Shell Scripting with Bash

by Reindert-Jan Ekker

Start Course

Bookmark Add to Channel Download Course

Introduction

Introduction

[Autogenerated] Hello. My name is financial backer and welcome to this course called Shell Scripting With Boesch. In this first module, I'll give you an overview of the course. First of all, I'll address the question why you would actually need shell scripting. I'll also tell you what you can expect from watching this course, and I'll tell you what you need to be able to follow the course.

Why Shell Scripting?

So, why would you need or want to use shell scripts? Well, basically the shell scripts let you execute Bash commands from a file. So, when there's a set of commands that you find yourself executing regularly, you can put them in shell script and run them over and over or even run them when you're away or, for example, schedule them every day at a certain time. Shell scripts will also let you take advantage of the whole UNIX toolset with all the power that comes with it. Now, who would typically need to do this? Well, the first thing that comes to mind is a system administrator. Bash shell scripts are a must have for any serious UNIX administrator, and they're also an important power tool for developers on a UNIX system who might for example want to write an installation or a build script. And, of course, there's the general power user for who the

command line and the shell are an important part of his toolset. Now, I would say that there are three main things that Bash scripts excel at. First, there's file manipulation. If you need to move sets of files around or search the file system or make a back up, I really don't think there are many tools that give you the power that Bash does. Secondly, there's the generally running of other programs. Bash is like the glue by which you put other UNIX programs together, and this can be any kind of program from a compiler to an image manipulation tool like ImageMagick to something that lets you download files from the internet. And thirdly, Bash works really well with text, and when you combine with utilities like grep and sed, it makes a very powerful text processer. And, of course, this barely scratches the surface of all the places where Bash scripts can be useful, but I have to admit that sometimes you should look for a better solution. For example, Bash is not very good at doing calculations especially when they need to be done very fast, and it also will not let you handle binary data very well. Also, if you need to display fancy graphics, Bash is probably not the best tool for the job.

What to Expect From This Course?

So, what can you expect from this course? Well, basically I'll show you all of the basics of shell scripting. So, we'll talk about how to do input and output; how to use variables; how to do conditional execution with the if, then, else, and case statements; how to write loops to repeat stuff; how to do arithmetic where Bash unfortunately only does integer arithmetic so you can't do floating points; string manipulation for reading text and changing it; a typical script will also take arguments, and we'll see how to handle those; and finally, Bash scripts allow you to write functions like most programming languages, and we'll see how to write those as well. Now, there are a couple of things that I've chosen not to include in this course. First of all, I'm not going to talk a lot about how to write scripts that are portable to other shells because that would mean that we would have to go into other shells as well, mainly the Bourne shell, and I really think that's beyond the scope of this course. I'm also not going to go very in depth into all the different UNIX utilities there are. I'm kind of assuming that you're already familiar with the tools you need.

Prerequisites

So, what do you need to follow this course? Well, first of all I'm assuming you already have a basic knowledge of the Bash command-line. So, that basically means that you already know how to use all the standard commands like cd and ls and how to remove and copy a file, etc., etc. You should also know about things like redirection and how to use wild cards to match path names. Now, in

case these topics are not familiar to you, I advise you to follow the Bash introduction course first, which will teach you all of these subjects. Now, of course, you also need a Bash installation. Now, for this course it doesn't really matter whether you're running Bash 3, which comes installed on Mac OS or Bash 4, which comes included with most current Linux distributions. By the way, if you're running Bash on any other system like BSD UNIX or maybe Cygwin on Windows, that's okay too because I'm only focusing on Bash and not on the rest of your system. So, now you know what to expect from this course. Let's go ahead and take a first look at shell scripts.

A First Look at Shell Scripts

Introduction

[Autogenerated] Hi, I am around some liquor and welcome to the second module off this course called the First look at Shell Scripts. The aim off this module is to give you an impression off what the Shell script is and what you need to write a shell script. Now we will create two very simple scripts, and while doing this we will learn the following first, of course, what Shell script actually is, and how to create one. How to choose a name for your script that doesn't conflict with other names. How to sit right permissions and, of course, very importantly, how to run your scripts.

Creating A Shell Script

So, let's start with the famous example program called Hello, World. It consists of a single line of code, and I can run it directly from the command line like so. (Typing) And all this does is print the text Hello, World to the terminal. Now, suppose you feel the need execute this code on a regular basis. Bash lets us save this code in a file so we can run it as a program, and such a program is called a shell script. Now, to create my script file, I'm going to use a text editor, and my personal favorite is the Emacs editor, so that's what I'm going to use throughout this course. Now, of course, you can use any editor you like. So, let's go and open Emacs. I'm going to drag it in here, and let's create a file called hw, which is short for Hello, World. Now, the simplest possible way to write our Hello, World scripts would be to just copy, paste the code in here. So, let's do that, and let's Save this, but unfortunately this will not allow us to run the script as a normal command, so let me show you. Here's the script file, and when I try to run it, Bash tells me the command is not found. So, in the current situation, Bash will not run our little program, and we need to take some extra steps to get this working. First of all, our script is not on the PATH that Bash searches for

executables, and we can fix this by specifying the executable we want to run is in the current directory like so. (Typing) Very well. Now, Bash does find the file, but it doesn't allow me to run it, and that's because the script is currently not executable. That means I have to change the permissions on the file so that I can execute it like a program. I'm not going to go deeply into UNIX file permissions here because, in my opinion, that's something more for a general UNIX course, but I can show you how to add executable permission for this file. First, let's list the current permissions. Now, this first part here tells me that I currently have read and write permissions, and the dash here means that I don't have executable permissions for the script. To add executable permission, I'm going to use the command change mod, and the +x part here means that I want to add executable permission. And the U means it's a user permission, so basically I'm adding executable permission only for the user that owns this file, which is me. Now, let's see what just happened. And as you can see, there's an X here now signifying that now I have executable permission. So, let's try to run again, and now it works. Basically what happens here is that Bash tries to execute the file, sees that it's a text file, and starts reading each line in the file, interpreting each line as a Bash command, and trying to execute that. And that's basically how a shell script works. By the way, I can revoke the executable permission by using a minus instead of a plus like so. (Typing) And here we can see that the script is not executable any more. So, let's go over our first steps in the slide. As we saw, a script is nothing more than a basic text file containing code. Usually scripts are run by an interpreter, and because this course is about Bash scripting, our interpreter will be Bash. And what it does is it will read each line in the file, and every line that contains a command will be executed in order from top to bottom. Well, actually that's not entirely true, but we'll get to that soon. Now, to create and edit your scripts, you will need a text editor. Choosing the right editor will make life a lot easier for you, so here are some suggestions. First of all, Emacs and vi are extremely powerful editors with the advantage that they're available for just about every platform. The downside is that they both have quite a steep learning curve. So, if you want something that's a little easier to learn on Linux, you might try Kate or gedit, and on Mac OS there's the excellent free TextWrangler. But of course there are many other editors available, each of which can be used to write shell scripts. Of course, before you can run your new text file, you have to make sure you have executable permissions for that file, and I showed you one way to do this with the change mod commands. In this command, the U character means we are setting a permission for the user who owns the file, the plus means we are adding a permission, and the X notes executable permission. By the way, you can make a script executable for everyone on the system by replacing the U with an A, which stands for all. Now, to remove executable permission, simply replace the plus with a minus sign. Now, in case you want to know want to know more about permissions, use the link I show here. It links to a

page from the Linux documentation projects about permissions. Now, let's go and look at some more useful scripts.

Demo: A Note-Taking Script

Of course the Hello, World script is the simplest possible example of a script, and so it's not really representative of what a real script looks like. So, to give you a more realistic example, let's start on a script that we will expand on in later modules. Suppose I like to be able to take short notes from the command line and add them to a file. I've got a script tn, which is short for take note. Let's start with looking at the first line. It starts with two characters called a shebang or a hashbang. It's these two, a hash followed by an exclamation mark, and this is a special line. This line declares that this script file is meant to be run by the Bash interpreter. I could also use other interpreters here like Bison or Perl or other shells like SH or the Z shell, but in our case we declare Bash to be our interpreter. Now, when a user runs another shell, let's say he runs the Z shell, he can still use our scripts because when he calls a script from his shell the first line will be read, and Bash will be started to run the scripts. So, by adding this first line, we make sure that the script doesn't accidentally get interpreted by another shell. Make sure to always put this at the first line of your scripts. Now, below the shebang are two comment lines. Adding comments to your script is a very good habit to make your scripts understandable and maintainable. And as you can see, everything after a hash sign is a comment, and the interpreter will ignore it. So, let's explain the actual code that is to work that's this single line here, and this is really quite simple script. The $1 here gets the value from a variable called 1, and that's a special variable which denotes the first argument passed to this script on the command line. Echo here will print that value to its output, and then we redirect it to a file called notes. txt. If any of this is unclear to you, I recommend watching the introductory Bash course first so you are familiar with call sets like redirection and variables. Now, let's try. I have already made the script executable, so we don't have to do that. (Typing) And right now there's no note file yet as you can see. I'm going to take a note, (Typing) and as you can see now the file exists and contains my first note. Let's take a second note. (Typing) Very well. This seems to work, and my notes end up in the notes file. But what if I need to save a note longer than a single word? (Typing) Only the first argument of our script ends up in the notes file, and that's not really what I want. One solution would be to quote my whole note like this, (Typing) but I don't want to have to add quotes every single time. Now, there's another solution, and that is to use another special variable in my scripts. Now, this variable holds all arguments passed to the scripts, not just the first one. Later in this course, I'm going to explain more about these special variables, but for now let's just try this, and it should save the whole

note even if I don't use quotes. (Typing) Okay, now suppose I want to add the date and time when the note was taken. I can use the date command, which outputs exactly that information, and to put this in a note file I can use something called command substitution. (Typing) Now, the colon here is nothing special. I just like to add a colon after the date to show where the note starts, and command substitution should be familiar from the introductory Bash course. Now, let's Save this and see what happens. (Typing) And as you can see, now the date and time are added in front of the note. Now, the only thing I don't really like about this script is that I have to call it with a full pathname every time. So, if I'm in another directory, this is not going to work. I will have to say something like this every time. But of course I'm lazy, and I want to avoid typing these two extra characters every single time, so let's put our scripts on the PATH so that Bash will find it regardless of my current working directory. I recommend making a directory where you keep all your scripts, and the traditional name for this would be bin. Of course put it in your own directory, and let's move the script there, and then I shouldn't forget to reopen this in my editor. And now that I've put my scripts in a separate directory, I can put that bin directory on my PATH. Of course I can save this customization to my profile, and, again, that's covered in the Bash intro course. So, now wherever I go on the file system, the script will be on the PATH, and I can call it directly. (Typing) Of course I meant to type root dir there, but that's okay. So, as you can see now, I can call the script very simply like a normal command from anywhere on the file system. And because I probably will be using it from all over the file system, it's very important that here I'm using a full pathname for my notes file because if I didn't do that and just used notes. txt like this, the script would try to save the note to a file in the current working directory. So, let's try that for a moment. If I dial my current working directory as the root and now if I run my scripts, Bash is going to give an error message because I don't have permission to write a notes file in the root. So, let's put it back. And finally, let's add a last line of code to the scripts because I like to show a simple confirmation to the user so he knows that something happened. (Typing) And like you would expect, Bash will simply execute each line in order. So, after saving the note, it will print the text note saved followed by the note itself. (Typing) Very well. Let's go and look at the slide to summarize all of this.

Calling The Script

So, we saw that you cannot call a script like a regular command when it's not on your PATH. And one way to work around this is to add the location of the scripts to your command line. For example, this works when the script is in your current working directory, or this will work from any location on the file system provided of course that your script is in a location called

/home/reindert. Now, if the script is on your PATH, you can simply call it like you would call any regular commands. So, my tip for you is make a folder where you keep your scripts like I just showed you in my demo. The conventional name for this folder will be bin. Any script you put in there will be easily callable from everywhere on your system as long as you make sure to add the directory to your PATH like this. Now, if you want to know more about customizing the PATH and how to save this customization for future Bash sessions, please refer to the Bash intro course.

The Shebang

I've already showed you that it's good practice to start all your scripts with something called a shebang or a hashbang. It should be at the very start of the file, and the line should start with a hash sign followed by an exclamation mark. By the way, another name for an exclamation mark is a bang, which explains the name hashbang. After these two characters, you declare which interpreter should run your code. You can also use this line to specify options for your interpreter. Later in this course, we'll see why you might want that. In this course, we only look at Bash scripts and all my Bash scripts will start with this line #! /bin/bash. It's probably a good idea to try and remember this line because I will start every single script with it. Or actually, I wouldn't really start every script with it. If you are on another system, the Mac OS or Linux, I cannot guarantee that Bash is actually located in /bin. So, if you're writing scripts on another system or scripts that have to be portable to such systems, the line I just showed you will not work. So, for portability you may want to use usr/bin/env bash, and this will run the env command, which will look up the Bash executable in the current PATH. So, assuming that the interpreter is on the PATH, this will work on any system, but unfortunately there are some drawbacks to this. First of all, on some systems including Linux, you will not be able to pass any options to the interpreter. And secondly, the executable that gets called depends on the configuration of the user that runs the scripts because we're trying to find Bash in his PATH, and this opens a whole new can of worms. For example, suppose some user has installed a different version of the interpreter locally on this account. In that case, if he calls the scripts, he may see a completely different result from everybody else on the system. And by the way, you can't even be sure that there will be an executable called usr/bin/env on every UNIX system. And the conclusion from all this, like always in the UNIX world, is that it is possible to write scripts that probably work on a lot of systems and in a lot of different situations, but it's very hard to be sure because of the extreme flexibility of UNIX systems. So, if your scripts will only be running on Linux and Mac OS, you can safely use the first variant with /bin/bash. If you need to be more portable, try the second one, although it's very

hard to guarantee anything. I will try to give you some tips about portability here and there in this course, but I won't go into it very deeply because it's a very complicated thing.

Naming Your Script

The last thing I want to talk about in this module is giving your script a name. A mistake that a lot of people make when they start writing Bash scripts is that they name the script test, but that name is already taken by a command built into Bash, which means that we have a naming conflict. Basically, that means we will only be able to run our scripts by giving Bash the full pathname because the name is already taken. So, if you call your script something like test or if, you can only run it by giving Bash the full PATH to it. The same problem occurs when you pick a name that's already taken by another executable on your PATH like, for example, ls. In that case, when you type ls, Bash will first find the systemwide ls on the PATH, and so it will not run your scripts. You might solve this by putting your script folder at the start of the PATH variable, but I really wouldn't recommend that. Instead, it's better to pick a unique name for your scripts. So, how do you check if a name is already taken? Well, you can use the type command, and let's see how that works.

Demo: The type Command

So, let's see what type tells me when I ask about the name test. And as you can see, this tells me there's a built-in shell command called test, so I should not name my script that way. And what about, for example, cp? And this tells me that Bash already knows a command called cp. It's not built in, but it's actually a file in /bin/cp. But when I try a new name, now I now that there's no command called foo on this system, and I can safely call my script foo. Of course you should keep in mind that this does not guarantee that other systems may not have a command called foo installed by default.

Summary

And this brings us to the end of this module. We have seen what a shell script is, namely a text file containing commands to be executed by the shell and how to create one. I've showed you to take care when you name your scripts, and also that after creating the file you shouldn't forget to set executable permissions with the change mod commands. Furthermore, your scripts should start with a shebang line like this. Also, any line starting with a hash as a comment will ignored by Bash.

Now, if you want to run the scripts, make sure to add it to your PATH or call it with a full pathname.

Variables

Introduction

[Autogenerated] Hi. This is writing some liquor and welcome to this module, which will teach you how to use variables in the best script. Script that doesn't use variables is probably kind of useless. You need very bills to contain the state of your program and to remember things. And without them there's a lot of things you can't do, like use input, change stuff, do logic and arithmetic and many others. After watching this module, you'll know how to create a variable and how to assign it to value and how to use it value. I'll also show you how to pick village names for your variables, and I'll show you some important things you need to remember about using variables.

Demo: Variables

Variables allow you to temporarily store data in memory and then retrieve that data by name. Let me show you an example. This stores the string hello in a variable called greeting. I can retrieve the value by putting a dollar sign before the variable name like so. Basically when you use a dollar sign followed by a variable name, Bash will replace that with the value of the variable. So, for example, to print this value I can use the echo command. I can also use this variable just about everywhere on the command line. For example, if I want to store a filename in a variable, now the variable filename has the value somefile. text. And I can, for example, use the touch command to create a file by that name, (Typing) and I can also remove it again. So, in all these commands, the $filename part here is replaced by Bash by this string, somefile. txt. So, for Bash this really spells out ls somefile. txt, this spells out rm somefile. txt, and here again ls somefile. txt. Now, I could also have multiple words in a single variable like so, (Typing) and then I can, again, use the variable files as the argument for a command. And as you can see, my variable files contain two words, file1 space and then file2. And then when I say touch $files, this here will be replaced by file1 space file2. And so the touch commands get two arguments, and that means it creates two files. And you can see that's why I did ls -l. You can see that each of them has a single line so that means there are two separate files. Now, there are also some predefined variables like, for

example, $user, and that contains my username so I can do something like this. Please note that variable names are case sensitive, and all predefined variables are in upper case. So, suppose I make a mistake and use a lowercase name here. You can see that $user isn't defined, and Bash doesn't give you an error if you use an undefined variable name. It will simply use an empty string as its value. Now, instead of printing the value, I can also store this greeting in a new variable. And, of course, I can inspect this value again. (Typing) Oh. I used a lowercase name. Let's use uppercase again. Very well. Now, you might wonder why I'm using quotes here. Well, in a variable assignment, the value you assign has to be a single word. So, if you forget to use the quotes like this, I get an error. It says command not found, and it says that about the word reindert, which is actually the value of this part of the command here. And how does that work? Actually, in a variable assignment the value you assign has to be a single word for the shell. So, that's why you use quotes here because they take away the special meaning of this space meaning that this whole part here, this will be a single string. So, for the Bash shell in this command, it will be treated as a single word. When it's used as the value of this variable, then it's two words again. So, like we saw before here, I did the same thing. We used two filenames as the value of this variable, but then when I use it here, this gets replaced by file1 space file2, and you actually get two arguments for the touch command, which will create two files. So, that's kind of subtle, and you have to really think about that when you use variables. Now, another thing you can do wrong is to accidentally use spaces around the equals sign. For example, let's say this x = 5, and that doesn't work. When I type this, Bash doesn't see this as an assignment, but as a regular command line so it thinks I'm trying to use a command X and give it a first argument of the equals sign. So, if you want your assignments to work, you really shouldn't put any spaces around the equals sign. And another thing is that you might try to inspect the value of a variable without using echo like this. And, again, Bash replaces it with the value 5 and then thinks that you're trying to execute a command called 5, and so it says 5: command not found. Now, you might think that this is an acceptable way to display the value of a variable, but it really isn't because suppose I have a dangerous command stored in a variable like this. (Typing) Now my variable contains a real command, which will remove a file. And if I do this, not only don't I see the value so it doesn't work to display the value, but actually Bash now silently ran this command and removed my notes file and didn't even give a warning. So, make sure that whenever you want to see the value of a variable use echo, and that will safely tell you what's in there.

Variable Basics

So, variables are used to store data and give the data a name. To create a variable, you just assign it a value. So, this example will create a variable with the name X and a value 10. Of course, if a variable with the name X already existed, its old value will be overwritten with the new value of 10. So, if there was an X with the value 5 before, after this statement it will have the value of 10. Now, this is an example where the value consists of multiple words. So, we have the variable name of filenames, and it contains three files names separated by spaces. Now, don't forget to use quotes here. What that does is it changes the meaning of the space from separating words to just being a space in a string, and so the whole string containing these three filesnames will be one value, and this entire string will be the value of the filenames variable. Also, don't forget that you shouldn't use whitespace around the equals sign. Now, to get a variable's value, you prefix its name with a dollar sign. The most basic use is this where you use the echo command for printing the value. Now, in case you are used to programming in a strongly typed language, it's important to realize that Bash doesn't really know about data types. Simply put, a variable just contains a string. Of course a string can contain numbers, so it is possible to count or do arithmetic with Bash, and we'll see how to do that later in this course. Now, when picking a name for your variables, Bash will only allow you to use letters, numbers, and underscores. Any other characters can't be part of a variable name. So, if you use any, Bash will assume that your variable name has ended. Furthermore, the first character should be a letter or an underscore, so you can never start a variable name with a number. Also, variable names are case sensitive like I showed you in the demo. Now, all of this still gives you a lot of freedom to name your variables, but I want to discourage you from using uppercase variable names because all of the predefined variables have uppercase names. So, if you accidentally change the value of them, that may change the behavior of your script in unpredictable ways, so you don't want to override these by mistake. In other words, it's a good habit to use only lowercase names for your variables.

Using Variables in A Script

So, let's go back to our note-taking script and make it interactive. (Typing) The read command will read user input and put it in a variable. In this case, I'm calling the variable note, and as you can see I'm not using a dollar sign here because I'm simply passing the name of the variable to the read command. Now, to save the note to my notes file, I have to get its value like this. Let's run this and see what happens. (Typing) Very well. That works, but it's not very user friendly that my script doesn't explicitly ask for input. It just sits there like this saying nothing. Let's see if we can use read to show a prompt instead. Now, the read command you should know is a built-in command in Bash, and there are two ways to get help. Of course you can try to use the man

pages like this. And, as you can see, this shows me the man page for built-in commands for Bash, and you can see that read is there as well, but I can't guarantee that this is going to work because there might be something installed on your Mac that's also called read, and the man page might show you that page instead. Another way to look at this man page is to say man builtins, which will take you straight to this page. But then still it's a huge page, and it lists lots of built-in commands, and I don't think it's really nice. So, I think what's better to do is to use help, which is a Bash command, and it will give you documentation specifically for Bash built-ins. So, if I say help read I get a tiny little page like this, and here I can see that the -p option will let me show a prompt to the user. So, if we try that, I can say here -p and ask for your note. And then if we try that, we say that's a nice prompt. Very well. Now, another thing I'm going to change is to add a date variable to improve the readability of my scripts. I like to put the initialization of variables at the start of my scripts, so let's say date and use the command substitution now to take the output of the date command and put that as the value of my variable. Let's add some documentation as well, get the date, and we can use it simply like this. Now, suppose I like to organize my notes a bit. I want to have various note files for different topics. So, for work-related notes I want to use worknotes. txt, and for my shopping list I'd like to have shoppingnotes. txt, etc., etc. And I'd also like to be able to pass an argument to my script that tells me the topic. Now, we already know there's a special variable called $1 that holds the first argument of my scripts. And let's start by reading that, get the topic, and now we can simply use topic variable in our file name. So, I can do it like this, $topic. But, of course, you will understand that this is not going to work. The code we have now will try to read a variable called topicnotes. It won't try to read a variable called topicnotes. txt because a dot isn't allowed in a variable name. So, Bash knows that in any case the variable name will stop here, but still it's too long because we don't have a variable called topic notes. We only have a variable called topic, and we have to tell Bash that that's the one we want to use. So, what I'll do is use braces. This tells Bash that I want to use a variable called topic and that the variable name actually ends here. Now, let's try this. Let's try a shopping-related topic. I want to buy milk, and let's see if now there's a file called shoppingnotes. Yes there is, and it contains my notes. Very well. By the way, if I don't use an argument like this, if I just call tn, then the first argument will be empty, which means this will evaluate to the empty string, which means the topic variable will contain an empty string. So, here this will just evaluate to the filename notes. txt. So, if I don't use an argument, I don't get a topic. My note simply ends up in the default notes file, which is really a nice little feature. Now, in the next module we'll see how to check whether a parameter actually is present on the command line. Now, let's clean up our script a little before we go on. First of all, I'm going to add a filename variable to make stuff easier to read. (Typing) And, of course, the filename will just contain this little bit here. So, instead

of using this I will use the value of the filename variable here, and let's also make our output a little more verbose. So, let's say note saved to this filename. Now, I like to put quotes around the note like this. The problem is, as you should know, single quotes escape every single character between them, so the dollar sign here doesn't get interpreted as meaning that we want to have the value of a variable. It just gets used in the string so this will print $note. Let me show you for a moment. If I do echo $note in single quotes, it will simply print $note. It doesn't take the value of any variable. So, if I want to print quotes around the $note here, I have to escape the quotes themselves. And if I do this, then it will print quotes, and in between the value of the $note variable. So, let's Save this and let's try again a work note. Let's say that I have to work harder. And, as you can see, there's a bug now here because it saves it to notes. txt and not to worknotes. txt. And why is that? Well, actually I set the filename here, but I set the topic here. So, when the filename is set, the $topic here still contains or still evaluates to the empty string because topic isn't set yet. So, let's move the declaration of the top variable up a little bit to make sure the topic exists before we try to use it, and let's run it again. Work harder! And now you can see it will actually save it to worknotes. If we print that file, you can see it contains my note now. Now, our script is turning into a handy little program already. The last thing we need to do for now is to fix another nasty bug because what do you think will happen if a user uses a topic that contains multiple words like this? Well, of course, the second word, things, here will be given as a second argument for our script, and our script will totally ignore it because it only looks at the first argument. So, let's not worry about that for now, but a smart user will, of course, do this. He'll group the two words together to make a single argument that contains a space. Now, let's see what happens if I do this. Well, it seems to work to start, but then Bash gives a strange warning. It says ambiguous redirect. And why is that? Well, actually my $1 variable here now contains a string with spaces in it, other spaced things. So, Bash sees a redirection here followed by multiple words with spaces between them, and this confuses Bash because it only allows redirection to a single filename. To fix this, we have to quote the user the variable here. And what this does is because we use double quotes, they leave the meaning of the dollar sign intact. So, this here will simply evaluate to the value of our variable, which will be other space things. And then to make that into or more correctly, of course it will be my home directory followed by other space things followed by notes. txt. Anyway, it contains a space. And because we have quotes around that here, this will be grouped into a single argument into a single filename to redirect to. So, that's why now we see, I didn't really mean to type ls there. That's just a matter of habit I guess. Let's try our script instead. And you see Bash doesn't give a warning here, and we can now say very well it worked. In general, it's good practice to always quote the user variables to prevent these kinds of surprises. You should also make sure to use double quotes and not single quotes as I showed you.

Now, let's fix this for the entire script. On this first line here the $date here is a command substitution and not a variable so we don't have to quote that, but here for the topic, yes I'm going to use quotes. And here there's another thing. You might be tempted to do this, and it's not really wrong either. And, of course, if you use double quotes you can remove the braces. Because of the quotes, Bash knows where the variable name starts and ends. But personally, to make things a little bit clearer, I prefer actually to quote the entire string because that kind of denotes for someone who is reading the scripts the entire value I'm putting into the variable, but then again, we have to brace the $topic here. Otherwise, again, Bash will take a topic note variable and put in an empty string. And then there's another thing you should know, and that's that tilde does get escaped by double quotes. So, the dollar sign stays in text, but the tilde gets escaped. And instead of the tilde, I'm going to use a predefined variable called ${HOME}. And, again, I'm using braces. Even though it's not really necessary in this instance, I think it's very nice to do because it makes very clear where our variables start and end. Very well. Let's fix that here as well for the date and the note. Again, let's quote the entire string. I just think it's very clear, and let's do that here as well. And let's do a final test now. (Typing) And the final fix now here is that you see that now actually the backspaces end up in our output, and that's because here we're in double quotes. Stuff gets escaped. We don't need the backspaces because the single quotes will be escaped here. So, again, last time, last test, and now you see our output is very neat without any garbage.

Using Variables: Good Habits

So, maybe the most important thing to take away from this demo is that you want to surround your variables with quotes. This will prevent surprises when your data contains spaces. Actually, there are some cases when you don't want to use quotes, but we'll get to that later. Now, make sure to use double quotes because using single quotes will escape the dollar sign, and you will end up with a string with a dollar sign in it. Double quotes keep some special characters intact, and one of those is the dollar sign. This makes it possible to use a variable in a string escaped with double quotes. We also saw that it can be wise to use braces to tell Bash where the name of your variable ends. For example, this will get the value of a variable called foo and to pend the string bar to that. But this will get the value of something called foobar. It's very important that you're aware of this, and it's a good habit to use braces wherever there may be confusion. Some people argue that you should use braces everywhere, but in my opinion that's just overkill. Finally, another thing we saw in a script, it's usually better to use $HOME instead of the tilde. This is

because in some contexts, like when you use double quotes, the tilde may not be expanded by Bash, and this will give unexpected results.

Reading Input

We also saw that we can use the read command to get user input and assign it to a variable. This will assign the input to a variable called var. Now, read is a built-in command, which means you can use the built-in help command to get info on how it works. Now, if you insist on using the man pages, try man builtins because man read may give you a page for something else called read. Finally, one of the options for read is -p, which will show the user a friendly prompt.

Debugging your Script

Now, finally I want to have a short note about debugging. We already saw some nasty bugs caused by spaces and arguments and stuff like that. Now, what do you do if something like that happens to you? How do you find what line in your script causes the error? Well, there's one technique you really should know about, and that is calling Bash with the -x option like this. And when Bash gets called with -x, what it does is it prints every single line it executes including the values of the variables that get used in these lines. So, for example, if I Save this and now run with let's say shopping, you can see it skips the comment lines so all of these lines get skipped, and it starts here, and you can actually see that at first it calls date, and then it puts the output of date here in the date variable. And then you can see the topic now has the value shopping. And then you can see that in this line the actual value that this gets evaluated to is this filename. So, all of this will make it a lot easier for you to debug your scripts. Now, it might be quite annoying that Bash really outputs every single line in this way and you have to step through your entire script, and sometimes you just want to debug several lines in your script. And there's also a way to do that. So, in that case what you do is you don't call Bash with -x. You just call it normally, and then starting at the line you want to debug, you set the -x switch. So, set is a built-in command that lets you set options to Bash. And then when you want to stop debugging, you set +x, and that turns off the debugging again. And in this case if I run my script, it will only debug the part where I set the filename as you can see. So, in case your script does something you didn't expect or you get errors you don't understand, you can always try to use the -x switch to find the error in your scripts.

Summary

And this brings us to the end of this module about variables. We saw how to assign and how to get a variables value and why it's important not to use whitespace around the equal signs when you assign a value. I explained what a valid variable name is in Bash and that you should use lowercase names for your variables. When using variables, make it a habit to use quotes and braces because that prevents strange bugs and makes your scripts more readable. Finally, we saw how to read input with the read command, and I gave a short word about debugging with the -x option either in the hashbang line or by using the set command to enable with set -x or disable with set +x.

If, Then, Else

Introduction

[Autogenerated] Hi. This is very delicate and welcome to this module where we'll discuss how to make decisions in your script using, if them else. Now any really useful computer program needs to be able to react to its input and behave differently, depending on what kind of data it receives. And chill scripts are no different in this module. I'll show you the basics off decision making in a shell script. We'll start with looking at the if then l statement, which is the basic mechanism to do this. Then I'll discuss returned coats, which are the values that if then l statement uses to make decisions. And finally, we'll see some special best Syntex for testing all kinds of values, called a conditional expression.

Demo: The If Statement

So, let's start with a little demo of what the if statement does. Basically speaking, it allows you to test whether a command succeeds and then react to that. Let's start with a simple example that tries to create a directory and then prints a message. You can see that the if statement starts with the word if followed by a command, in this case making a directory, and we tell Bash where the command ends with a semicolon. If the command succeeds, the command after then will be executed, and the word ok will be printed to the terminal. If the command fails, the part after else gets executed, and we'll see the word error printed. The end of an if statement is denoted by the word fi. So, let's run this. Now, the mkdir command has been run, and a directory has been made. Since the command succeeded, the then part of the statement is executed, and ok is printed. Now, if I run the same command a second time, because the directory already exists, mkdir now

fails, and our if statement causes the word error to be printed. Now, of course, using if, then, else makes much more sense when we use it in a script, so let's go back to our note-taking script and add an if statement. Now, let's start with simply checking whether the user actually typed anything. Of course we start with the word if, and we do that after reading the input. Then, to check if the string is empty, we do the following. I'll explain this a little better in a moment. (Typing) Again, we need a semicolon before the then keyword. After then we execute our normal actions because we know that there is user input. For readability I will indent the code, and what we want to do are just the normal things we already did normally. And now for the part where there is no user input. (Typing) And we end the if statement with the fi keyword. So, now if we run the scripts and I don't give any input for a note, the script detects that and shows that nothing was saved. On the other hand, when I do type something, it does save it like before. So, let's see a slide for a more formal discussion of the if statement.

The If Statement

The most basic use of the if statement looks like this. First, there's the if keyword followed by the code you want to run and see whether it returns success. Note the semicolon at the end here. Don't forget it. After the semicolon comes the keyword then. By the way, you can also put the then on the next line and leave out the semicolon, but you don't see that very much. Now, in case your code reports success, the part after the then gets executed. This can be any list of statements. In case the command fails, nothing will happen at all. And, of course, don't forget to include the fi keyword to end your statement. Now, what if you also want to execute something when your test code fails? Well, we've already seen that. In that case, you use an else clause. The code after else will be executed when the test code fails. Again, you end this with the fi keyword. Now, note that else and fi both are on a new line, and the reason for this is that these keywords have to be the first words in a command so that Bash knows that you're starting the next part of your if statement. The same is true for the if and then keywords by the way. Each of these keywords have to be the first of a command. To make this more clear, let me show you another way of writing an if statement. That's the one we started the demo with, and this is how you would use an if statement on the command line if you ever feel the need to do that. On the command line, we want our code to be in a single line, and the use of semicolons makes this possible. So, in general, the if, then, else, and fi keywords need to be the first word on a line or come after a semicolon. Also, might you forget this syntax, you can still use the help command to ask Bash about it.

Return codes

Now, you're probably wondering how Bash knows whether a command succeeded. Well, actually, my explanation up to here wasn't formally correct. In reality, when a UNIX program ends, it returns a number between 0 and 255 called a return code or exit status. Now, the convention for this is very simple. A value of 0 means success, and all other values are errors. Of course a program may choose to return various different values for different kinds of errors. Now, just about every UNIX utility follows this convention, so this is why in the demo I could easily test whether the mkdir had succeeded because mkdir returns 0 when it succeeds and another value when it fails. Of course it's never a bad idea to read the man page for a command to see what it tells you about its return codes. Now, if you write a shell script, you can use the exit command to return a value. So, if you call exit 0 your script will end and return success. Of course it's important that you make it a habit to write scripts that follow the UNIX convention. In other words, make sure your program always exits with a value and with a correct value. And basically all of this enables the if statement to do what it does because it simply looks at the return code for the code I just called testcode in the previous sheet. So, the code you call after if is what gets examined for each return code, and that determines whether the then or the else part gets executed.

The Conditional Expression

Bash has a special syntax for doing various logical tests. It's called a conditional expression, and it lets you do several things. You can use it to perform various kinds of tests on files like whether they exist, what kinds of permissions are set, whether it's a file or a directory, and much more. You can also test a string to see if it's empty or equal to another string, and also you can do several kinds of arithmetic tests. So, how does this work? Well, first of all it looks like this, two brackets followed by a space and an Expression, then another space, and then two closing brackets. Some examples would be just use the variable value on its own to check if it holds any value at all. So, this expression will return true if str holds any value at all apart from the empty string. If it holds the empty string or if the variable str isn't set at all, it will return false. Another example would be this, using the equals sign to check if the variable holds a specific value. So, in this case, if the string that is the value of str is exactly equal to the string something, this expression will return true. But watch out. It's very important to put spaces around the equals sign here. If you don't, like in this example, the whole expression str is something will be read as a single word. And since that word is not an empty string, the conditional expression will always return true because, as we saw in the first example, if you use the conditional expression with simply a single value in

between, it will check if that value is not empty. And in this last example because there are no spaces around the equals sign, the conditional expression will think oh it just has one argument and it will always return true. So, a completely different example would be to use the -e operation, the -e option to check if the filename variable holds the name of an existing file. And this does pretty much the same, but returns true if the variable holds the name of a directory. Now, keep in mind that as always the correct use of the whitespace is very important. If you put the expression immediately after the two opening brackets, Bash will think you are calling a command with a name that starts with two brackets so you will get an error. And the same is true for the space that comes before the two closing brackets, and, again, the same is true for spaces around the switches -e and -d and spaces around the equals sign like I explained.

Demo: The Conditional Expression

So, let's try and use an if statement in a new script. While creating this course, I'm constantly creating and testing new scripts, and I'm getting tired of creating new files and setting permissions, so let's write a little script that will facilitate the creation of new Bash scripts. It seems like a nice meta-task right? So, let's call it create_script, and there's some comment here already. This script creates a new bash script, sets permissions, and more. Now my script needs a single argument, and that is the name of the file to create. So, let's start with testing that there is actually an argument. (Typing) Now, remember that $1 here holds the value of the first argument, and using it on its own in a conditional expression will test that it's not empty. Now, using the exclamation mark here negates the tests. I would normally pronounce this character as not so you can read this code as if not $1; then echo "Missing argument. " So, if $1 is empty, we do this. Also note that I will immediately exit the script with the exit command and return count of 1 signaling failure. This will make it possible to use this script in another script if I want to. Now, let's go on and set up some variables. First, I'm creating a variable that points to my bin directory, (Typing) and then let's create the filename for the new file. (Typing) Now, note the use of quotes and braces around the variables here. They aren't really all necessary, but using double quotes every time you get a value from a variable is a good habit in my opinion, and I think that braces improve readability when I use a variable in a string with other data. Now, let's check whether a file with this name already exists in the bin directory. So, this is just the general framework for an if statement of course. And to check if a file exits, we use -e, and then we use the filename we just set. And let's say if it already exists, and, again, exit. Now, of course, it's also important that our bin directory already exists, so let's add that as well. Now, this part of code I'm going to copy in again, and this is an example of a nested if. Since the then part of the if statement can contain any

code, it can also contain another if statement. So, this statement will create a new bin directory. If it succeeds, this message is printed, "created ${bindir}. " If not, it prints an error and exits. And all of this code, this whole if statement, gets executed only if the bin directory doesn't exist already. Remember, -d checks if this is actually a directory, and not -d negates that. So, actually what this does is it succeeds only if $bindir doesn't evaluate to the name of an existing directory. And in that case, we go into this if and try to create one. Now, again, note the quotes around my variable here in the mkdir statement. I don't think it's possible to overestimate the importance of these. By now, you should know what could go wrong if I forget the quotes here. Now, if we end up here at the end of this code, we know that the bin directory exists and that there's no file yet with the name that's given as the first argument to our script. But now I'd also like to check if there's no command with the given name either, so let's check that as well. First I'm going to add a new variable. Now, the scriptname variable just holds the name of the file without the directory name. Let's fix this line as well. And now I want to check whether a command with this name already exists, and I want to do that before I check the bin directory and start making changes in the file system. So, the right spot to do that seems to me to be here after checking whether the file and the bin directory already exists. Now, as you can see, this executes the type command, and if it returns success, that means that the command is already known, and we exit the script. Now, before doing any actual work, let's test the logic so far. I'm going to add an echo statement at the end to see if we reach the end of the scripts, and then we're going to test. Of course it's always a good idea to exit with a successful return value. Very well. And let's test. And, as you can see, I forgot to set executable permissions again. Let's fix that. Okay. Now, let's try again. And apparently my first test works, and it gives an error when I forget to give an argument. Now, let's try the second test. We already have the script called tn. Will our script detect that? Yes, that works as well. So, what if I try to create a script called if? Very well. This second test works as well. Now, if is a built-in command, so if yours clashes with the built- in Bash command, will it also clash with a command that's on the file system like /bin/ls? Yes, that's also detected. Very well. Now, you may notice that the first line of output here doesn't come from my scripts. It's actually the type command that sends this to the screen. And if I try something that doesn't exist as a command yet, we reach the end of the code, which is correct, but we also see an error message here, which is an error message generated by type. Of course you don't always want the programs you call to mess up your script's output, and we'll see how to deal with that later in this course. For now, let's just accept that some commands generate output. Now, to test whether my script will actually make a new bin directory if none exists yet, let's move my current bin directory out of the way. And now, of course, I have moved my create_script as well so I have to call it with a full path name. And now you can see that the test for the bin directory works and created a new

directory. Let's check that to be certain. Very well. All the tests work. Let's get some work done. So, I just removed the last two lines, and now I'm copying in some code. So, this should create my file with a single line, the bin/bash, add permissions to that file, and then open it in my EDITOR. $EDITOR isn't always predefined, but it's one of these environment variables that users can customize to set their preferred EDITOR. Now, let's try this. Of course, don't forget to move the bin directory back. Oh, I forgot to save the script. Let's Save it and try again, and my EDITOR doesn't seem to open at all. Does my file exist? Yes it does. Does it contain any data? Yes, it contains the hashbang, and this star here also denotes that it's executable. So, apparently almost everything went right except the last line to open with my EDITOR, and this is because Mac OS doesn't set the EDITOR variable by default. So, this variable expands to the empty string. That means my new script file is actually the first word on this line, and my new script actually gets executed, but it doesn't contain any code so nothing happens. So, let this remind you not to make assumptions about whether a variable is set or whether a command or file exists. Always test these things in your script before doing anything or strange things may happen. Now, let's fix this as well. And, again, here we test if the EDITOR variable is set, and if it is we try to execute the EDITOR and open the filename. Also note that here I escaped the dollar sign in the string, which takes away its special meaning, and the string will contain the text $EDITOR. Of course you could get the same effect by quoting the entire string with single quotes. Now, let's try again, Save the file, and run. Okay, this doesn't work because it already exists. Let's make a new file, and it detects my EDITOR isn't set. Now, in case I want to set my EDITOR, let's say export EDITOR. I want to open this file in Emacs, and you can see that now my file opens in Emacs when I run the script. Now, if you want to be really, really perfectionistic, you might argue that I also have to test whether my file was actually created and whether the permissions were actually successfully set, but I feel it's pretty safe to assume that the bin directory is writable and that change model will work. But maybe it's a nice exercise to write some tests like these by yourself. By the way, you can use help to find out which tests are supported like this. And this is the help page for the conditional expression, this expression here. But you'll find that it actually refers to something called tests here, the test builtin. And if we open that help test, this actually contains most of the information you need. So, here you can see all the possible string operations, file operations, all the different tests that you can do can do with a conditional expression. So, apparently there is some built-in command called test that does more or less the same thing as the conditional expression. Now, let's look at some slides again so I can explain how this works.

The Conditional Expression 2

The story goes like this. Historically there was only a command called test, and this is still there in Bash for compatibility. Another name for this command is the opening square brackets. So, you can use the test command in an if statement just like we did up to now, but it uses a single square bracket instead of two brackets. The thing is, since it's an actual shell command, it gets parsed normally, and everything you pass into it gets parsed like a normal argument including things like expanding wildcards or a command substitution, and this makes it quite hard to use. There are a lot of tiny pitfalls you need to know about. And although it's important that you know it exists, I would advise you not to use the old-fashioned test command unless you really need to write a shell script that will run on non-Bash shells as well. Now, the syntax I showed you so far with the double square brackets is actually a Bash extension. It works pretty much like the test command, but it's a lot easier to use. And the reason for that is that the two brackets are not actually a command, but built into the Bash syntax. Everything that goes between the brackets get parsed in a special way, and that's why, for example, you may have noticed that I didn't quote my variables when using conditional expressions. And that's just one of the benefits of using this syntax. So, basically when writing scrips for Bash use the double brackets instead of the single brackets. Now, still when you need to get help, help test will show you most of the support operations. And if you need to know more about the Bash specific extension, try this, and that will show you some of the special operations that are Bash specific.

Arithmetic Tests

I already showed you how to test files and stings, but now I'd like to tell you how to test numbers. You can only compare integers because Bash doesn't handle floating points at all. Now, the syntax looks like this, and it's basically the same as when you're comparing strings, but the difference is that the operators are different. To name some, for equality, so to test whether two numbers are equal, you use -eq. You use -ne to test whether they are not equal. -lt means less than, and -gt means greater than. Now, there are some others, and if you want to know about them use the help command. The main thing to remember is don't use the normal operators for numbers you may be used to because these work on strings only. So, the normal greater than sign will compare strings and look at their alphabetical order. So, don't use the mistake of using the equals sign or the greater or lesser than sign to compare numbers because that will probably not work the way you wanted them to. Now, there are several ways to get numbers you may want to compare. For example, there are two special variables you should know about. First of all, there's the $# variable, which contains the number of arguments the script received. So, this can be quite useful if you want to check if you received enough arguments. Now, the second one is

the $? variable. This contains the exit status for the last command you ran. I really don't use it a lot, but sometimes it can come in handy. Another very important thing to know is that to get the length of a variable's value, use this syntax. So, let's go and try some of these.

Demo: Arithmetic Tests

So, let's see a little example. I created a script that takes two directories as arguments and determines which one contains the most files. Looking at the code, I start with checking the number of arguments. The special variable $# contains the number of arguments for my script, and the test here returns true when this number is not equal to 2 in which case I end the script with an error message. When I'm sure there are exactly two arguments, I check each of them for being a directory. This here checks whether the first argument is a directory and this block here does the same for the second argument. I used the -d test to check that they are actually directories and not just files. Also note that I'm using the exclamation mark to negate the tests, so this returns true when the first argument is not a directory. When that's done, I'm going to put these values into variables called dir1 and dir2, and here I'm using a command substitution. I'm calling ls to list the files in the directory and then piping that through wc, which will count the number of files in the output. So, after that, these two variables, count_1 and count_2, will contain the file counts of both directories. And then here I use a last if statement to check which one of the directories contains the most files. Now, as you can see, this if statement contains a keyword I haven't mentioned before. It's elif. You can think of it as short for else if. So, first I check whether the first value is greater than the second, so I check whether the first directory contains more files than the second. If that's the case, I can print that to the terminal, and then I exit the if block, so I exit the script here. If the first directory does not contain more files than the second one, I'll have to check whether they're equal, and in that case I will print this. And if this test here isn't equal either, that means that the second directory must contain the most files. So, in that case I print that directory has the most files. Now, basically this is exactly the same as writing this, so converting it into a nested if, (Typing) and you can see exactly what the meaning of the else/if is. So, this test is done first, and if it succeeds, this is executed, and we exit the if block. If this test doesn't succeed, we go into this block, and, again, we first do this test. And if that fails as well, we do this. And basically shorthand for the else/if here, this combination is simply elif. And in that case we don't need the extra indentation, and we also don't need the extra fi keyword. Now, I'm not going to run this script right now, but I will include it in the course materials so you can check for yourself that it works. By the way, we can easily transform some of the code from this script to compare the length of two strings. So, let me show you another piece of code. I'm going to make

a new text here, and basically this script does more or less the same thing. It checks whether there are two arguments, and here it has the same kind of if statement to check which one of the two values is largest, but the main difference is we don't count files in directories. We just see what the length of the string is, and for that we'll use the special syntax we saw on the slide, which is a dollar sign with braces and a hash at the start of that. Now, the rest of the logic is pretty much the same as the other script, and I'll add this to the demos file as well. So, in case you have access to the extra course materials, you can see these scripts as well in there.

The If Statement Revisited

Now, there are two ways to use the if statement we've seen that I want to mention in this slide. First of all, you can nest if statements. In the example on the right, the inner if statement is contained in the then part of the outer if statement. In other words, when the outer if is true and the variable bindir is not a directory, the inner if gets executed. Another thing we saw is that there's also an elif keyword. This is short for else if. So, if the first test fails, the test after elif gets executed. Now, you should note that in this case the else belongs to elif. So, the part after else gets executed only when the first test fails and the test for elif fails too. You can also use multiple elifs in a single statement. Every time a test fails, the next elif gets tried until one succeeds. You can include an else statement in the end that gets run when no test succeeds at all. So, in this example we see if the first argument of our script contains the name of an animal. If it contains cat, we print meow, but if it doesn't we check if it contains dog and print woof. If it doesn't contain dog either, we test if it contains cow. If none of these tests succeed, the else clause gets run, and we say unknown animal. Now, you can compare this statement with a switch statement in languages like C# or Java.

And, Or, Not

Now, just some last remarks to wrap up this module. When you use a conditional expression, you can use an exclamation mark to negate the test like so. This will return true when the file doesn't exist. Now, remember to use spaces around the exclamation mark or Bash will complain. It's also possible to combine tests with two logical operators called and and or. The and operator is denoted by two ampersands like you see here. So, as an example, this test is true if the first test and the second test succeed. That's why it's called the and operator. The first test checks that the number of arguments is exactly 1, and the second test checks that the first argument has the value of foo. So, combining this gives a test that's true when there's exactly one argument with

the value of foo. Similarly, there's an or operator denoted by two pipe symbols. For example, this test is true if the first test or the second test succeeds. By the way, it also succeeds if both tests succeed. So, the complete test is true if a or b contains a value or when both have a value. Now, there is one pitfall. If you look at help test, it says that you can use -a and -o for and and or. Well, don't do this. It will give you nothing but trouble. The -a and -o switches are for use with the old-style test command, not with the modern conditional expression. So, just use the double ampersand and double pipes I just showed you.

Summary

So, what did we see in this module? First, I showed you the if, then, else statement and how to use it. Then I also showed you that you can use another keyword, elif, and how to use nested if statements. Now, an if statement can work with any command as long as it returns a meaningful return code. The normal UNIX convention is that 0 means success and that any other value means failure. You can return a status code from your script with the exit command. This example will exit with value 0. Apart from using if with normal commands, there's also special syntax called a conditional expression. It looks like this. And I also explained to you that you normally shouldn't use the single bracket syntax or the test command because it's old fashioned and hard to use correctly. But although that's the case, you can still look at help test to see all possible tests. Of course you can also use help for the two bracket syntax. In the help, you will find how to test values for strings, files, and numbers. And finally, I talked a little bit about the logical operators and, or, and not. Now, before you go on to the next module, I strongly want to encourage you to get some practice and write some little scripts for yourself. This will greatly help your understanding and the effect of watching the rest of the course will be a lot greater.

Input and Output

Introduction

[Autogenerated] Hello. I'm running on backup and welcome to this module where I'll discuss how to do input and output with the best fruits. Now I'm guessing I don't have to explain why it's important to know about input and output after watching this module. You know some more about how to generate nice output for your script, including more about the Echo command, and knew more. It found command called Prince F, and we'll see some more about reading input with

the res commands. We'll also look into the three standard streams that your script uses for input and output and how to manipulate the streams using redirection.

Output: echo and printf

Now, let's start with the very basics. You already know about the echo command and that it prints its arguments to the standard output followed by a newline. It also takes the following options: -n will suppress the newline at the end, and -e allows you to print several escape sequences like \t for tab and \b for backspace, and there are several others as well. Now, keep in mind that using these options will probably not work in other shells and is generally discouraged. Instead, if you want to do a more sophisticated output, you can use another command called printf. Now, let's look at the next slide for that. So, if you want to do any kind of sophisticated formatting, you're better off using printf than echo. To enable you to get a nice output, printf takes a format string as its first argument. Also, it doesn't print a new line at the end of the output by default. Now, instead of explaining how that works right here, I'm going to show you some examples. So, let me show you some of the basics of printf. First of all, I can call it with only a format string like so. Remember the format string is the first argument. And you'll see that this works more or less like echo with one big difference. It doesn't print a new line, so my command shell prompt gets printed immediately after it on the same line. Now, to fix that I can include /n, which is the escape sequence for newline in the first argument. And then it's pretty much the same as echo. Now, suppose I want to print a variable. Let's print a greeting for the user. We've seen this before with echo, but with printf you do it a little bit differently. First of all, I'm going to do the format string. I'm going to say hello, and then I'm going to do this, %s, and let's say how are you? And what this means is that this is the format string. It's going to print hello, how are you? And in between here it will print a string. And it will expect to read that string as another argument. So, here I'm going to say $USER. And remember that Bash will replace $USER here with the value of that variable, and then printf will read this argument and put it on the spot of this code here. So, if I press Enter now, it prints hello reindert, how are you? Now, again, I shouldn't forget to add \n, and now it prints it on a single line by itself. Now, you can also give printf more arguments than it expects in the format string. Let me show you. So, what will this do? Well, I have a P here, then the value of some string, and then a T. Of course I'm almost forgetting the newline here, so let's fix that. Now, let's see what happens when I press Enter. So, what does printf do? Well, it takes every argument in turn and puts it in here, and then for every argument it prints this output. So here, the first time this gets replaced by the A and the second time by the E, etc., etc. Now, of course I can also have a format string that takes multiple strings

like this. (Typing) Now, what happened here is that the first $s here was replaced with the value of $USER and the second one with the value of $HOME, and that's why it prints reinderts home is in /Users/reindert. And, again, I forgot to use \n, which is kind of stupid. Well, well. Now, printf can do many, many more things, and one of the nicer things it can do is it can format your data for you. Let me show you an example of that. (Typing) So, let's look at this command. What does this mean? Well, first of all, this here is a command substitution, and it will put the output of ls here onto the command line. So, Bash will replace this with a list of the files in this directory. Then you can see that printf will take three arguments in turn. Those are these percent codes here, and the 20 here means that it will make sure that they are printed in columns that are 20 characters wide. So, it will make sure that it will pad the files names with spaces until they are printed as exactly 20 characters wide. So, let's see what happens when I press Enter. And here you can see I get a very nice column-based view of the contents of my home directory. Now, one last thing I want to show you is that printf can also save its output to a variable. So, let me show you. Let's save this into a variable called greeting, and now printf will generate its normal output, but instead of printing it to the standard output, it will save it in the variable greeting. Very well. Let's go back to the slides. Now, printf gives you all kinds of ways to pretty print your data, but I'm not going to show you all the possible options here. To find out more, I suggest you look at this link, which will give you a page on the Bash hackers wiki, which is an excellent source of information on Bash. Of course there's a help page for printf as well, but that really doesn't tell you a lot, and instead you can take a look at the man page for the C function of the same name. Now, on some systems like Ubuntu Linux, even this won't give you all the info you need, and you'll have to explicitly ask for the printf page in section 3 of the man pages like this. By the way, one option that may come in handy now and then is -v, which puts your output in a variable.

Input: read revisited

So, now we've discussed the basics of output. Let's see a little more about the basics of user input. The command for this is read, and it reads input into a variable. So, read x will set the user input as the value of X. Now, you can also choose not to specify a variable name, and in this case the value will end up in a variable called REPLY. Now, there are also some options that are good to know about. First of all, -n lets you read a specific number of characters. I'll show you one example of this in a moment. If you use the lowercase N, read will stop reading input when it encounters a new line, and with the uppercase N it will keep reading until the exact number of characters have been read. -s will not echo whatever you type to the output, and this is nice in case you need to safely read a password from a user. This is nice in case you need to safely read,

for example, passwords from a user. Something else you need to know is that normally read allows the user to input escape sequences and multiple lines of output. This is a very strange feature, and it's hard to think of any use for it. Basically, all it does is give you weird and unexpected results now and then. Now, in case you use -r, all of this is disallowed, and you simply receive the raw input, which is generally what you want. So, it's a good habit always to use -r when you use read. Now, there are several more options that can be useful, and of course we can use help to find out about them. One last thing, read can also split your input into multiple words and assign each of them to a variable. For example, this will read two variables, X and Y, and the first word will be read into X and everything after into Y. So, if the input is 1 2 3, the first word being 1 will be read into X and the rest, 2 and 3, into Y. Now, the characters that determine what separates words from each other are in the variable IFS, which is short for Input Field Separator. This defaults to space to have a new line, but you can set it to anything else if you want something else to separate your words. Now, let's go and look at the little demo. So, let's start by simply reading some input. So, what I do here is I read some input, and it ends up in the REPLY variable, and you see I have a semicolon here that ends the read statement, and the next statement I immediately print the REPLY variable to the terminal. So, let's try this. And like I said, this will allow the user to input escape sequences. And one example is a backslash. And the user can't actually normally simply input a backslash because if he tries to and presses Enter, now the new line from the Enter has been escaped and I end up in my secondary prompt, which for most users will be extremely weird, and they'll have to end it with either Control+C or another Enter, and that just seems weird. And then if you see what REPLY actually contains afterwards, it prints an empty line because it simply contains a new line and no backslash. So, that's kind of unpredictable. So, in order to actually input a backslash, the user will have to enter two backslashes, the first escape and the second one. And then, as you can see, REPLY actually contains a single backslash. Now, like I said in the slides, you can fix this by simply using the -r switch for read. And what that does is it disallows escape sequences so a backslash will simply be a backslash and REPLY will also contain a backslash, so that's exactly what you want. Now, I'd also like to show you what the -n option does, so I've written a very simple script for that. Let me get it. And this script is called areyousure. What it does, it asks the user first are you sure, and then it calls the read command here with the -r option of course because that's a good habit and the -n 1 option, which means I'm only waiting for a single character or, in other words, a single key press. And all I want to do is check whether the user presses the Y or the N key for yes or no. Then there's an if and else statement here that checks whether the answer is either a Y or an N, and as you can see this is not a normal string comparison. This is actually a pattern matching expression, and we'll see more about that later in the course. Now, there's something else we

haven't seen yet, and we'll see more about that soon too, and that's the while loop. And what this will do is it will repeat this part. It will repeat the whole if and else part until this string answered isn't empty anymore. You can see I start with setting answered to the empty string so the first time we go in here this will be false and we go into reading the answer. And then depending on the key that's pressed, we either set answered to yes or no. And if the key pressed isn't Y or N, answered won't be set to yes or no, and the whole loop will repeat again. That will go on as long as necessary until the user presses either Y or N. And then at the end when we exit the loop we actually print a newline followed by a string followed by another newline, and this string here will be $answered, which will be yes or no. So, I wrote all of this just to show you what we can do with the -n option for read. So, let me show you what will happen. I'm going to make this window a little smaller. Very well. And let's say areyousure. Well, it prints are you sure yes or no. Now, let me press another key than Y or N, and nothing happens here because the while loop will continue reading until either Y or N is pressed. And when I press the Y key it prints yes. So, basically what this does is it reads a single character from the input every time until it's either a Y or an N. Now, the last thing I want to show you is that we can read multiple words. So, I can do this, and this will read two variables, one called A and one called B. And, for example, if I say hi there, now $a will contain hi, and $b will contain there. So, read will split up the input into words and put each word in a separate variable. Now, if the string contains more words than there are variables, all the extra words will end up in the last variable. Let me show you again. Hi there reindert, and now the extra word reindert will end up in B as well as you can see. Now, how does read know where one word ends and the next word begins? Well, there's actually a variable for that. It's called IFS or Input Field Separator, and let's set it to something else because normally it's set to whitespace only. Let's set it to a column. And as you know, if I use a space after an assignment I can start a new command, and this assignment will only be true for the duration of the command I'm going to type now. So, now I can say read A B, and I can input stuff separated by a column like this. And now you will see $a contains 1 and $b contains 2 because now they are split by a column and not by whitespace. Also, let me show you that IFS will not contain the column after this command anymore. It just contains whitespace because in this syntax the assignment is only valid for the duration of this single command.

Standard Streams and Redirection

Whenever a shell script starts, it automatically connects with three standard streams. These are standard input, standard output, and standard error. When running your scripts from an interactive command line session, these streams will be collected with a terminal, and that means

that any input will be interactive user input from the terminal that's sent to standard in, and all the output, both normal messages that are sent to standard out and other messages that are sent to standard error, are printed to the terminal and presented to the user. Now, in UNIX programs and shell scripts, these three standard streams are identified either by a number or by a special file. First of all, a standard input has file descriptor 0. There's also a special file called dev standard in, which represents the stream. Now, standard out has file descriptor 1 and special file dev standard out. And this is a stream I haven't discussed before. It's called standard error, has file descriptor 2, and there's a file called dev standard error that represents it. UNIX programs should write any diagnostic or error messages to this stream, and in normal command line usage this stream is collected to the terminal just like the standard output. So, these messages just get printed to the screen, and you won't ever notice that they come from another stream than normal output, but because it's another stream, this makes it possible for us to treat errors differently than normal output, and I'll show you an example in a moment. Also, there's another special file that you should know about, and it's called /dev/null. Any data you send there is simply discarded, and writing this file always succeeds. This can be very nice if you need to hide a command's output. You simply redirect it to /dev/null and the output disappears. /dev/null is also called the bit bucket since it's like a trash can for bits. So, now we know about these streams, and let's see how we can manipulate them. Bash offers syntax for redirecting the streams so that their data flows to or from another place. In other words, we can take our input from a file or another command and send our output or error messages to a file or another command as well. If you followed the Bash intro course, this slide will pretty much be a repetition of stuff you already know, but the next slide will go beyond. Basically I'm assuming that everything in this slide is already familiar for you. First of all, there's input redirection. Using the less than sign I can have a program take its input from some file like this. Here the grep command will read from standard input, which in this case is simply the file shoppingnotes. txt. Now, you can also redirect your output using the greater than sign. For example, this will save a list of files in the file listing. txt. Remember that this will override existing files. By the way, using the built-in set commands you can customize this, but you also have the option to use two greater than signs, which will append your output to the end of a file keeping any data that was already in there intact. Finally, there's another kind of redirection called a pipe. This example will collect the standard output from ls to the standard input of grep allowing the chaining of commands. Very well. All of that should be kind of familiar. Now, let's see what other kinds of redirection there are. First of all, if you prefix the output redirection with a number, you can specify the specific stream you want to redirect. This is mostly used to redirect errors to a file. For example, this will execute a command and redirects the stream with the script of 2 to /dev/null. Stream 2 is the error stream so all error messages from

the command will be discarded. The number defaults to 1. So, if you are redirecting standard output, you don't need to type a 1 before the greater than because it's the default. You can also redirect the stream into another stream. To do this, you put an ampersand and a number after the greater than. For example, this redirects standard out into the standard error stream. Again, because standard out is the default, this is equivalent to the case where you put a 1 in front of the greater than sign. Similarly, this will send all errors to the standard output. Now, in case you're wondering what the use of this would be, it will allow you to log both errors and normal output to a single file. So, this will take the standard output from a command and save it in a log file, but it also sends all errors into standard output. And since standard output points to the log file, the errors now also end up in that same file. Now, you might expect that just redirecting both streams separately to the same file will work as well, but you would be wrong. The two streams would override each others data in the file, and that's not what you want. Instead, use the other syntax I already showed you. Furthermore, Bash offers two additional ways to point both streams to the same file. It can be done with &> and >& as well. However, both of these are considered deprecated, and you should really stick to the first option I showed you. However, you may run across these in scripts written by others, so it's good to know about them. Now, for a final note, you can use redirection anywhere on the command line, but order does matter. So, for example, these two commands are exactly the same. They will redirect standard output to the output file, and they will take standard input from the input file. But these two commands are different because the first command first sends standard output to the log file and then duplicates standard error to standard output, which means that errors will go to the log file, and the second command does it the other way around. So, first it duplicates a standard error into standard output, and at that point standard output is still pointing to the terminal so errors will end up on the terminal, and then it redirects standard output to a log file. So, in this case only standard output will end up in a log file, and errors will still end up on the terminal. Now, Bash offers even more kinds of redirection, but those are more advanced, and I don't think you will need them any time soon.

Demo: Redirection

So, let's start with the very basics. You should already know what happens when I do this. The output from the ls command will be sent to a file called demo. Now the file demo contains a list of my files. Now, let's go beyond that and redirect some output from our own scripts. Let's send the output to the demo file as well. Now, to start with we see that at least some of our output still ends up on the screen, and let's look at our script. Apparently it's this here, the prompt from read

that still ends up on our terminal, and we'll find out why that is in a moment. But then when I actually type a note nothing is printed to the terminal anymore. And then when I look at the demo file, we see that the normal output from the script now ends up in the file, and we're talking about this line here. This line simply prints output to the standard output stream for the script, and the script itself doesn't know that's actually connected to the demo file right now, so that's transparent to the script. It just happens here on the command line, but the script still sees just a normal standard output stream. Also notice that we already do some redirection here, and we redirect to a filename given on the command line. So, the fact that we redirect this standard output stream here doesn't really have any effects on a line like this because it doesn't use the standard output stream. Or actually, the standard output stream for this command now ends up here, so this doesn't really use the standard output stream from a script. Remember that because we didn't give any argument this will simply be notes. txt. Okay, so far so good. The next thing I want to show you is what happens when I redirect standard input. I prepared a little note file, which contains a single note. This note came from a file it says, and let's have our scripts take that as input. And, as you can see, apparently read does detect that standard input is now a file and not an interactive terminal so it doesn't print our prompts, but it does read the file because apparently the note is saved. And if we check our notes file now, you can see that's actually saved. So, again, what happens here, the read command uses a standard in and doesn't really know it comes from a file. It just sees a stream. Well actually, apparently it can detect that it's reading from a file and not from an interactive session. But apart from that, here to write the commands you really don't have to know whether the standard input is going to be redirected or not. Now furthermore, this here is an error message. It says you didn't have any input, and in my opinion that's an error so I'm going to send this to the standard error stream. Again, this echo command here prints stuff to its standard output stream and we redirect that standard output stream to a special file representing standard error so now this should be printed to the error stream. Let's test that. First of all, let's simply test our script with no input, and you can see we still see our error message here because on the terminal we can't see which data comes from standard error and which data comes from standard out. But what if I try to redirect again like we did before? Let's do this again and give no input at all. And now you still see the note here, and that's because we're only redirecting standard output to our demo file so the errors still end up here. Now, in case we redirect errors like this saying 2 for the error stream, now if don't give any input nothing is shown. And if we look at the demo file now, you see that our error ends up here, and actually you can see that the prompt from read also ends up here, so apparently read prints its prompts to standard error. Now, in the same case here if I do enter a note we still see our message here because this is printed to standard out, and we're not redirecting that. Now finally

let's try redirecting both. Let's start with just redirecting output to demo and then saying I want to redirect the error stream as well into the standard output stream, and now I shouldn't see any output at all. As you can see, standard output ends up in demo, and when I generate an error by not inputting a note at all, that ends up in demo as well. Now, let's just change this syntax for redirecting to the standard output stream because I'd prefer here to say I'm redirecting this standard output into standard error. Basically it's exactly the same, but this is more like a standard syntax. But if you prefer redirecting to dev standard error like before, that's okay as well. So, let's look at our other script called create_script again, and remember there was something wrong here because the type command messes up our output. So, let's take a look. Suppose I try to create something called new. Now, this is a new file, and it should just succeed, and it does, but you can see there's an error here from the type command that ends up in our output. So, to prevent this from happening what I can do is redirect errors from the type command to /dev/null. And if you remember, this will simply discard these kinds of messages. So, let's save and try again. Of course I have to create something with another name now, so let's call it new2, and now it works without giving it an error message, but there's another problem as well because suppose I try to create something called cp. This will clash with an existing command. We still see a message here, and that's because this is a normal message for the type command because it finds a command and it exits successfully and it just prints this to the standard output so we have to redirect standard out as well. And, again, let's do it as we did before, just redirect standard output to /dev/null, and then redirect the error stream to the standard output stream. And trying again with cp. Okay, very well. We see no output from type here anymore, and let's try again with a new file, and that works as well.

Summary

So, this brings us to the end of this course, and let's do a little summary. We've seen a lot of things about input and output. First of all, I showed you some new ways to use the read command. We've also looked at echo and some extra options we didn't know about yet and the new printf command, which lets you do a lot of more things to make nice output. I also told you about the three standard streams that are open by default in every script, the standard input, the standard output, and the standard error stream. Then I talked a bit about redirection. You can redirect the input stream with the less than sign, and the greater than is for output redirection. When you use two of them, you can append to a file. Also, by specifying the number of the stream you want to redirect, you can redirect a specific stream. This example will let you redirect only errors. And you

can also redirect into another stream. So, this example will let you redirect the error stream into the standard output stream. So now let's go onto the next module about flow control.

Control Flow

Introduction

[Autogenerated] Hi, This is Rachel Lacquer and welcome to this module called Control Flow. At the start of this course, I said that Boesch executes the lines in your script in order from top to bottom. But you already know by now that this is not actually correct with an if statement, for example, only part off the lines in script will be executed. We go the order in which the lines for colder executed control flow and in this module I will show you several ways to manipulate this flow. To start with will see several kinds of loops, allowing us to repeat a block off coat. First, we'll look at while and until which test for condition every repetition. We'll also see the four loop, which repeats for a fixed number of times. Then we'll see how to prematurely end loop with break and had to skip a cycle. With continue another structure. You'll be learning about this cold case, which lets you choose among several alternatives. Then we'll have a short look at how to group multiple commands into a block called a compound. Come out and we'll see another use for these two operators we know as and an or

While and Until

The while loop is really quite simple. It comes down to this. You start with the keyword while followed by a test. This is the same kind of test as we have seen with an if statement. Basically it's a piece of code, and we look at its return value. Don't forget to include a semicolon to denote the end of the test code. Then you need the do keyword and then the block of code you want to repeat. As with the if statement, this can be any list of commands including other if statements and while blocks, etc. Finally, you end this block with the done keyword. Now, the effect of this code would be to repeat the code block as long as the test returns true. Another very similar kind of loop is the until loop, which works just like the while loop, but it treats the test code in the opposite way. It repeats the code in the block as long as your test returns false. In the previous module, we already saw an example of how to use a while loop to read a single key, and in a similar way we can use a while loop to read a file. Let me show you this little script. It's basically an implementation of the get utility. First of all, note the input redirection here. Since the whole

while statement, which is this block, counts as a single command, we can redirect input into that. We take the first script argument as an input file. So, this read statement here will read a single line from its standard input, which has been redirected from the input file. Then this printf statement here will print that to the output, and then the loop repeats this. Now, at some point the end of the file is reached, and read here cannot read anymore input so it will return a num 0 value, and that is when we reach the end of the while loop because the test is no longer true. Now, let's see a little example with an until loop. This is a little guessing game. The user has to guess a number, and the computer gives you hints. The first line here takes a random number below 100. The random variable, this one, generates random integers, and we do a modulo operation here to make sure a number below 100 comes out. Now, the syntax here with the double parentheses and the dollar sign lets us do arithmetic, and we'll see more about that later so don't worry about it for now. Now, then we set the guess variable to an empty value just in case it was given a value outside of the script. Then we have the loop here. You can see the until keyword. And, as you can see, the game will loop until the guess equals the target of the random number we've taken. Inside the loop we let the user input a guess. This if statement here will print a hint. It says Higher or Lower depending on whether the guess from the user is higher or lower than the target. When the guess is correct, we print "You found it! " Now, you can find both of these scripts in the demo files in case you have access to those so you can try these scripts for yourself. And it's actually trivial to convert a loop like this into a while loop, and I will leave that to you as an exercise.

The Classic For Statement

Another loop structure is the classic for loop. Now, there's also another kind of for loop, and we'll see that next. The classic for loop begins with the for keyword followed by the name of a variable. It takes the name of a variable, not its value, so you shouldn't use a dollar sign here. Then it takes the in keyword with a list of words. It will take each word in the list and assign it to the variable var. And each time after setting a new value to var, it will run the code between do and done, and it will stop when there are no words left. Now, because you want the word list to be split, do not quote it. This is the one case where you don't want to quote the use of a variable. Of course you don't need to quote a variable in a conditional expression either, but in that case nothing bad happens if you quote it anyway. But in case of this for loop, things won't work if you use quotes around your variable because it will expand to one long word with spaces in it. Now, let's see how we can use this loop. First of all, you can use the for loop to loop over words in a string. So, for example, this will print each word in my word list on a separate line like this. What happens here is

as follows: Bash takes the first word in my list which is just and puts it in the variable I here and runs the loop, and the loop only contains this command so we print the value of $i, which is just. Then it takes the second word which is A, and it does the same thing, and we print A, etc., etc. Like I said, don't use quotes because when you do this now the for loop just sees one single long word like this, and the same is true when you loop over words in a variable. So, let's say this variable now contains a list of words. (Typing) Now, when I quote the variable like this, again we see it doesn't work because we have quotes around the variable here and so the spaces and its value will be escaped. So, to use the different words inside the variable, we shouldn't use quotes, and now we see this really works. Now, this for loop really shines when you use it with a wildcard expansion. I wrote a nice script to show you, and this script will let me change the extension on a whole set of files at once. It takes the old extension as the first argument and changes it into the new extension. First of course I check that we actually have two parameters, and here's my for loop. There's a wildcard here, and it will match all the files names ending in the extension given as the first argument. I quote $1 here because of course things will go wrong if it contains spaces and I don't use quotes. So, again, the for loop will take all the words in this list of filenames, and that means that the variable F here will contain each filename in turn. Inside the loop we first take the part of the filename without the extension, and I'm using the base name utility for that, which will strip the extension of the filename. So, now the variable base here contains my filename without the extension. Then I move the filename with the old extension to a file with the new extension. Again, I make sure to use quotes to prevent surprises. Also note that I'm not actually doing anything. I prefixed the command with an echo, which is a very nice debugging technique. I can see what my script will do without actually changing anything on the file system. When I'm sure everything works, we can remove the echo. So, let's try this script. First I'll create some files. This will create a number of files with the. txt extension as you can see. Now, let's change that to the extension. zip. This is my scripts, and I'm telling it to take all files called. txt and change their extension to. zip. And apparently we have a syntax error on line 10, and that's strange because let's see I'm pretty sure line 10 is fine, and actually my error is on line 9 because I forgot the do keyword. So, let's try again, and you can see it prints the commands we want it to. So, this wildcard here expands to all my file names with the. txt extension, and these get assigned to F one by one. For each one of them, we execute this code. So, now we can safely remove the echo command and turn this into a working script like so. And again let's try, and you see now all my files have the. zip extension.

The C-Style For Statement

This is the second kind of for loop called the C-Style for loop, and it basically lets you count the number of times the loop will run. It's very much like the standard for loop in languages like C or C# or Java so I'm assuming you're familiar with it and know how to use this kind of loop. It's very powerful, and it will let you write many, many different kinds of loops. I'm not going to explain in detail all the possible ways to use this loop because basically I think that's something for a general programming course, and I think you probably already know how it works. And anyway my Bash scripts at least mainly focus on manipulating files so I don't really use the C-Style for loop a lot. Now, instead of the in keyword with a list of words, this uses three expressions in double parentheses. The first expression is only run before the loop starts, and you can use it to initialize your loop variables. Then there's a test expression. This will be evaluated every time the loop runs, and the loop will only run as long as this evaluates true. And then thirdly there's an update expression, which lets you update the values of your variables. And, again, of course between do and done there's the actual code that gets executed every single time the loop runs. Now, in a moment I'm going to show you a demo, and you'll see that the syntax for the expressions is actually a bit different than you're used to. And actually what we use is the syntax for something called arithmetic evaluation. And I'm not really going to explain that to you right now because I'm going to take some time to explain that to you later. Now, I'm just going to show you one simple example of how to use this loop. So, here's a little script that will print a line, and it can be used, for example, for pretty printing output. It will take the length of the line to draw and the actual character to make the line from. Of course again first we start checking the arguments. Here we say that we need a length argument as the first argument, and we also have to check that the first argument is a number. We do that here. Now, actually we use something called pattern matching here in the test condition, and we'll see later how that actually works. For now you should just believe me we're checking that this is actually a number. Also notice that I'm printing the errors to standard error. I didn't do that on the other scripts, but as you recall that's actually the right way to do this. Also note that the char variable here is set to the equals sign, and only if the second argument is given on the command line we set it to something different. Then there's the for loop here. Again, remember it has three expressions inside the parentheses here. The first one is the initial expression that's only run once before we start the loop, and we make a new variable called I, and we set it to 0. Then every update of the loop we increment I with 1. So, it will first be incremented to 1, then to 2, then to 3, etc., and this will run as long as I is smaller than the length given on the command line. Now, every time the loop runs what we do is we take the current value of line and add one character from the char variable to the line and then put the new string into the line variable. Also note that I'm initializing line to be an empty value before we start the loop. So, this loop will run for the length times, and at the end line will

contain length characters. And then we print it. So, let's see how this works. Let's say drawline. This asks the script to draw a line of length 5, and as you can that's exactly what it does. Again, length here will contain the value 5, so I will go up to 4 incrementing each time the loop runs. And as soon as I reaches the value 5, we end the loop. So, because we've started with 0, from 0-4 is exactly five times of looping. Now, there's another way we can call this loop of course, and that's with a different character like this, and there's also a little bug. Let's say I want to print a really pretty line like this, and we get some strange errors. Printf says invalid option here. And this is because we are using the input in the format string here. Since the line now starts with a minus like you can see here, printf now thinks we are trying to give it an option here. So, you should know never to use a variable like this in the first argument of printf. You should do this instead. And now printf will see a normal format string containing just the percent code here, and it won't think the value of line here is an option because line is given as the second argument. So now if we try this, we actually get a really pretty line.

Break and Continue

For all kinds of loops there are two keywords that will change the program flow through the loop. First of all, the break keyword lets you break out of the loop at any time before the loop would normally end. The continue statement will skip the rest of the current iteration through the loop and continue with the next iteration. This is very similar to the same statements in Java or C#. Of course the test will still be done to see if the next iteration is actually necessary, and in case of a for loop it will still update your variables for the next iteration. So, both keywords can be used in any for, while, or until loop. Now, to show you an example of the break and continue statements, let's consider a little script that will strip the beginning of a file until it encounters a certain string we're looking for. The first argument will be the string we're looking for, and of course we have to check that it's actually there. We do that here. And then I'm going to read the file up to our first matching line and discard all the lines up to there. So, I have a while loop here again that reads the file. It simply reads from standard input. And every line it checks if the line contains our argument. Note that I'm using this operator, which I've also used before to check if a variable holds a number. And, again, that does pattern matching. Now, I'm going to explain pattern matching more in detail later, but for now you should know that this will check whether the first string here contains the second string. If it does so, we print this current line, which contains the string we're looking for, and then we break out of this first loop. What that means is we simply end up here after the loop and will execute the rest of the script file. And then we have a second read loop, which simply continues reading from standard input and prints every single line. Now,

to give you an example of how this works, I made a simple little web page, and what I want to do is I want to remove the whole head here and just see the body. And to do that I'm going to call my script to tell it I want it to strip every line until I see something containing the string body. And because my script is reading from standard input, I have to use input redirection here. And that prints my body and discards the head. Note there's also a little but here because it doesn't print the last line of the script containing the closing HTML tag, and we'll see how to fix that in a moment when we know about compound commands. So, let's go back to the script, and here we have the break command, and I'm going to show you how to rewrite this with a single loop and a continue, and that's works like this. Now, I'm using a separate variable here that doesn't contain a value initially, but that gets a value as soon as we find our string. Again, we have a while loop that reads lines, and as long as our string isn't found here we execute this if statement here. So, we're still at the start of our file. We haven't found our string yet. And then if our line contains the string we're looking for, then we're setting found to a different value. But if it doesn't, we want to discard this line so we end up here, and what I do is continue. And what continue does is it simply skips the rest of the loop including the printing line here so we don't get any output, and it just goes here, and we read the next line. So, as long as we haven't found our string and the current line doesn't contain our string yet, we're calling continue every single time and discarding all these lines. Then once we find our string, we have found a value, and we start actually printing. So, let's test this as well. It's called stripto2, and it works as well. Again, there's the bug that it doesn't print the last line, and we'll see how to fix that very soon.

The Case Statement

The case statement lets you execute a piece of code depending on the value of a variable. In that way it's kind of similar to the if statement when you use it with multiple elif branches. The case statement looks like this. It starts with the keyword case and ends with its reverse, esac. After the case keyword comes a WORD, which is the value we are going to test to decide what code we will be executing. After this word you use the in keyword. Then between in and the end of the statement you can use any number of patterns to match against a value of our WORD. Now, these patterns work exactly the same as when you're using wildcards to match filenames. So, you can use a star to match any string or a question mark for any single character and square brackets to match a specific set of characters. Note that you end the pattern with a right paren. After that comes the code you want to execute when the word matches the pattern. Bash will test a word against every pattern from top to bottom and the code for the first matching pattern is executed. Of course this code can consist of multiple statements and things like if statements or

loops. So, to make sure we know where the block ends and the next pattern begins, you have to use a special syntax, namely two semicolons. Actually, there's also a different construct here you can use to enable fall through and execute the next pattern as well, but that's considered bad practice by a lot of programmers. Now, a nice feature is that you can also use multiple patterns at the same time for a single code block by using a pipe symbol to separate patterns. So, let me just show you how this actually works in a demo. You probably remember the animal script from the module about if, then, else, and it turns out we can re-implement that same script pretty simply using a case statement. And if we do that, it looks like this. Basically, here we have the case statement, here is the first argument to our script, and then we simply match that against several different patterns, and the patterns here of course are cat, dog, and cow. And then if none of those match, we have to use a separate pattern that will match everything else. For that, we simply use a single star, and as you know the star will match any string, so that's basically the same as using else after a couple of elif statements in an if, then, else. So, in case none of the other patterns match, we print unknown animal. So, that's a very basic case statement. I also made a slightly more complicated script called mytar. It's basically a wrapper around the tar command. Now, tar is a standard utility on UNIX for archiving files, and sometimes it can be quite hard to remember the correct invocation for what you want to do with tar. So, this is a simple script that tries to compress or uncompress a file for you without having to remember exactly how to do that with tar. It has two uses. If you call it like this with a directory and a file, it will compress all the files in the directory into the given file or you can call it like this, and it will try to extract the compressed file that's given as the argument. Now of course I do a lot of checking here to see what's in the arguments and which of the two invocations I've actually done. I'm not going to really explain them in detail here because I want to focus on the case statement here, and that's this statement. As you can see here, here's the actual filename we have to use, and what I'm trying to do is look at the extension of the file and see what kind of compression we have to use because based on the compression we are using, we have to give different arguments to tar. So, for example, let's look at this first pattern here. You can see that there are actually three patterns in this line separated by these pipe symbols. So, this here will match any filename that either ends on. tgz or. gz or. gzip. All these are different conventions for naming gzipped tar files. And if we want to compress or uncompress a gzipped tar file we have to give tar the Z option. So, in this case what we do is we remember that we have to use the Z option, and then we print a nice little diagnostic message to the standard error. Remember diagnostic messages should be printed to standard error and not to the standard output. We do the same here with four different conventions for naming bzipped files, and then we use the J option. Then we have compressed files, and then we have just tar files that don't have any compression at all. And for any other

possible filename, again we match tar that will match anything, and in that case we say it's an unknown extension, and we exit the script. So basically after running this case we know what kind of compression is used, and we saved the option that's used for that in the variable called zip. And then here we actually create our tar command. Operation here will be a C for creating files or an X for extracting files. This will be our zip option, so that will be either a Z or a J or an uppercase Z or nothing at all if the file isn't compressed. And then there's the F option, and that's needed so we can give tar the actual location of our file. Now in case we have a directory as well, we put the directory after that command, and then we try to run the tar command we just made. And if it fails, we print that. And if it succeeds, we print Ok. So, to give a little demo, let's say I want to compress all these zip files here. First, let's move them into a directory. Very well. And now I can compress this whole zips directory with the mytar script into something. Let's call it zips. tgz. And using the case command now, my filename should match this pattern here, and that means we're going to use gzip. Let's see if that works. Very well. Let's check, and file tells me my zips. tgz is a gzipped compressed file. I can also extract the files from this archive. Let's just remove my zips directory and use the mytar script to extract it. And, again, now the zips directory is back. And if we look what's in there, there are my files. So, that should give you an impression of how to use the case command.

&& and ||

Many of the structures we have seen until now take groups of commands and execute them like a block. This happens in the if statement and in loops for example, but you can also group commands into a single statement outside of these structures, and to do this you group multiple commands together by surrounding them with braces. This will group the block into a single statement as far as the shell script syntax is concerned. Practically speaking, this means you can use redirection for the group as a whole. For example, letting multiple statements read from the same file or for example you can use a list of multiple statements in an if statement or a while loop to test its status. Note that the return status of the whole statement is equal to the return status from the last command in the list. So, a group consisting of three commands looks like this. Now, it's important to know that you should put a semicolon or a new line between the statements in the group. Also, you need to put a space after the first brace and another space before the ending brace. Or in other words, the braces need to be words that are separated from the commands by whitespace. Finally, don't forget the semicolon or new line after the last statement. Now, there are two other things you can put between statements besides semicolons and new lines, and those I'm informally going to call and and or. You've already seen them in the

module about the if statement, but we can use them in another way as well. They will let us execute a command depending on the return status of the previous commands. Simply speaking, this gives us a very nice shorthand for a simple if statement. So, to start, the double ampersand is similar to a logical and operator. It looks at the result of the command preceding it, and when that succeeded, it will execute the statement after it. So, for example, this will try to make a new directory and cd into that, but the cd will only be called if mkdir reported success. The alternative is the double pipe symbol, which is similar to a logical or. It executes the command after it only if the command before it fails. So, for example, this will test whether the first script argument holds a value, and if not, it will print an error. Now, you can combine these statements like this. And here we see the same test and error message, but at the end is the double ampersand that exits the script with a known 0 status. Unfortunately this line will always exit your script. If $1 does hold a value, the test returns true, and no message is printed. But the successful return of the test is passed to the double ampersand at the end, and exit is called. On the other hand, if $1 is empty, the echo command is called, and that one will just about always be successful, and then that successful return value is passed to the ampersands, and again we exit. So, instead of combining the statements in this way, you can use grouping like we saw on the previous sheet. So, if you do this after it turns out that $1 is empty, we run two statements grouped by braces. First we print a message. Then we exit the script. And now if $1 does hold a value, exit isn't called, and the rest of our script is executed. Now, let's clean up some things in the scripts we've seen in this module. First of all, this is the script I made to change extensions on filenames, and there are a couple of things I don't like about this. First of all, I forgot to print this to standard error. Very well. And I also forgot to exit the script after this error, so let's do that as well, and then we now know that there's a nice shorthand for the if statement here. And basically when you read other peoples shell scripts they will not use the if statement to check variables like this. They will normally simply use the double ampersands or the double pipes. So, let's replace our if with to ampersands, and you may remember that we need a command group here to make sure that the exit gets executed as well, but only if the test at the start of the line succeeds. So, basically now we've put the entire if statement into a single line. Now, another problem we had was that the stripto script here didn't print the last line of my file. You may remember this. Let's run it again. It doesn't print the final closing tag for the HTML file. And why is that? Well actually, here the read statement will return false whenever it finds the end of the file or line. So, even though it does read the final line, which is this line, there's the end of the file here and so it will return false. Now, one way to fix this is to use a new line at the end of my file, and then it will read this line and print it and then find the end of the file. But of course you can't count on all your input files always having a new line at the end of the file. So, instead of doing this, we should do something else. What I'm going to do is

that I'm going to use two pipe symbols here and then check if the REPLY variable holds a value. So, in this case whenever read returns successfully, that means it didn't see the end of the file yet, it will report success, and we don't have to do this test. But when we do find the end of the file, this returns false, sorry, this returns false, and then we'll have to test whether there's something in REPLY because even though read found the end of the file, it will still put the last line in the REPLY variable, and in that case we print the last line here. Now, of course, don't forget to do this here as well because otherwise we would have a bug if we never find the string we're looking for. And let's try again. And, as you can see, now the last line is printed. Of course we have to apply the same changes to the stripto2 script, which uses continue instead of break, and to the mycat script, which I showed you at the start of the module. And you can see I actually did the change already, so there it is. So, one of the reasons I'm showing you this is to show you that in a test like for the while statement or an if statement you can actually use a list of commands like this as a test.

Summary

So, in this module we've seen several kinds of loops, namely while/until and two kinds of for loops, how to use break and continue to alter the flow of control in your loop, the case statement, you've seen how to use compound commands where you group commands with braces, and how to use and and or to look at the previous statement, see what it returns, and then see whether you're going to execute the next statement as well.

Variables 2

Introduction

[Autogenerated] Hi, I am arrange a lecher and welcome to this module, which will show you some advanced features of best variables. Off course. We've already done a module about variables, but in this module I want to expand on that and show you some extra features you can use with variables. First of all, variables in best have a tribute, which you can set with the Declare command. For example, you can tell Boesch that's a very well should only hold interviewers. When you do that, you can use special expressions called arithmetic expressions to do calculations you can also mark. Variable is being read only, which means you won't be able to change its value after it's set. And I want to tell you some more about what it means to export a

variable and how you can do this. Finally, I'll tell you how to store multiple values in a single variable with a special kind of very vocal dhe, an array

Integer Variables

Now, as you know, Bash variables normally hold simple string values without a type, but it is possible to give a variable some extra attributes to enable Bash to do a few extra checks on what happens with your variables. You can set and unset these attributes with the declare command. You can also call declare by another name, which is typeset, but that one is really there for compatibility, and it's considered deprecated. Now, before I go into any actual variable attributes, you should know that you can print the attribute set to a variable with the -p option. One attribute you can give to a variable tells Bash that it should only hold an integer value. You set it with declare -i followed by the name of the variable. After using this, your variable will only hold integers, but unfortunately when you try to set it to a string value, Bash won't give you an error. Instead, your variable will be set to 0, and that's something to be aware of. By the way, you can always unset an attribute using a plus instead of a minus, so this will remove the integer attribute from the variable num. When the integer attribute is set on a variable, when you give it a value, you can use something called arithmetic evaluation, which lets you do calculations in a friendly way. Now, let me first show you how to use an integer variable, and then I'll go a bit deeper into arithmetic evaluation. So, just a really short demo of how to use an integer variable. First of all, let's say I have a variable P, and I want to do a calculation. I might try this. But of course 4+5 here is just a string. So, if I look at what's in P, we see it holds the string 4+5. But what happens if I declare P to hold integers? Now, let's try again. And now we see because P is declared as an integer variable this here triggers a special evaluation, and now it's not seen as a string, but as an integer expression. And the calculation is done, and P now holds 9. Now, of course you have to use quotes in case you want to use spaces, so of course I can do something like this. Let's say 2 * 5. But don't forget the quotes because if I don't use quotes I get an error. Also, in case I set P to a string, that evaluates to 0 because P is an integer variable, and it can't hold strings.

Arithmetic Expressions

You can use arithmetic expressions to do calculations with a C-like syntax. You can tell Bash you want your expression to be evaluated in this way in several ways. First of all, there's the let command. For example, this will set N to the value 50. You can do similar things by surrounding your expression with double parentheses. For example, this will increase the value of X by 1, and

this will set P to x/100. You can also see here that we can safely uses spaces in this expression. We can also use command substitution like this. This will call ls, count the lines in its output with the wc command, then multiply that output by 10, and set the result in the variable called P. By the way, the let command and this expression with double parentheses are exactly equivalent, but because the command with the parentheses supports spaces among other things, I prefer it above the let command. This is a similar construct where we prefix the parentheses with a dollar sign. Now, the difference is that this is not a command, but a substitution. So, if we use it like this, Bash will first divide X by 100, and then replace the whole expression with that value, and the result will read p=50, which of course sets P to 50. Bash will also assume you are using an arithmetic expression whenever you set a value to a variable declared as an integer. This means this will set num to 30 modulo 8, which of course equals 6. So, let me show you a little example to make the difference between a command and a substitution more clear. Let's start with declaring an integer variable. Let's call it X, and let's set X to some value. Let's say 100/2. I don't need quotes right now because I'm not using spaces. So, let's see what's in X. It's the value 50. Now, suppose I want to increase X by 1. I might try to do this. And here ++ of course is the operator from C that will increase the value of my variable with 1. And this is a command substitution, so let me show you what happens now. X does get increased by 1, but then the expression returns 51, and the whole expression, because it's a substitution, the whole expression is substituted by that value so then the command line reads 51. And so Bash tells us 51: command not found. So, a substitution by itself is not a command. It's simply an expression that gets replaced by some value whereas when I use this, which is simply a command, Bash doesn't print anything. But now if we look at the value of X, we see that it's now 52. Of course you can also do this. And here I don't want to use a command. I want to use a substitution. And now this expression will evaluate to x+1, and then we put that in X as the new value.

Arithmetic Expressions 2

Just some more remarks about arithmetic evaluation. First of all, you already noticed that there's no need to quote variables inside arithmetic expressions, although you can if you want to. Furthermore, you can use an expression in double parentheses as the command in constructs like if and while, but be aware that in that case 0 is false and anything else is true so this is the opposite of the truth value of the return count of a normal shell command. So, for example, this will print false because we're using the double pipes here, and the expression returns 0, which is a false value. Also, there's an important pitfall you should know about, and that is that Bash will see any numbers with leading 0's as being octal numbers, not decimal. So, giving a number the value

010 will result in it having the decimal value 8, and this can give you some pretty strange surprises. Also, we've seen the double parens before in a C-style for loop. But formally speaking, in that case this syntax is not an arithmetic expression. It's simply part of the syntax for the for loop, but the three expressions inside, which are separated by a semicolon, are actually evaluated as an arithmetic expression, and in there you can do any kind of calculation you want. Now, in case you want a full overview of operators you can use, go to this website or, of course, you can consult the Bash man page. Now, to show you how to use these expressions in a script, I changed the game script I showed you before to use arithmetic expressions wherever possible. So, let's start at the top here. I've also changed the user variables to use declare where necessary. So, I start here by declaring target to be an integer, as well as a read-only variable. This R here means target will not change, so we can only set the value of target once. Then this here is an arithmetic substitution. And as you can see, I use parens here. First I take a random number from the special random variable, then I use modulo 100 to make sure that this number here will always be below 100, and then what I do is I add 1 because I want the target to range from 1 up to 100 instead of from 0 up to 99. And why that is I'll show you in a moment. Then we declare the guess as well to be an integer variable. Then you can see here in the until command I'm now using an arithmetic expression as a command. This here is the equality operator, which will return true if these two variables are equal. Now then we start reading our guesses, and here I don't have to use pattern matching in this case to check whether guess is a number because whenever we try to put something that's not a number into guess, it will be 0 because it's been declared as an integer. So, all we have to do is check whether guess is 0. Now, remember that in this case 0 will be a false return value so I have to use double pipes here, and whenever guess is 0 we simply continue, so we go back up here and start reading the next guess. And in the same way now with an if, then, else, we can simply check whether guess is smaller than target or whether guess is larger than target. So, now, of course, we have to make sure that our target is larger than 0 because we're not accepting guesses that are 0, so that's why I added the + 1 here. Now, I hope this gives you a little bit of an impression of how you can use arithmetic expressions in a script.

Read-only Variables

Now sometimes you want a variable to have a constant value. That means you set it once, and you don't want it to change. In that case, you give it the read-only attribute with -r. Make sure to set it to the value you want it to have because after this the value of your variable cannot change. If you try, Bash will report an error.

Exporting Variables

Now, normally if you use a new variable in your script, it's local to that script. Or if you're on a terminal in an interactive session, it will be local to that session. Now sometimes you don't want that, and you want to make your variables available to other commands you are running, and to do that you have to export it. So, if you export a variable, you make it available to subprocesses. Now, unfortunately this doesn't work the other way around so you cannot pass a variable from a subprocess to the command that called it. Now, how do you export a variable? Well, first of all with the export command, and you can also use export and immediately give your variable a value, or with declare -x, which gives it the export attribute, which is exactly equivalent to calling export. And, again, you can immediately give it a value. You should also know that any attributes you give your variables, like that it should have integers or that it's a read-only variable, are not exported to other processes. So, I created two little scripts to show you how to export variables and what happens when you do that. One script is called outer, and the other one is called inner. Now of course the outer script calls the inner script, and that's why I named them like that. And let's look at the outer script first. It sets a variable called var to the value outer and then prints the value before calling inner and prints the value after calling inner as well. Then the inner script first prints the value var as this at the start of the scripts, then sets var to the value inner, and then prints var again. Now, let's see what happens when we run this. And what you see here is that at the start of the inner script var is not set to any value at all, and, of course, that's because we didn't export var. Also note that after setting it to a value of inner in the inner script that doesn't get passed to the outer script, and, of course, as you know, that's because you cannot pass a variable to a parent process. Now, let's just start with exporting the variable in the outer script like this, and now you see when we reach inner the value is set. So, what we print is here. There's a value set to var, and we print it, and then we set it to inner. And, again, it doesn't get passed to the outer script. Now, even if we export it here that doesn't matter at all because, as you can see, for the outer script it's still set to outer. Now, of course, although we export var in the outer script, that doesn't get passed to the parent process of the outer script, which is my shell here. So, now if I do echo $var now, it doesn't hold a value at all. Now, basically this is because every process has its own environment, and environment variables are not just a concept in Bash, but for the whole operating system. And, for example, another thing that's part of your environment is your current working directory. So, that means that if I change my current working directory in this script, let's change it to let's say the root dir, and we run outer now, the current working directory for this terminal session, which is the parent of the outer script, doesn't get changed. And, again, that's because changes in the environment for a child process don't get inherited to their parent

process. Now, another thing that doesn't get passed to the environment is variable attributes. So, for example, if I give this variable the read-only attribute, I cannot change its value. Let's say that I'm going to change it here. Bash gives an error. But of course we're not just changing it here. We're already changing it in the inner script. And, as you can see, in the inner scripts Bash doesn't give an error so apparently the read-only attribute doesn't get passed to a subprocess. So, even though you can pass variable values to subprocesses, you can't pass variable attributes. Now, let's look at another script we already saw, take note, and I did quite a lot of work on this script. I did a lot of stuff like declaring all the variables in a nice way, some of them read-only, and maybe more importantly, I added this reference to an uppercase NOTESDIR variable. And I made it uppercase to make it clear that it's meant to be an environment variable set by the user to customize the behavior of my script. What I'm doing here is I'm detecting whether NOTESDIR is set, and if it is, I'm copying its value into the lowercase NOTESDIR variable. And as you can see, by default I'm setting that to #{HOME}. Now, of course, the user can set this variable to anything he likes, so I have to check first whether it's actually an existing directory. And if it's not, I'm going to try to make one. And if I can't, I'm going to print an error and exit the script. Then I determine what filename I would be writing to, and then I have to check whether that's actually an existing file. And if not, I'm going to try to create one. And, again, if for example the directory isn't writable and I cannot create a file, I'm printing an error. When it's a preexisting file, it's still possible that the file itself is not writable, so I have to check that as well. So, that's quite a lot of logic to make sure everything will work. But of course this module is about the variables, and specifically right now we're looking at exporting so let's look at how we would set this variable. Well of course the correct way to do that would be in. bashrc or one of the other files you can use to customize Bash. And here you see I'm setting the NOTESDIR variable to ${HOME}/notes. Also note that I'm not using a tilde here of course because a tilde within quotes wouldn't be expanded. Now, in case I do this, I have to start a new Bash session because I want Bash to read my. bashrc. There it is. And now let's say take note test. My note would also be test. And you can see that it doesn't write to the NOTESDIR. It still writes to my home directory. And, of course, again, that is because I didn't export my NOTESDIR variable. Let me check that it's actually set. Yes it is set, so let's try to export it. And now it writes to my NOTESDIR very well. And, of course, I should put export here before the NOTESDIR. Now, you might wonder why this is possible because I just showed you that exporting like this from a script won't effect any changes in a parent process. Well, that's because Bash when it starts doesn't read in the. bashrc in a subprocess, but in its own process so all the variables it reads here will be set in this interactive process here. And so if I call export here, these variables will be available to all subprocesses of this interactive terminal session.

Arrays

Apart from strings and integers, Bash actually supports one more kind of special variable, and it's called an array. And an array can hold multiple values, and you store and retrieve them by index. Now, this is a concept that's present in just about any programming language, and I think you're probably already quite familiar with it. So, how do you store a value in an array? Well, like this. So, in this case the word some is stored in an array called X with the index 0. And I can store another value, word, in X under the index 1. But when you want to retrieve a value, there's a very different kind of syntax. It looks like this. First you use a dollar sign and then braces, and then you can use the name of the array with what we call the subscript, so the index between brackets. And in this case, this would retrieve the value some. And in the same way, this would retrieve the value word. Now, there's also this special syntax where you put either an at or a star inside the brackets, and that will retrieve all values in a string. And if you would quote this expression, that works exactly like $* and $@, which means that if you use a star and quote that will just put everything in a long string, and if you use $@ and quote, that will quote every single item by itself. Now, how do you tell Bash that you want a variable to be an array? Well, you can declare it -a or you can simply assign with an index like we already saw, and Bash will simply assume it's an array. Now, there's also a special syntax for initializing a whole array in one go. That looks like this. So, this will make an array ar and put six values in there. So, 1 will be at 0, 2 will be at index 1, 3 will be at index 2, A will be at index 3, etc., etc. But there's some special syntax. How do you for example ask an array how many items it holds? Well, this is what you do. And prefixing it with a hash is the same thing you would do with a normal variable that holds a string and that would give you the length of the string, so this should be familiar to you. It can also ask the array to list all the indexes it holds like this, or indices I should say. So, if you use an exclamation mark before the name of the array, it will give you a list of all the indices in there. Note that there can be gaps in the indices. So, you can have like an array with two elements, and one would be at index 100 and the other one at index 200. Now, since we can't export any of the Bash variable attributes, it shouldn't come as a surprise that you cannot export an array either. And there's something else you should know. Since Bash 4, Bash supports associative arrays where you don't store your data by number, but by name. And to do that, you should declare it with an uppercase A. Now, I actually don't think those associative arrays are that useful in a normal Bash script, so I'm not going to go very deeply into them, but I am going to show you a demo where I use them in the next module. Also, if you want more information about arrays, here's a nice website where you can read all about them. So, let me give you a short demo of how to use arrays. First, let's just create one with the syntax I showed you. When Bash sees that you're doing an assignment and it starts with a single paren, it

will assume you're making an array. Now, one way to list the array would be to use declare -p. And that shows you that it's an array, and it also shows you all the values with their indices. So, for example, if I want to retrieve the value with index 2 I do this, and of course I should say echo, and that gives us this value. Now, like I told you, an array can have gaps so there's no problem with let's say putting something at index 15. And let's look at the array again. Very well. Now, if I ask this array for its length, note that you should use an at here, and of course I should use echo. That tells us the length of the array is 5 elements, so it doesn't really care that there's a gap between 3 and 15 because it simply holds 5 elements right now. Now finally, let's ask for its indices like this, and it will tell you it holds values at 0, 1, 2, 3, and 15. Now, I'll show you a real demo where I use arrays a little bit more in depth in the next module.

Summary

So, let's wrap up this module. First of all, we saw about integer variables and that they can be declared with the -i switch. Once you start using integer variables, you'll probably also need arithmetic expressions, and we've seen that you can use a special command for that with double parentheses. And if you prefix that with a dollar sign, you get a substitution. We also saw that you can declare read-only variables with -r, how to export variables with declare -x or the export command, and finally, I showed you how to use arrays.

Handling Script Parameters

Introduction

[Autogenerated] Hi. Welcome to this module, which will teach you how to correctly handle the arguments given to your street in the real world. Shell scriptural. Not simply take one or two arguments. They tend to have any number of options, and some of these are required and others aren't. And there may even be options that take extra arguments for themselves. Now, after watching this module, you'll know several ways to handle these scenarios. During this module will see several special variables defined by Boesch. These are the positional parameters with with numbers one to et cetera, which we already know about. And the star and at Variables, which hold all arguments given to your scripts. The hash, which gives you the number of arguments. And the Lord zero, which looks like but actually isn't a positional parameter. To handle the

parameters given to your script, there are two commands we will be looking at, namely shift and get upped

Special Variables

So, let's take a look at the different kinds of special variables you will have to use when you're handling your script's arguments. First of all, there are the positional parameters, and they're called $1, $2, $3, etc., and they hold all the different arguments in order. So, $1 holds the first argument, $2 the second one, etc., etc. Now, of course, when you get above 9 it's wise to use braces when you have parameters called 10, 25, etc. because, otherwise, Bash will think it's $1 followed by a 0. Then there's another special variable called $0, which isn't actually a positional parameter. It holds the name of the script as it was called when someone called your script. And since UNIX systems have things like symbolic links, this means that the way your script is called can be by a different name than the actual filename of the script file that you made so sometimes it can be wise to inspect what's in $0 to see with which name your script was actually called. Now, in some cases you want, for example, to loop over the arguments in your script or maybe you want to call another script and just pass all your arguments to this other script, and for that purpose there's the $@ variable, and it simply contains all the arguments given to your scripts. So, it's basically equivalent to putting $1, $2, $3, etc. behind each other, and that would give you the same value as when you would be using $@, but there's some magic involved as well because when you use double quotes around it, $@ will give you each argument in turn again, but then all of them are quoted so that means that the spaces in your individual arguments are actually escaped so parameters containing multiple words stay intact when you use double quotes around $@. There's also another special variable, $*, and it's pretty much similar to $@ when you don't quote it, so in that case it's equivalent again to $1, $2, $3, etc. But when you use it double quoted, it simply returns the whole argument string, so all arguments behind each other, and it doesn't do the magic kind of quoting that $@ does. And basically in most normal usages you don't want that. You want to get the separate arguments from your command line. So, in normal usage you never want to use $*, but you always want to use $@ instead. Finally, there's $#, and you already should know this. This simply holds the number of arguments passed to the script. Let's take a short look at these things I showed you in a little demo. First of all, here's a little script called scriptname, and all it does is it prints the $0 variable. And as I told you, $0 holds the name of the script as it is called. Now, basically if I call it directly like this, we see that the script prints the full path name. That's because my script is in the bin directory, and Bash has to look it up on the path, and we'll call it with the full path name. Now, if I call it with a relative path name like this,

it will print that. If I use a different relative path name, again, that ends up in $0. Now, another thing I can do on a UNIX system is make a symbolic link. There's the ln -s command, which will make a link. Let's make a link to bin/scriptname, call it sn, and now if I call it through the symbolic link, the actual name of the symbolic link ends up in $0. Now, why is this useful? Well, sometimes you have a script that you can call through various differencing bulk links with different names, and you want the script to behave differently depending on how it's called. And actually, for example, Bash does that. Of course Bash is not a script, but it does the same kind of detection. I can call it as sh, but actually the sh shell is Bash on Mac OS, but it will detect that it's called this as an H, and as you can see, it behaves a little differently, and it will behave in a compatible mode. So, even though the chance is small that you'll actually do this, sometimes it's nice to know that there's the $s variable to see how your script was called. Also, when your script prints diagnostic messages, it might be nice to use $0 instead of the original script name to make sure that the user that reads the message knows which script is actually reporting the message. So, in my example here if this script would print an error, it would be nice if it called itself sn because that's the name by which I called it. So, if I were to print a diagnostic message, I would say "Running $0" like this, and now the script reports that I'm running sn, and that's nice for me because that's actually how I know this script. So, moving on from $0 to the $@ and the $* variables, this is a very simple script called printargs, and it simply has a for loop where it will print all the words in $@. So, let's try this script, and of course $@ will now hold the string first second third, and the for loop will break that up into words, and every word is printed by itself on a single line. So, that's pretty basic. And in this case $* does exactly the same thing. But sometimes I need to give an argument that has several words in it, and I might use quotes for that like this. Now, you should realize that what ends up in $* here is simply the string "first arg" second third without any quotes or special characters in there. There's just first space arg space second space third. That's what's in $*. So, when I run this, every single word ends up on a single line. And, of course, that's not what I want because I want these two to remain grouped. Now, simply using $@ will not solve that for you. Again, it does the same thing, but using $@ with quotes will make sure that the arguments remained grouped or, to put it more formally, will quote each argument by itself. So, now when I do this, we see that the whole first argument stays intact as a single word and will be printed on a line by itself. Now, using $* like this has a completely different effect. Remember $* is simply the whole string "first arg" second third with spaces in between them, so when I quote that, we'll get the quoted string "first arg" second third, which is a single word for the for loop, so now the whole string gets printed on a line by itself. Now, like I said before, most of the time, actually almost always, you simply want to use $@ in quotes.

Shift

One command you should know about that manipulates the arguments that your script received is called shift. It removes the first argument, and that means that all other positional parameters shift. So, what used to be the second argument to your script is now the first one. So, $1 holds now the value that previously was $2. $2 holds the value that used to hold $3 etc., etc. This also means that after executing shift your $#, the number of arguments that your script received, is lowered by one. So, let's look at a scenario where you might use this command. Now, remember that I wrote a script that will create other scripts, and sometimes I want to create multiple scripts at once, and that means I want to be able to give multiple filenames on the command line and create a script for each of the arguments. Of course what we can do for that is loop over the arguments. So, I changed the script a bit, and let me show you. First of all, I did some little changes like declaring my variables. And here I'm saying only when I just get one argument I'm going to open editor, and if I get multiple arguments and creating multiple scripts I'm not going to open the editor. So, but let's go down a bit. Here's my major change. And as you can see, we loop as long as there is a $1. So, as long as there's still an argument on the command line we loop, and every time at the end of the loop we call shift. So, gradually we're removing each argument from the command line as we are processing them. And then in here it's basically the same set of commands I was using previously. The only difference is that now when I find an error like the file already exists or there's a command already with that name, instead of exiting, I just call continue. Now, of course, because continue will skip here immediately, then I have make sure to call shift as well because otherwise $1 will not change, and we will be stuck in a loop. So, that's probably one of the most basic ways to loop over your arguments and handle multiple arguments for your script. Now, in case you want to do more advanced stuff like maybe have arguments that have options for themselves and stuff like that, this will become nasty really quickly because then you'll have to do some kind of parsing and check whether everything is present. And how are you going to handle options that are not allowed, stuff like that? So, although this while loop with shift works pretty well, if you want to get a little bit more sophisticated, well actually there's a better solution for that, and let's look at that next. But let me just give you a last remark on the shift command. We've just seen that calling shift by itself will put the command line argument in the $1 variable one- by-one, but it's also possible to give shift an argument. And if you give it an argument of a number, it will shift multiple arguments at once. So, shift 3 will remove the first three arguments, and we'll see a use for that soon.

Getopts

Like I said, when your script takes multiple arguments and some of these arguments have options for themselves and some are not allowed, etc., etc., parsing this gets nasty really quickly, and fortunately Bash has a very nice utility for that. It's called getopts. It processes options that start with a dash followed by a single letter like -x or -a, and it also allows options to take an argument. So, if you have a -f option that takes a filename as an argument, getopts will also help you with that, and you call it like this, getopts optstring name. And the optstring here is a list of expected options. So, for example, if you call getopts with an optstring of ab, getopts will accept the options -a and -b. Now, in case you have an option that takes an argument like the -f example above, you append a colon to that. So, in case your optstring is a:b, A will get an argument, but -b won't. Now, the name option to getopts is simply the name of a variable. And every time you call getopts, it will take the next option and put it into $name. Of course this means you will have to call getopts multiple times, so you will usually use getopts in a loop. And to facilitate that, getopts will return false when there are no more options left on the command line. So, let's look at a little example. So, here's a script that will simply count numbers from 0 up to any number you give on the command line and print that. And here you see the while loop where I'm going to process my options. As you can see, I take a B option with an argument, an S option with an argument, and an R option, and there are some comments here as well. As you can see, -b tells our script where to start counting. It defaults to 0. -r reverses the count, and -s sets a step size. There's also a separate argument that comes after all these options, and that separate argument is called stop. Now, getopts only parses options, so it will only parse this part, and then it'll stop and we'll have to handle this one ourselves. So, how does getopts work? Okay, we're telling it here what options we will accept, and opt is the name of the variable that will get each option as it is read. So, there's a while loop, which means we call getopts until no options are left on the command line, and then for every option found we use a case statement to see which option we actually found. In case it's the -r option, well it's very simple. All we have to do is remember that the user wants us to reverse the count, and we'll handle the logic for that later. In case it's the -b option, we need a number as an argument, and the argument to this option will be in the OPTARG variable. Of course we have to check that OPTARG is a number. Again, this is a pattern matching as we've seen before to check that it's actually a number, and if it's not, we print an error and exit the script. The -s option works exactly the same. Note, that I store the argument to the option in a variable. So, here the number we want to start with is stored in start, and the step is stored in step. My case statement also has a part that matches a question mark. This is not a pattern. If I would use this, it would be a pattern that matches any character, but in this case I'm simply matching just a question mark. And in that case, that means that either the user gave an option that we're not accepting or we're actually missing an argument for the B or the S option. Both of

these cases will be seen by getopts, it will print an error message, and then we can simply exit the script. So, at the end of this of course there's still an argument left, and we'll have to handle that ourselves. Now, getopts leaves all our arguments to the script intact, so we'll have to shift all the options away so we only have this one left. And to do that we can use the OPTIND variable, and that gives the actual index of the next option that getopts was going to handle and to shift everything away and make sure that our last argument is in $1. We can use this. So, we subtract 1 from OPTIND and give that as an argument to shift. So, then our last arguments will be in $1 here, and of course we have to check that it's actually there. So, if not we will print an error, and if it's there we put it in end. And then after that we can simply use a for loop for start to end using the step here as well and then print every number in the loop. And if reverse is true, we use this loop, which loops the other way around. So, to give you an example how this works, let's just start by making an error like this. Now, we've told the getops that we want -s to take an argument. So, if I do it like this, getopts prints option requires an argument. So, I should give it an argument. Let's give it an argument of step size 2 and count to 10. Very well. Now note that if I will do this, if I forget an option for -s and give another option after it, now actually getopts will think -r is the argument to -s. So, we do get here, but we detect -r is not a number, and this is what we actually print. Another thing we can do wrong is to use an invalid option, let's say -a, and then getops says illegal option. So, that's a basic use of getopts. You simply tell it what kind of options you expect, and then you use a case statement to handle each option in turn, and normally you will set some variables to remember that for the rest of your scripts, and there's the question mark here in case something goes wrong. At the end you can use this line to shift all the options away and go along with parsing other arguments that come after that. So, getopts will only parse options that start with a dash and consist of a single letter. So, that means that any word that's not starting with a dash will end option processing, and you'll have to handle the rest of the line yourself. So, this means basically all your options will have to look like this. First you can handle some options starting with dashes like -x, -y, and then you'll have to handle the rest yourself, file1, file2, file3, etc. Of course in this case it is possible that file1 is an argument for -y, but file2 and file3 will not be handled by getopts. Also, if there's an option that consists of two dashes, that will be seen as the end of options as well, and this enables you to add at the end of the line arguments that you don't want to be parsed by getopts, but that start with dashes anyway. Now, any options that take arguments will have those arguments put in OPTARG. Also, OPTIND holds the index of the next argument to be processed, and we've seen how you can use this to shift all the options away and then handle the rest of the arguments yourself.

Getopts: Handling Errors

We've seen that getopts can handle some errors in the options for you. By default it's a non-silent mode, and that means getopts handles errors for you and also prints these errors to the command line. Now, if anything goes wrong, the option variable name holds a question mark, but in my experience most of the time that's not really what you want. First of all, you may want to decide for yourself what the error strings look like, and secondly you may want more information than just a question mark. So, how can you process these errors by yourself? Well, if you start the option string with a colon, you put getopts in silent mode, and that means it won't handle errors, and it won't print anything. So, in that case, the option string in our script would start with a colon like this. And when you do this, the following happens: When getopts finds an unknown option in our example case, and that wouldn't be an S or an R, but maybe an A, again it will put a question mark in your option variable, but it will put the actual option in the OPTARG variable, and that's something it doesn't do when it's in non-silent mode. But when it's in silent mode and it finds a -a options when you're not expecting one, it will put the letter An in OPTARG so you can handle that and print a nice error message for yourself. And in the case an option needs and argument, but that's missing, it will put a colon in the option variable, and again it will put the actual option in OPTARG so you can handle that yourself. So, let's change our script a little bit to take advantage of this. So, this is what our script would look like if we don't let getopts handle errors. Again, now there's a colon here in the option string, and I added a colon pattern here in the case statement, and in that case I print which option is missing an argument. So, let's call this script with an option that needs an argument like -b, and let's forget the argument for -b. Sorry, I need to use the count2 script of course. And now we end up here. Getopts will put a colon in our opt variable, and I can print the message that this option is missing an argument and the B here will actually be an OPTARG. The same way, we can now detect unknown options like this. So, this gives you some extra control over the error messages that your script prints, and, for example, you may want to print a helpful message that explains the option that misses an argument.

Summary

So, this concludes this module. We've seen several special variables, namely the positional parameters, $* and $@ and $# and finally $0, which holds the name of your script. We've seen the shift command, which will remove the arguments from the command line one-by-one. And mainly we've taken a look at getopts, which helps you parse your options.

Shell Functions

Introduction

[Autogenerated] Hi, this is Rachel Ecker, and in this module, I'm going to talk to you about how to use functions in a shell script. Now, like any decent programming language, fish will let you define a function in your scripts. And after watching this model, you know how to declare a function and how to call it, and also how you can have a function returned. Some useful data that you can use offer that in the rest of your script you can also export functions. Now, during this module, I'm going to show you several demos, and we'll also come across some features off bash that I didn't talk about yet.

Shell Functions

Now, in Bash basically a function will let you define your own command, and you do that like this. First you give it a name, then you use an opening and a closing parens, and then comes the code you want to execute within braces. After you do this, you can run this code between the braces just like it's a command. Now, there are two different ways to define a function. First of all, you can use the function keyword, and then it looks like this or you can use the function keyword and just leave out the parens altogether, but I do recommend that you use this syntax in bold for compatibility. Now, like I said, when you do this, you can run the code in the braces like any normal command, and that also means you can give it arguments. And when your function generates output or takes input, you can also redirect that output and input. Also, when you give your function arguments, you can use the normal positional parameters, 1, 2, 3, etc. just like when you take arguments with a script. So, within a function, $1, $2, etc. will stand for the function's arguments and not the script's arguments. So, in a way you might think of a function as a tiny little shell script inside a shell script, and for that reason, because you're making a new command, it's very important that you name your function correctly, and basically the same rules that you have when you name your script also apply to your functions. So, if you have a function called ls, it will override the ls command. So, here is a little script containing a simple function. The function is called sum, and as you can see it takes two arguments, $1 and $2 and adds these together with an arithmetic expression here and then returns the outcome of that sum. And here's an example of calling that function, and you can see it's basically the same as calling any command. And because this returns the outcome of the sum as its return value after calling the function here, the return value will be in $?. So, if we now run this script, it prints 9. Now, of course, this return status

in my opinion isn't the most elegant way to pass outputs to the rest of your script. So, for example, I'd like it more if we'd just print it here to standard out, and then I don't have to do this at all because the function here will simply print stuff to standard out. So now if we run it, again it prints 9. Or of course I can also now use this output in a command substitution. So, again, this prints 9. And of course this will let you, this command substitution will let you pass this output to many other commands than just echo. Now, let me show you another example. I'm just going to copy that in here, and this is a simple function where we actually have a more useful use of the return value. So, let me make this a bit bigger. And in this case our function is simply a check whether the first argument starts with either a lower or an uppercase A. And as you can see here, this returns the actual return status of this test. So, starts_with_a the function will return 0 if its first argument starts with an A and another value otherwise. And that means that now we can write an if statement that uses this. Let me show you. And as you can see here now we use this function in an if, then, else, and of course an if, then, else actually looks at the return value of your function. So, in the case you're actually using your function in a test, calling return actually makes sense. So, in this case let's call it again. We see that now because this string here starts with an A we say yup. Now, I can also completely leave out the return statement, and in this case the whole function will simply return the value of its last command. So, basically this is exactly the same, and as you can see, it runs as well.

Shell Functions 2

Variables you declare in your shell scripts are normally visible for the whole script, so they're globally visible, but in a function it's possible to make a variable that's local only to that function so it won't be visible outside of that function. And to do that, you simply declare it inside the function or you use the local keyword instead of declare, and that will have the same effect. Now, like I said before, you can think of a function as a tiny shell script inside a shell script, and of course you can exit a function as well. And for that, there's a separate command, not exit because if you use exit you will simply exit the whole script, but there's a command return, and that simply returns a status code just like exit would. Now, if you don't use a return code with the return statement, the return status of the function will simply be the return status of the last statement inside the function so you don't have to use return. Now, of course, a status code is just a number, but sometimes you want your function to return something else like a piece of text. And how do you do that? How can you return something else than a number? Well, there are two basic ways. First of all, you can use a global variable where you just have a variable that's declared outside of the function and so that's globally available so the other code outside your function can also see

it, and when you change that variable inside your function, the rest of the script will see that. What you can also do is simply set the data to output, so use something like echo or printf. And then when you call the function with, for example, command substitution or redirection, you can then use that output in the rest of your script. Now, you can also think of a function as being a variable that simply points to a piece of code, and thinking of it like that it's logical that you can also export a function. And to export a function, you simply call export with the -f switch. After that, the function will be available to subprocesses just like an exported variable.

Functions: Demo

So, now I'm going to take a look at several examples of functions inside shell scripts, and we'll come across some peculiarities as well. First of all, here's a little script called box, and I'll show you what it does. It takes a single argument and then prints that inside a nice little box. And to print these lines, I'm using a function called drawline. And we've seen this functionality before as a script, but now I've put inside a function. And as you can see, this function inside it has two local variables, line and char, and line will be filled inside the for loop and then printed, and char actually represents one single character of the line. So then all we have to do is first, of course, check whether we have an argument, then determine the length of our first argument, and I add four characters to that because that will be the whole length of the whole line, and then I can simply say drawline len. And as you can see, the loop to print the line will go up to the first argument, and then we actually print the word in between. There's a format string here that will put the two pipe symbols at the start and the end of the line, and then we print another line. Now, you might be surprised that I'm not using a dollar sign before len, and this shows you a little peculiarity because $1 in the for loop will be expanded to the string len, and that will be interpreted as meaning the variable len inside the arithmetic expression. I have to say that's probably not something you want to use in production code, but I wanted to show you anyway. So, let's look at something else. There's the count script. We've seen this before, and I've added two functions here. First of all, there's the usage function, and that prints a nice little message in case the user doesn't know how to use the script. So basically right now if I call count like this, my script prints a nice little message about how to use the script, and such a usage function is something you will run across a lot in shell scripts. Now, we see a little bit of new syntax here as well, and that's a new kid of redirection, and it's called a here document. What it does is take this part of input from the script and use it as the input for this command. So, it will send this text here to the standard input of cat, and the syntax for a here document is that you use two less than signs and then some tag of your own choosing. I like to use END as you can see. And as

soon as Bash sees that word again, which is here in this case, it knows that my input has ended. So, everything between this occurrence and this occurrence of the word will be the input of my command. Now, the most basic use I have of this function is when the user gives the -h option in which I simply call usage, which will print my information to standard out, and then I exit the script. And in case something goes wrong, so for example someone uses the -b option with something that's not a number, I use the error function. And I've declared the error function here, and it simply prints the word error followed by its first argument, then it prints the usage, and then it exits the script with its second argument. So, going back here where I call it, the message will be the actual parameter supplied and then the text that it's not a number, and it will exit the script with status code 1 because that's the second argument of my script. Now, note as well that the error function here has a redirection applied to it, and this will mean that all the output of the error function will in this case go to standard error. So, although usage, the other function I made, simply prints to the standard output, that's the standard output of that function, but I call it from error and then redirect that to the standard error. So, this output, as well as this output here will go to standard error. Then here you see the isnum function, which uses a pattern here to check whether its first argument is actually a number. So, now I can simply say something like this. Is my argument a number? This is a test so it will simply look at the status code of this function, and if it's not a number I call my error function, exit the script with 1, and print this error. And this is what that looks like. Here's my error, and then I print the usage of the script. Now, let's look at a very common pitfall that a lot of new Bash users tend to hit at some point, and it looks like this. Suppose I have a function here that counts the lines in its input, so it simply uses a read loop and increases a variable called count for every single line in the input. Now, note that the count here is a global variable because it has been declared outside of the function, so you would expect this count variable to be available both inside and outside of the function. So, here I print all the arguments given to the function and call them as a command, and then I pipe that through my count_lines function. And let's see what happens. So, my script should execute ls here and then count the lines, but it actually prints 0. Now, let's see whether my function actually works. (Typing) Well, apparently I do have 26 files. That's what it prints here. So, why is count 0 outside of the function? Well, actually when you pipe something like this, the command in the pipe, which in this case is my function, runs inside a subshell, which means it runs inside a separate Bash process, and that means that this count variable will be local to that subshell and will not be available to its parent process as you know. So, that means that this count variable, which is local to this script doesn't get updated. Now, there's not really an easy way to get around this, but I can show you a different script, and that's going to be the most complex script I'm showing you in this course, and it's called hist. Let me start by showing you exactly what it does. It prints a nice

little histogram of how large this subdirectory to the current directory are. And as you can see, 64% of all the data in my home directory is in my dropbox. So, let me walk you through this script and see what it does. First of all, there's the error function. We already know that one. And then there's something called my_mktemp, and the mktemp command will make a nice temporary file for you. It will generate a random filename and put that in something like /tmp or in another place. It kind of depends on the kind of UNIX you are running. On Mac OS, for example, it won't be /tmp. But anyway, what's important here is that I call mktemp, but depending on whether I'm running on Linux or BSD, I have to give an argument. So, first I try without an argument. If that doesn't run correctly, I try it with an argument. And as you can see because this first one will print an error message if it doesn't run, I have to redirect errors inside this function to dev/null so we don't see that in the terminal. Then I'm actually checking whether we are using Bash 4, and that's because I'm using associative arrays in this example, and those are only supported by Bash 4 and not by previous versions. But Mac OS comes with Bash 3 installed, and that's why I'm using a different Bash executable in the first line because I had to install Bash 4 separately. So, then I declare several variables. First of all, here I declare an associative array, and that's an array that will hold all the sizes of the different directories. Then I have a tempfile variable that will hold the location of my temporary file, and then I call the tput command to find out how wide my terminal actually is to be able to do some nice pretty printing. Then I declare some more global variables like the longest file name I'm going to print, the largest file I've seen, and the total file size. Here's my function to draw a line that draws all these little lines here, and then here I have a function called read_filesizes. What this does is it reads in a while loop from every single line in its input a size and a name, and this data it stores in my associative array. And as you can see here, the index to my array isn't a number, but in this case it's a filename, and that basically is the difference between an associative array and a normal array. And the value associated with that index is the file size for that file. Then in the next three lines I update several global variables. So, basically we calculate some data here that we need in the rest of our scripts. So, this shows you how you can use global variables to pass data to the rest of your scripts. The array by the way is also global. Now, how do we get our input? Well, there's a command called du, and du will list the amount of data that is stored in your directories, and I call it here with all the directories in my current directory. You can also see that I use the same trick as with mktemp above because these are the options du would take on BSD, and this is the equivalent for Linux. And of course I could have put this in a function like mydu like I did with my mktemp, but for some reason I chose not to. Again, we redirect the errors to dev/null, and then there's a little trick. We save the output to tempfile. And remember that you might try to use a pipe here and pipe this directly to read file sizes, but of course then all these global variables here would be suddenly local to our subshell. So, all this

data would be generated, but it wouldn't be available to the rest of the script. So, that's why you first save it to tempfile, and then you use tempfile as the input for our read_filesizes. So, after reading all that data I do some calculations of how wide the lines should be, and then I have a nice for loop where I loop over all the keys in my array. And for every key I get the sizes in there and calculate the length of the line we should show, what the percentage is for that line, and then we use a single printf here to print all the data for that line. And here you can see I use a command substitution. This part will draw all these dashes on the line, and we include that here on the line with the rest of the data. Well, then finally I print some totals, and of course I clean up my temporary file, and then I exit the script. So, what does this script show you? Well, a lot of things. We have a redirection here and another redirection there for functions. I've showed you an associative array for the first time. We also see how to use global variables to pass data around and how to use a temporary file for sending data to a function. Of course you can use this to send data to a script or a command just as well. And finally, here's a command substitution, which uses one of our own functions.

Some Miscellaneous Remarks

So, there were three things in these demos that we came across that I want to show you again in a sheet. And first of all we saw functions and redirection. Now, since you call a function just like a normal command, you can apply redirection whenever you call it, but you can also apply redirection immediately after defining the function. And in this case, the redirection will be executed every time the function is run. To do this, you define your function like this. You simply apply the redirection immediately after the function. And in this example every time I call fun, all the output that's generated inside the function body for a standard output will be redirected to the standard error. Now, there's also a very important pitfall you should know about, and that's the fact that a command inside a pipeline runs in a subshell. So, for example, this loop here, this is a read loop that reads from the ls commands, tries to update a count variable, and actually it will be successful in updating that count variable, but unfortunately it does that inside a subshell. So, the updated value of the count variable won't be visible outside of the loop so it's basically a useless loop. And similarly we've seen this in the histogram script trying to send the output from the du command to the read_filesizes function will not work with a pipe, and instead I had to use a temporary file. Then there's some new syntax we came across, and that's called a here document. And using a here document you can put data inside your source file and have a command read that as its input. And it looks like this, two less than signs and a tag. And you can choose this tag yourself. And the second occurrence of this tag defines the end of your input. So,

for example, in this case I have the text to use as input goes here including the leading spaces by the way, and its starts and ends are denoted by an uppercase END, and the whole line then is used as input for the cat command. Now, basically you can use this to send large amounts of text data to your commands.

Summary

So, what have we seen in this module? First of all how to define a function, and this is my preferred syntax for that. Then we saw how to call a function and how to send it arguments. And within the function you can use these arguments with the same positional parameters as when you're using the arguments to your script. We also saw how to redirect input and output when you call a function, and we also saw how to return data from your functions. And the most basic way would be to simply set a return value, but you can also send stuff to output in which case you can do something like use a command substitution or use redirection to have the output end up somewhere else, and finally there's the option of using a global variable. There was also a bit of more or less unrelated new syntax we saw, and that's the here document. And finally, there's an important pitfall you should know about, and that's the fact that a command in a pipeline runs inside a subshell.

Fun with Strings

Introduction

[Autogenerated] Hi, this is Ranger Lacquer. And didn't this module will see some techniques for manipulating strings with Boesch? The first thing we'll see some new Syntex called perimeter expansion and that will let you do various things. First, we look at removing a pattern from a string. Then we'll see how to search for pattern and replace it. And finally, we'll see how to give you a variable, a default value. When it's not set, the second thing will look into is how to do pattern matching inside a conditional expression. And then, finally, there's something more or less related, called the end off options.

Removing Part Of A String

So, parameter expansion is a technique that allows quite powerful string manipulation inside Bash. And we've one example of it already, and that's this, using the name of your variable within braces and then prefix that with a dollar sign. And then before the name of your variable you use a hash, and in that case you get the length of your variable. And the parameter expansion syntax is exactly that. It's using braces and a dollar sign, and then inside the braces you use some special characters to do a manipulation of your variable. So, let's look at the first application of this, how to remove a pattern. So, by removing a pattern you can remove a part of your string, and this is the first example. Here you can see that we put the hash sign after the name of the variable instead of before it, and then we use a pattern. And in that way, you can remove something from the start of your string, and it will match the pattern, but it will only take the shortest possible match and remove that. Now, sometimes you don't want to remove the shortest possible match, but you want to remove the longest possible match, and in that case you use two hashes. Now, similarly you can also remove something from the end of your string, and in that case you don't use a hash, but you use a percent sign. And that again will remove only the shortest possible match, and if you use two percent signs you can remove the longest match from the end of your string. Now, these patterns are very similar to what you're used to with filenames. You can use a star for matching any string, you can use a question mark for matching any single character, and you can use sets of characters inside the brackets. Now for some examples. Let's take the string Users/reindert/demo. txt and put that in a variable I. Now, if I do this, this is a parameter expansion where I say I want to remove the shortest match for my pattern from the beginning of the string. And my pattern is a star, which will match anything, followed by a slash. And of course this star will also match an empty string. So, the shortest possible match of this pattern is simply just the leading slash, so this only removes the leading slash from my filename here. Now, suppose I use two slashes like this, and the star will match as much as possible, but it still has to be followed by a single slash. So, in this case a star will match anything up to the word reindert, and then the slash will match the slash after that, which means we're removing all the directory names and only leaving the actual filename. Now, sometimes, for example, you want to remove an extension, and for that you have to remove something from the end of the string, and you could do it like this. So, this will match the shortest possible string that starts with a dot. And in this case that removes the extension. And in the same way we can say okay I'm going to remove the whole filename and only leave the path, and for that you remove a slash followed by star, and that will match the whole filename including the slash before it.

Search and Replace

You can also search for a pattern and replace it, and to do that you use this syntax. So, the name of your variable followed by a slash, then the pattern you want to match, then another slash, and then the string you want to replace your matches with. And in this case we only replaced the first match of the pattern with a string, but you can also say I want to substitute every match. And in that case you use two slashes after the variable name, and that will substitute every single match with a string. Sometimes you only want to match something that's at the start or the end of your variable. So, in that case you can use something like this. Here we have a slash and then a hash, and that will only match a pattern at the start of the string. And again, of course, we can use a percent sign, and that will only match the end of the string. So, let me just show you a little bit of these replacements in practice. So, I have a variable I here, and it contains the filename mytxt. txt. And let's say I do this. (Typing) Of course this is a replacement and it will replace the occurrence of txt with jpg. And as you can see, it only replaces the first occurrence. And, as you know, if I want to replace from the end, so just replace the extension, I use the percent, and now the extension has changed. I can also change every single occurrence like this, and I can even remove every occurrence by simply saying I want an empty replacement string, and then we're only left with my and a dot. Similarly, if I only want to replace the last occurrence, I would do this, and that's similar to using a removal like this. Now, removing from the front of the string doesn't do anything at all because the string doesn't start with txt. Now, of course we can also use patterns. So, let's say I want to replace every Y or X with an A. Well, if I do it like this, it will only replace one single occurrence. And if I want to replace all of them I use two slashes, and we end up with this. So, that should give you a little bit of an idea how this works, but let me show you a very simple script example, and this is a script we already saw before, and it takes two arguments. The first one is an extension to remove, and the other is an extension to replace it with so you can use it to change the extensions on groups of files. So, now let's create some files. Let's say I want to create three text files, and I can use mvext to change all of these extensions from txt to say jpg. Very well. And before in this script we used the base name utility to find out the extension of a file, but since now we know about replacements, we don't need that anymore. All we do is we take the current filename and replace the current of the first argument with the second argument. And note here I use a percent because I only want to match the end of the string, which is the extension of the file. Now, one big advantage of doing it this way is that we don't have to call an external utility, and that means we won't have to start an extra process. So, in case I use this script to rename huge amounts of files, let's say 10, 000 files, this will run faster than the other version of the script that uses an external process because in this case we don't have to start a new process for every single filename. So, in general, instead of using external utilities to manipulate your strings, it will be a lot faster if you simply use the built-in Bash parameter expansions.

Setting A Default Value

Parameter expansion will also allow you to detect whether your variable is set and if not give it a default value. To do that, you use this, the name of your variable followed by a colon and then a minus sign. And now in case your variable either isn't set or it contains an empty value, this expression will evaluate to the value you give in the expression. Now, sometimes you want to allow a variable that does contain an empty value and only use this default if the variable isn't set. And to do that you leave out the colon. So, this behaves kind of the same, but the difference is in case the variable is sent to an empty value, it will evaluate to the empty value, and only if your variable is unset it will evaluate to the default. Now, we can go one step further and not just evaluate to a default, but immediately assign that default to our variable, and it looks like this. In this case, we use an equals sign. Now, this works pretty much the same. In case my variable is either set to an empty value or not set at all, this evaluates to my default value, but it also assigns it to the variable. So, in that case, after seeing this expression, Bash will have set your variable to the default. And, again, if you leave out the colon, this will allow a variable with an empty value. Now, there are many more things you can do with parameter expansions like, for example, change the case of a string or extract a substring by index. And in case you want to know more, visit this page, or you can of course look at the Bash man page. So, just a little practical example of this. Remember in the note taking script I'm trying to read this environment variable called $NOTESDIR. And in case it's set, I set this lowercase notesdir variable to its value. But if it's not set, it already has a value, and that's this default value ${HOME}. So, I could replace these two by saying this, checking whether NOTESDIR has a value, and if it doesn't we return the value of $HOME. And, of course, I shouldn't make typos here. So, that's one example of where this might come in handy.

Conditional Expression Patterns

Now, since we're talking about strings, I also want to have a short word about conditional expressions. We've seen the equals and unequals operators in the conditional expression, but what I didn't really tell you yet is that they actually do pattern matching. And before I go into that, I also want to say that the single equals sign is exactly the same as the double equals sign. Now, this actually will return true whenever $var matches the pattern, so this is not a string comparison. It's a pattern matching. And again here, pattern matching is exactly like pathname matching with a star and the question mark and the brackets. So, for example, this will return true whenever the filename variable contains a filename ending with. txt. So, it doesn't match for the string *. txt, but in case you use quotes around your pattern, it does do string matching. So, this actually matches

the exact string with the brackets and the digits and the star, and that's because we're using quotes around it so it's not interpreted as a pattern. So, basically what this means is that in this conditional expression here the right part is interpreted as a pattern, and so the star here will match these three characters. And of course then the test returns a true value, and that means we see the word yep here. Now, in case I do this, we now have string matching so it won't return true, and so we see that there's no echo yep here because of course this will only be executed if the test returns true. Now, I'll just show you that this is actually an XX string matching because this returns true again. And the final thing I want to show you for now is that if you by mistake would use the classical test, so the classical test command with one bracket, which I specifically told you not to use, that if we try to use pattern matching right now, and let's put the word hello back here, that's also going to fail in this case because here test is a simple command, and that means that this is not treated special in anyway by Bash, and Bash will see this as a filename expansion. So, it will look for any files in my current directory that match this pattern and replace the pattern with the matches. And right now I don't have any files in my home directory that match this, so this part will be empty, and that means that the test will return false. And in general of course you don't want filename expansion to occur in your tests here. So, that's really one of the reasons why I told you to use the conditional expression because the conditional expression, if you use a pattern without quotes, will not try to do pathname expansion, but to keep your pattern intact and then use it for string matching.

Regular Expressions in The Conditional Expression

Now, there's one special operator in a conditional expression, and that's the equals sign followed by a tilde, and that will do regular expression matching. And regular expressions are different from the normal pattern matching which Bash does, and it's much more powerful. And what it uses is something called POSIX extended regular expressions. Now, these are extremely powerful, and they're almost a little programming language by themselves so explaining everything about them here is beyond the scope of this course, but I can tell you some of the differences between the normal patterns and regular expressions. For example, the question mark inside a regular expression doesn't match a single character, but it matches the token before it for 0 or 1 times. So, in this example we have a list of characters between brackets, which will match any digit between 0 and 9, and the question mark after that modifies it to mean 0 or 1 times. So, this expression here will either match a single digit or the empty string. The same is true for the star. It will match the token before it for any number of times. So, again, let's have an example with brackets. There we match a single character from A to Z in lowercase, and then if we use a star

behind it, it will match that either 0 times or any other number of times. This will match any string made up of letters or nothing at all. And then there's a third one, and that's the plus, and that will match the token before it one or more times. So, this will match exactly one digit or a string made up of digits, but it won't match the empty string. Two other basic things that are really useful to know is that the caret will match the start of your string and the dollar sign will match the end of your string. Now, instead of going into this very deeply, I'm just going to show you a little demo. Now, what I want to show you is the isnum function. We've seen this before in the count script. And previously I only used one single expression to check whether the string was made up of digits, but I made it a little bit more complicated and a little bit more sophisticated. First of all, here's our original expression. And as you can see, we use the caret to mark the beginning of the string and the dollar sign to match the end of the string, and that is to make sure that we're not just matching a part of the string. We want to match the whole string. And then we take this expression within brackets, which will match any digit, and then the plus sign, which means we need at least one digit. Now as you can also see, I'm assigning my regular expressions here to variables. And the reason is that these regular expressions tend to use a lot of special characters, and because you need to use them here without quoting that can sometimes give problems because you have to escape these special characters. So, a nicer way is to use the expression within single quotes, which will escape everything for you, then assign that to a variable, and then using that variable inside the conditional expression without quoting again. Then here of course is the operator for matching a regular expression. And what we're doing here is checking that our first argument is actually completely made up from digits. And then if that's true, I'm using a second check. And here I'm checking whether at the start we have 0 because as you know numbers that start with a 0 in Bash are understood to be octal numbers, and I'm assuming the user is using the decimal system. Now, after this 0 I'm checking that there's any character, and that's what the dot means in a regular expression. So, a dot just doesn't match a dot, but it will match any character. So, in that sense it's equivalent to the question mark in the normal pattern matching with Bash, and I'm saying that after the 0 should come one or more other characters. And because we already checked here, we will know that all these characters will actually be digits. So, basically if our number matches this expression, we know it's a 0 followed by other digits, and that means it's octal. So, in that case I say $1 is not a number, and then comes something special. It uses the BASH_REMATCH array, and this will contain groups of matches. And I made a little group here with the parens, and it's the only group we are using here so everything that's matched by the dot and plus here will end up in a group. And after matching we can find it by asking for one of the elements of the BASH_REMATCH array. So, basically if the user enters 010 instead of just 10, then this will match the 1 and the 0 from 10, and that will end up

here. So, let me show you how this works. We received this error, and that's because this string is first matched by this regular expression, and then it's matched by this regular expression, and we get the error that you're probably using an octal number. And as you can see, this group here matches the part 20 here. And also if I want to use 0, we don't get our error because that won't match this pattern because this pattern is matching 0 followed by other digits. So, basically that's a very simple demo of stuff you can do with regular expressions, but it's really the tip of the iceberg because regular expressions are extremely powerful. So, this demo was in no means intended to teach you all the features of regular expressions, and you shouldn't feel the need to remember any of this. But in case you need regular expressions, I strongly encourage you to learn about them.

End of Options

Now I want to talk to you about something called the end of options, and this is actually pretty important because it may prevent some very nasty bugs. Now, the end of options is denoted by two dashes, and we've seen this before with getopts. And it's actually supported by just about any UNIX command. What it means is that anything that comes after it on the command line will not be interpreted as being an option. And why do you want this? Well, it makes it safe to work with data that might start with a dash. So, for example, in case you have a file called -l. txt on your file system and you try to remove it with rm -l. txt that will give you an error because rm will tell you I don't know an option -l. So, instead of doing that, you can do this. First you tell rm okay this is the end of my options, and then you're free to put anything after that including stuff that starts with a dash. And this is especially nice when you're using loops like this. So, let's say I want to touch all the text files in my current directory. I might want to do this. So, here I have a loop that loops over all filenames that end with. txt, and then I touch each of them. And just in case there might be a file that starts with a dash, I use the end of options with the touch commands so that won't give an error then. So, basically it's a very good habit that whenever you have a variable as an argument for a command and the contents of that variable are not under your control because maybe you don't know what's on the file system, use the end of options with the commands you call. And actually some of the scripts I showed you up to here contain bugs that can be prevented by using this, so let me show you that in a demo. So, let's start by making a filename that starts with a dash, and let's call it -a. jpg. And, as you can see, touch tells us -a. jpg is an illegal option. So, what I do is I insert the end of options, and that creates our file. But now using our move extensions script where I want to change all the jpg extensions into something else, now our script gives an error. And of course, again, that's because here I'm using move with data that's

generated by my script. So, to be safe, again, I have to use the end of options here. And let's try now. And now it works. Here's our file. And basically this really affects a lot of the scripts that I showed you. So, for example, let's find another one. So the note taking script for example. Here we have a mkdir command that uses input from the user, and it doesn't use end of options. And here's a touch. So, again, in both of these cases I need to use end of options to be safe. And really these bugs are all over the place. But instead of forcing you to watch me fixing all these errors, I challenge you to find some of them for yourself and fix them in the scripts I showed you as examples. But I want to be really clear about this. It's very important if you have external data, so data from the file system or from user input, that you don't forget to use end of options to prevent breaking your script when there's input that contains a dash at the start.

Summary

So, what have we seen? First of all, we've seen parameter expansion, which lets you remove a pattern from a string or search and replace a pattern, and you can also detect whether a variable is set and then replace it with a default value. We've also seen that you can use pattern matching in a conditional expression, and that there's even a separate operator that lets you match regular expressions, which are even more powerful. We've also seen the very, very important end of options switch, which provides some safety against data that might start with a dash.

Many Ways to Run Your Script

Introduction

[Autogenerated] Hi, I'm running to maker and in the lost module off this course, I'll show you many ways to run your scripts. I'll start by showing you several ways to run called from a file. Then I'll talk a bit about putting your coat in the background and using that no help command. Then we'll talk about the accept command and how you can use it to provide global redirection for your whole script. Will also see how to make sure that your script runs at another time. Then right now and finally I want to discuss to commands set and shopped with which you can change the behavior off Bess when it runs your coat.

Running your Code

Now for most of this course we've been using a hash-bang as the first line of our script and running our script as regular commands. And if you do that, Bash will start your script in a separate Bash subprocess. Of course it's important that if you do this you have executable permission for your script. But in case your script doesn't have a hash-bang and you can't edit it, maybe because you don't have permissions, you can call Bash and then give the script as an argument. And, if you run it like that, you also won't need to set executable permissions. It's also nice to note that if you do this you can give Bash extra options. So, if you want to debug the script, you can simply give Bash the -x option, and that will show you everything Bash does while it's running. Now, a very different way to run code is to import code in the current shell process, and there's a command for that. It's called source. Now, source has an alias, and that's a single dot. So, if you call your script like this, the code in it will be executed in the current shell process, and that means that all the variables and functions defined in that code will be available in your script, so you won't have to export anything because the imported code will simply be run as a part of your script. And this is basically what happens when you read. bashrc. So, Bash starts up and reads the code in your. bashrc in the current process. So, as a little example, let's remove the executable permissions for the count script, and let's see what happens if I try to run it now. Bash tells me the permission is denied, but I can still run it by calling it like this. And like I told you, I can also say bash -x, and then we see every single step Bash does in the process. Now, if we look at the script, there are two functions here I might want to reuse in other scripts as well. So, one thing I could do is move all of this into a separate file, which would be my function library, and then use the source command to read that file in here, and then these functions would be available inside the script.

Nohup and The Background

Now, you should already know that you can put any command in the background by ending the line with an ampersand. So, for example, this would put my script in the background, and that means it will be disconnected from the interactive session. So, it won't take any user input, and if it tries to read from the terminal it will be suspended. This can of course be very useful if you want to get other stuff done while having a long-running script run in the background, so basically that's multitasking. But there's also another use. And this is if you have a long-running script, and you want to keep your script running when you exit your terminal session. So, for example, when you have a remote connection and you have to log out, but you don't want your script to exit, then you use the nohup command, and nohup is short for no hangup. Nohup will

disconnect your script from the terminal as well and keep it running, and usually you'll also want to put your script in the background in that case. Now, when you do this, and especially when your script does a lot of work, sometimes it's good to run your script with a lower priority to make sure it doesn't take up too many system resources. And there's a command for that as well, and it's called nice. And if you say nice myscript, it will run with a lower priority. Now, of course, you can combine these commands. So, if you have a long-running script you can say nohup nice myscript, and nohup will run the command that comes after it, and so will nice. So, nice will execute myscript, and nohup will execute nice myscript. So, here's a very basic script. It simply has a while loop, and inside the loop it prints a message with a number that increases, and then it sleeps for a second. And every iteration of the loop we call true, and true doesn't do anything. It simply returns a true value, so this is basically an endless loop. Now, if I run it, this is what I see. Now, suppose this is a very important command and I'm logged in remotely, but I need to log out, but I want this command to keep on running. This is what I would do. First, I would redirect all the output to a log file to make sure the data is still there when I get back. Of course I put it in the background as well, and I call nohup. Now, if I run this, the program runs in the background, and we can tail the log file, and we can see that it's printing. Actually, we can also follow the log file, and then we can see that it's actively updating. So, let's exit this session and start a new one. And now when I do a tail, we can see that the program is still updating. So, this shows you that with nohup you can create a command that keeps running even when you end your terminal session.

Exec

Another very useful command is called exec. One of the uses of the command is to redirect I/O for the whole script, and that's useful, for example, when you're going to do logging. So, for example, if you call exec like this at the start of your script, you're redirecting all the output to standard out in the script to logfile and all the output to standard error to errorlog. So, let me show you what that does in practice. Here's the stillhere script again, but this time I'm checking for an option -l. And if it's given, I'm calling exec and redirecting all the output to a logfile. So, let's see if this works. If I run it like this, I still get standard output, but if I call it with this, and let's put it in the background as well, let's see if it's now running and sending everything to logfile. And yes it is. So, basically this is a very nice way to set the summations for your output for your whole script.

At and Cron

In some situations you might want to run your script at a time when you're not there or at a specific time or repeat it at a given interval, and UNIX systems come with two standard utilities that let you do exactly that. By the way, I'm not going to explain in detail how these utilities work because that's more of a system administration topic, but of course you can simply consult the man page and find out how they work. Now, first of all there's at, and at will run your script at a specific time. For example, if I call at like this with a -f switch, I tell it that the myscript file is what I want to run, and then I tell it to run it at noon tomorrow. And there's many, many other ways to give it specific times. Now, the second utility you should know about is called cron, and cron will run your script according to a schedule. Now, this schedule can be just about anything. You can have it run your programs weekly or daily or hourly or whichever combinations of those you like. Now, in case you're running on Mac OS, there's also something called launchd, which is actually the preferred way to do it on that system because it provides some specific features that cron doesn't. Then if you run Ubuntu, there's a new demon that's set to replace both at and cron called Upstart, and you may want to look into that.

Set and Shopt

Now, there are two Bash built-in commands that you can use to customize Bash's behavior. First of all, there's set. Set has a lot of options. I'm not going to show all of them to you, but I'll just show you some that may be useful when you're debugging a script. Now, note that most of these settings are not intended for using production, so use them when debugging and then remove them. Now first of all, this is one we've already seen. Set -x will print each command with its arguments as Bash executes it. You can also set -u, and that will give you an error when you're using an uninitialized variable. So, you might want to use -u to guard against using unset variables or typos. There's also the -n command that will only read the commands, but it won't execute anything. So, if you want Bash to simply read your script to check if it's valid, but don't execute it, you can use -n. There's also -v, which will print each command as it is read. So, if you combine -n and -v, Bash will print every line, but won't execute anything. Finally, there's -e, and that will exit the script whenever a command fails. So, any external command that your script executes that returns a non-0 status code will terminate your script. That is, unless you're checking the status code yourself with something like if or while. Now, there are many, many more options, and of course you can always check them out with the help command. Now, I also have to say that some of these, mainly the -e option, can give unpredictable results and should never be used in a production environment. Then there's the shopt command. It also sets Bash behavior, but in a very different way. You can set options with -s and unset them with -u, and these options have

names. For example, nocaseglob will make Bash ignore case when it does pathname expansion, or the extglob option will enable extended pattern matching, which has some more powerful pattern matching. By the way, this is not the same as regular expressions. Then there's dotglob, and dotglob will make Bash include hidden files by default when you do pathname expansion. So, in this case, the star will also match hidden files that start with a dot. But unfortunately to make things a little bit more complicated, there are also some options, for example noclobber, that are set by set and now by shopt, and there really doesn't seem to be any logic behind this so it's often very hard to tell if an option has to be set by set or shopt, and I really can't give any pointers. You will have to read the documentation. But an example would be noclobber, and that will prevent overwriting files when you're doing redirection. So, if you're using the greater than sign and try to write to a file, but that file already exists, then the noclobber option will make Bash give an error so you won't override anything. And like I said, again, there are many more options, and if you want to know more about them, read the documentation.

Summary

And that brings us the end of this course. In this module, we've seen how to run your scripts in various ways, how to put your script in the background and use nohup to make sure it keeps on running when you look out, how to use exec for global redirection for your whole script. UNIX offers two services named at and crom to schedule the running of your script for another moment. And finally, we've seen the set and shopt command to change Bash's behavior. I hope you enjoyed watching this course, and I wish you good luck and a lot of fun writing your own Bash scripts.

Course author

Reindert-Jan Ekker

After studying Artificial Intelligence, Reindert-Jan has worked in many different roles in the software development life cycle, from tester and developer to Scrum Master. Then he decided to go...

Course info

Level Intermediate

Rating	★★★★½ (560)
My rating	★★★★★
Duration	4h 33m
Updated	16 Sep 2019

Share course