit the script with value `0` (i.e. no error)
- Make the script executable and execute it

Now, let's move on to the first exercise. Use the script you just typed from the slide and work on the following three points. Firstly, add the she-bang line. Then, change the call to the Linux date command so that only the current day of the week is output and no longer the whole date. Please refer to the manual to find the appropriate option for this. Also, add an EXIT command with argument 0 at the end of the script. Then, make the script executable if you still need to do so. Execute the script to see if it works. Please click the button when you are finished or want to view the solution.

11 Screencast 2

Screencast 2

- todo

Let's go through the modifications step by step. So we open the script and the first is to add the shebang line here. So hash exclamation mark slash bin slash bash. Next we want to add that we print the weekday and not the full date Because we don't have the syntax for this we have a look at the main page here. So with plus format we can change the output and we only need to know what is the correct value for format here. And from the description you can see it's the percentage and capital A to print the weekday. So we have to modify the script. We open the editor again and we go to the last line and we add now the plus percentage sign and capital A. And we can add the exit zero with the exit state again. So we start the script and you can see the output is what we wanted to have and we check the exit code with a variable and that's correct.

12 Variables in shell scripts

Variables in Shell Scripts

- Bash has a concept of variables but not data types
- Define a variable: `VARIABLE=5`
- Access the value of a variable: `$VARIABLE`
- Sometimes necessary: `${VARIABLE}`

Shell scripts that call other Linux commands in sequence in the same way again and again can be extremely helpful but are not powerful. Therefore, we will examine two ways of processing user input in the next block. This is needed to make the scripts more flexible. With the first type of user input, we wait for input from the user while the script is running. In the second variant, we allow users to enter arguments when the script is called. In both cases, we must save the user input so we can use it again later. For this, we need to know the concept of variables. Let's take a look at that first. The Bash programming language has a concept of variables but no idea about data types. Unlike in other programming languages, you don't have to declare variables before you can use them. They write an assignment operation anywhere in their script, with the variable's name on the left and the value after the equal sign. You can see an example next to me. The name of the variable happens to correspond to the word variable. In our examples, variables are often written with capital letters, but this does not have to be the case. If you need to access the value of a variable, write the dollar sign before the variable's name. It is sometimes necessary and, in general, recommended that you put a curly brace after the $ and a curly brace at the end of the variable name when accessing the value of a variable. You will see why this is important later in the examples.

13 Reading User Input

Reading User Input

- `read` waits for input until `RETURN` is pressed
- Content is stored in a variable passed as parameter

```
#!  /usr/bin/env bash
read input
echo "The user typed:  ${input}"
```

Now, we come to the command by which we can expect input from the user. The command is “read,” it waits for user input until the Return key is pressed. When this command is called, the name of a variable in which the value, i.e., the user input, is stored must be specified as an argument. We can see an example of this next to me. The argument for the “Read” command, i.e., the variable name, is specified here without dollar sign. In the next line, we use the “Echo” command to first output the text "the User typed,"followed by the content of the variable input. Pay attention to the positions of the double quotes here.

14 Configure Reading User Input

Configure Reading User Input

- `read -s` does not print the input
- `read -t SEC` waits the given amount of time
- `read -n N` only read given number of characters

The Read command can handle three options. These options can be used to influence the behavior of the command. The first option is “-S,” meaning the user’s input does not appear on the screen. This can be useful, for example, if you want the user to enter a password. The second option, “-T,” then expects a further argument: a number. This option specifies that the “Read” command waits a maximum of seconds before it is automatically canceled. In this variant, the Read command does not wait until the return in the input, but may terminate beforehand after the specified time. Of course, if the user enters the return character beforehand, the input ends. Finally, you can select the maximum number of characters to be expected by using the “-N” option and entering a number. The Read command is also canceled when the return key is entered first. Otherwise, it would wait for a maximum number of characters before stopping.

15 Arguments in Shell Scripts

Arguments in Shell Scripts

- Every shell script defines few variables automatically
- The file name of the script is written into `$0`
- `$1` till `$9` contain arguments of the executed script

The second option for processing user input uses arguments in shell scripts. You are already familiar with the principle of arguments from Linux commands. If you enter the command's name in the command line, you can pass further options or arguments. It is the case that everything you enter after the actual command is processed as an argument. This space character is understood as a separator. For example, if you enter a space after the name of the command, then an A, then a space, and then a B, you have passed two arguments to your program. In Bash programming, some variables are set automatically when the script is started. These variables begin with the usual dollar, followed by another character. The variable Dollar Zero contains the name of the program or the name of the script. The first argument is then stored in the variable dollar one, the second in the variable dollar two, and so on. In this way, you can process a maximum of nine arguments. But don't worry; other methods exist to process the arguments, and you can expect more than nine arguments.

16 Exercise 2: Using Arguments

Exercise 2: Using Arguments

Write a shell script that accepts two arguments and prints

- the name of the script
- the value of the two arguments

Why don't you try it out? Either use your script from the first exercise or write a new script. The script should display three things on the screen: firstly, the script's name. As just explained, the name is in a special variable. Then, the values of the two arguments should be displayed on the screen. Save it, make it executable, and call it up. Why don't you try to find out what happens if you enter only one or three arguments instead of two? If you want to see the solution, please click on the button.

17 Screencast 3

Screencast 3

- todo

So we start with a new script for this exercise. So we open the editor and the name of the script should be printargs.sh. We start with a shibang line. Next we output the name of the script, so echo, then name as label and $0 is the variable name for the script name. Next we want to show the arguments, but I think it's better not to use dash n here, because we want to have the output in subsequent lines. So next we print the first argument, argument 1, and the variable for this is $1. And next argument 2 with $2 as the variable name. So we save this, oh yeah, we exit no, exit 0, then we save this and we change permissions for this script. And we call the script in the first run without arguments, but we can see the name of the script. And now we start the script with two arguments, hello and world, and then we see the output.

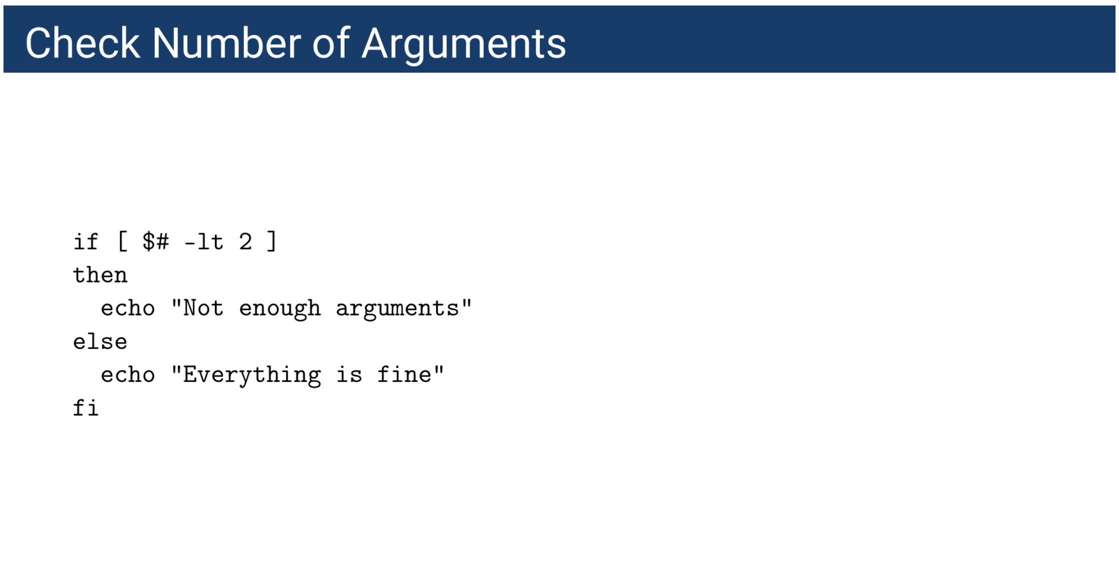
18 Arguments in Shell Scripts

Arguments in Shell Scripts

- `$#` contains the number of arguments
- `$*` contains all arguments as one string
- `$@` contains all arguments as string array
- We talk about arrays in another unit

In addition to the special variables with the names of the program and the arguments, there are three other variables in connection with the processing of arguments that you need to know. You will first see the Dollar Hash variable in the list next to me. This variable contains the number of arguments. The second variable has the name Dollar Star and includes all the arguments the user specified when calling their command a long string. Sometimes, it makes sense not to process the arguments individually but to have them available in one string. But this depends very much on the use case, of course. The last variable is called "Dollar-At," this variable contains all the arguments that the user specified on the command line but is separated by and stored as an array. You can then access these arguments with a position specification, as is usual with arrays. For example, this variable is often used if you want to process all arguments in sequence with a loop.

19 Check Number of Arguments

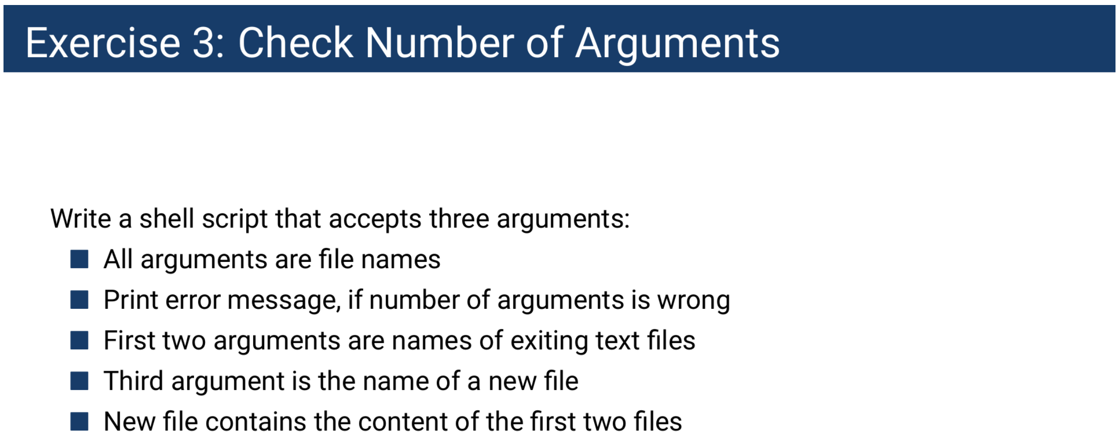


Check Number of Arguments

```
if [ $# -lt 2 ]
then
  echo "Not enough arguments"
else
  echo "Everything is fine"
fi
```

Before we come to the last exercise, I would like to preview the control structures. When processing arguments, you often need to query whether the number of arguments passed matches the expectation. To do this, you need the IF statement. And you can see an example of this next to me. If you already have some knowledge of programming, you should be able to recognize the basic structure immediately. The condition follows the IF keyword. Inside the square brackets is the expression we require that the number of arguments, i.e., the content of the variable dollar hash, is compared with the number 2. The operator "-LT" stands for less than. So if the number of arguments is less than two, the ECHO statement is executed in the THEN block. Otherwise, the ECHO statement is executed in the Else branch. We, therefore, use this instruction to check whether we have received at least two arguments.

20 Exercise 3: Check Number of Arguments



Exercise 3: Check Number of Arguments

Write a shell script that accepts three arguments:

- All arguments are file names
- Print error message, if number of arguments is wrong
- First two arguments are names of exiting text files
- Third argument is the name of a new file
- New file contains the content of the first two files

Please use this in today's last exercise. Now, please write a new shell script. First, choose a good name; the shell script should expect three arguments. There are exactly three arguments. In all other cases, you can output an error message. The first two arguments are the names of existing files, and the third argument should be the new file's name. The task of the script is to save the contents of the first two files, which already exist, one after the other, in the new third file. You may have to do some research to find instructions for this. When you are finished or want to view the solution, please click the button.

21 Screencast 4

Screencast 4

- todo

So for this exercise we start a new script. We start the editor, the UI and the name of the script should be concat because it's about concatenating of files here. First we add the Shibang line and next we have to check the number of arguments. So we write an if statement here. And we check that the number of variables or sorry the number of arguments here which is stored in the variable dollar and hash. If it's not equal and E stands for not equal, if it's not equal to three then we output an error message here that the number of arguments is wrong and you have to provide three arguments. And in the other case else, so if the number of arguments is correct then we use the cat command and you know the cat command already to print the contents of one file. But with the cat command you can also give any number of arguments or any number of file names here and then all the files are printed one after the other. So we can just give here the variables dollar one, dollar two for the two first arguments of this script here and everything is then printed to the console. But we don't want to have this on the screen. We want to redirect this into a file and the file name is given as the third argument. So dollar three and we finish the if statement and we add finally the exit code with zero here. And we can add another exit command here because if the number of arguments was wrong we could exit with the code one which indicates that there is a problem. So we have to change the permissions as usual and then we start the script. And first without any arguments we get the error message and the exit code is one. And next we start the script with two file names of text files. You can see here we have some text files prepared so we start concat with text one and text two and the output should be in a file named result dot txt. And first we check the exit code which is now zero so everything is okay. We check that the file does exist. The file is there. Then we check that we check the contents of the two text files text one and text two and finally the contents of the result.

22 Summary

Summary

- You learned the foundation of shell scripts
- You learned how execute shell scripts
- You learned how to pass arguments to a script
- You learned how to use arguments in the script

That's all for today. In this first of three units on shell programming, you have learned the basics. You now know that you can use shell scripts to develop new commands composed of simple Linux commands. We have shown how to process user input, and you have seen how to process arguments given by the user on the command line when calling their script. In between, they also saw a first example of how to work with variables and what the IF statement looks like. In the next two units, we will deepen this knowledge, and you will learn how to develop more complex scripts.

23 Literature

Literature

- Patrick Ditchen: Bash: Einstieg in die Shell-Programmierung. mitp, 2018.
- Jürgen Wolf: Shell-Programmierung. 5. Auflage, Rheinwerk, 2016.
- Sander van Vugt: Beginning the Linux Command Line, Apress, 2015.