ALX LESSON

0x03. Shell, Init files, Variables and Expansions

Shell - Bash

TABLE OF CONTENTS

01

OVERVIEW topics

Ubuntu Shell Commands

Topics

what are the /etc/profile file and the /etc/profile.d directory

what is the ~/.bashrc file

what is the difference between a local and a global variable

what is the reserved variablehow to create, update and delete shell variables

what are the roles of the following reserved variables: HOME, PATH, PS1

what are special parameters

what is the special parameter #?

Ubuntu Shell Commands

Topics

what is the expansion and how to use them

what is the difference between single and double qoutes and how to use them properly

how to do command substitution with $() and backticks

how to perform arithmetic operations with the shell

how to create an alias

how to list aliases

how to temporarily disable an alias

shell arithmetic expansion converts

Ubuntu Shell Commands

Topics

```
printenv
set
unset
export
alias
unalias
.
source
printf
```

Slides On Telegram

https://t.me/alx_2023

Ubuntu Shell Commands

Topics

@ALX_2023

02

Learning Objectives

what are the /etc/profile file and the /etc/profile.d directory

The /etc/profile file is a system-wide configuration file that is executed by the shell whenever a user logs in. This file is typically used to set environment variables and define system-wide shell settings that apply to all users on the system.

For example, the /etc/profile file might contain commands to set the PATH environment variable, define shell aliases, and set default shell options. These settings would be applied to all users on the system, regardless of which shell they use.

The /etc/profile.d directory is a directory that contains additional shell scripts that are executed by the shell when a user logs in. These scripts are typically used to set environment variables and define shell settings for specific applications or subsystems.

For example, the /etc/profile.d/java.sh script might be used to set the JAVA_HOME

what is the ~/.bashrc file

The ~/.bashrc file is a shell script that is executed by the Bash shell whenever a user opens a new terminal window or starts a new interactive shell session. This file is typically used to define user-specific environment variables, aliases, and other shell settings.

The tilde character (~) at the beginning of the filename is a shorthand notation for the user's home directory. For example, if the user's username is johndoe, the ~/.bashrc file would be located at /home/johndoe/.bashrc.

The ~/.bashrc file is often used to define aliases for commonly used commands, such as ls or grep, and to set environment variables that are specific to the user's needs.

what is the difference between a local and a global variable

A local environment variable is a variable that is defined within a specific process or shell session. It is only accessible within that process or session, and it is destroyed once the process or session terminates. Local environment variables are often used to store temporary values or intermediate results in a program or script.

```
MY_VAR="Hello, World“
echo $MY_VAR
```

what is the difference between a local and a global variable

A global environment variable is a variable that is defined system-wide and is accessible by all processes and shell sessions running on the system. Global environment variables are often used to store system-wide settings and configurations.

For example, the PATH environment variable in Linux and Unix systems is a global variable that contains a list of directories where the system searches for executable programs. Any process or shell session running on the system can access the PATH variable and use it to locate programs.

what is the reserved variable how to create, update and delete shell variables

A reserved variable is a variable that is predefined and reserved by the shell for a specific purpose. These variables are typically used to store system-related information or configuration settings. Some examples of reserved variables include PATH, HOME, and SHELL

To create a shell variable in a bash shell, you can use the = operator to assign a value to a variable name:

```
MY_VAR="Hello, World"
```

To update the value of an existing shell variable, you can simply reassign it a new value:

```
MY_VAR="Goodbye, World"
```

To delete a shell variable, you can use the unset command:

```
unset MY_VAR
```

what is the reserved variable how to create, update and delete shell variables

It is important to note that shell variables are local to the current shell session by default. If you want to make a variable available to other processes or shell sessions, you can export it as an environment variable using the export command:

```
export MY_VAR
```

now MY_VAR is global variable available to other processes and shell sessions.

what are the roles of the following reserved variables: HOME, PATH, PS1

HOME: The HOME variable stores the absolute path of the user's home directory. This variable is used by many applications and scripts to determine the user's home directory, and it is also used by the shell to determine the default location for many operations, such as creating new files or opening a file manager.

```
echo $HOME
```

what are the roles of the following reserved variables: HOME, PATH, PS1

PATH: The PATH variable stores a list of directories where the shell searches for executable programs. When a user types a command in the shell, the shell checks the directories listed in the PATH variable in order to find the corresponding executable program. This variable is critical for running programs and scripts from the command line, as it determines where the system looks for these programs.

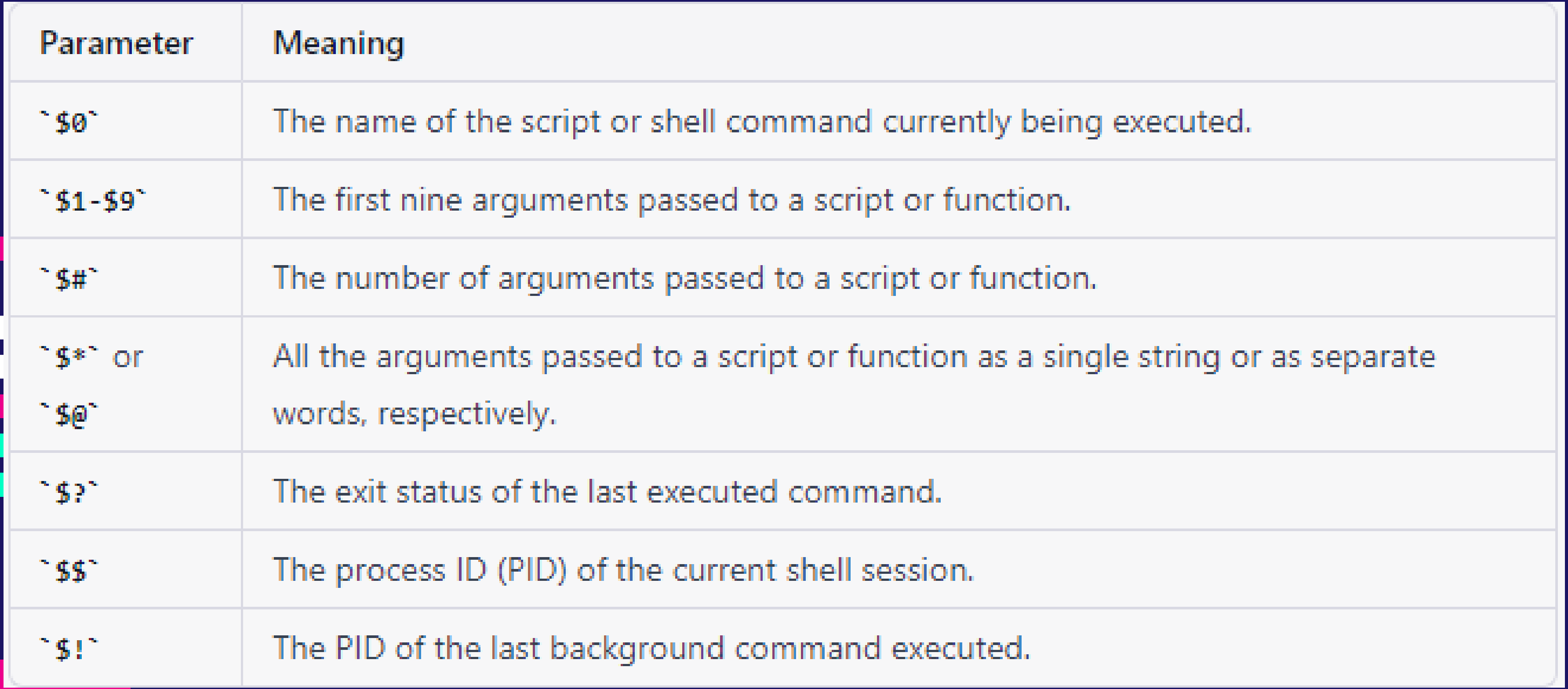
```
echo $PATH
```

what are the roles of the following reserved variables: HOME, PATH, PS1

PS1: The PS1 variable stores the primary prompt string for the shell. This is the text that appears before the cursor when the shell is waiting for user input. The value of PS1 can include various escape sequences and special characters that provide information about the current shell session, such as the current working directory, the username, the hostname, and more. By customizing the value of PS1, users can create personalized shell prompts that provide the information they need in a format that is easy to read and use.

```
echo $PS1
```

what are special parameters



Parameter	Meaning
`$0`	The name of the script or shell command currently being executed.
`$1-$9`	The first nine arguments passed to a script or function.
`$#`	The number of arguments passed to a script or function.
`$*` or `$@`	All the arguments passed to a script or function as a single string or as separate words, respectively.
`$?`	The exit status of the last executed command.
`$$`	The process ID (PID) of the current shell session.
`$!`	The PID of the last background command executed.

what is the special parameter #?

The special parameter $? contains the exit status of the last executed command in the shell.

The exit status is a numeric value between 0 and 255 that indicates whether the command was successful (exit status 0) or encountered an error (exit status greater than 0).

For example, if you run a command that succeeds, such as ls /etc, the exit status will be 0. If you run a command that fails, such as ls /nonexistent_directory, the exit status will be greater than 0 (usually 1).

You can access the value of the $? parameter using the echo command or any other command that can display shell variables:

```
ls /etc
echo $?
```

This will run the ls command on the /etc directory, and then print its exit status, which will be 0 if the command succeeds.

what is the expansion and how to use them

"expansion" refers to the process of replacing a specific set of characters or patterns in a string with their actual values or other text. There are several types of expansion in shell scripting, including:

Variable expansion - This is the process of replacing a variable name with its value. To perform variable expansion, you simply need to prefix the variable name with a dollar sign $. For example:

```
NAME="John"
echo "My name is $NAME"
```

what is the expansion and how to use them

Command substitution - This is the process of running a command and using its output as part of another command or assignment. To perform command substitution, you can use the backtick () or the $()` syntax. For example:

```
DATE=`date +%Y-%m-%d`
echo "Today is $DATE“
```

Arithmetic expansion - This is the process of evaluating an arithmetic expression and returning its result. To perform arithmetic expansion, you can use the $(( )) syntax. For example:

```
NUM1=10
NUM2=5
SUM=$((NUM1 + NUM2))
echo "The sum of $NUM1 and $NUM2 is $SUM"
```

what is the expansion and how to use them

Brace expansion - This is the process of generating a list of strings by iterating through a pattern or range of values. To perform brace expansion,
you can use the { } syntax. For example:

```
touch file{1..3}.txt
```

what is the difference between single and double qoutes and how to use them properly

there are two types of quotes that are commonly used: single quotes (') and double quotes (").

Single quotes are used to create a literal string, where all characters within the quotes are treated as plain text and no expansion is performed. This means that variables and special characters within single quotes are not evaluated and are treated as plain text. For example:

```
echo 'info $PATH'
```

This will print ‘info $PATH', since the variable $PATH is not evaluated and is treated as plain text.

what is the difference between single and double qoutes and how to use them properly

Double quotes, on the other hand, allow for expansion of variables and special characters within the quotes. This means that variables within double quotes are evaluated and replaced with their values. For example:

```
NAME="John"
echo "My name is $NAME"
```

This will print "My name is John", since the variable $NAME is evaluated and replaced with its value within the double quotes.

Double quotes also allow for certain special characters, such as \n for a new line or \t for a tab, to be used within the quotes. These special characters are not interpreted when used within single quotes. For example:

```
echo -e "Hello,\nWorld!"
```

how to do command substitution with $() and backticks

Command substitution is a feature in shell scripting that allows the output of a command to be substituted in place of the command itself. There are two ways to perform command substitution in shell scripting: using the $() syntax and using backticks (`)

Using the $() syntax:

```
echo "The current directory is $(pwd)“
echo "The number of files in the current directory is
$(ls | wc -l)“
```

Using backticks:

```
echo "The current directory is `pwd`"
echo "The number of files in the current directory is `ls
| wc -l`"
```

how to perform arithmetic operations with the shell

you can perform arithmetic operations using the built-in expr command, or by using the $(( )) syntax.

Here's an example of using expr:

```
a=10
b=20

sum=$(expr $a + $b)
difference=$(expr $a - $b)
product=$(expr $a \* $b)
quotient=$(expr $b / $a)

echo "Sum: $sum"
echo "Difference: $difference"
echo "Product: $product"
echo "Quotient: $quotient"
```

how to perform arithmetic operations with the shell

you can perform arithmetic operations using the built-in expr command, or by using the $(( )) syntax.

Here's an example of using $(( )) syntax:

```
a=10
b=20

sum=$((a + b))
difference=$((a - b))
product=$((a * b))
quotient=$((b / a))

echo "Sum: $sum"
echo "Difference: $difference"
echo "Product: $product"
echo "Quotient: $quotient"
```

how to create an alias

To create an alias, you can use the alias command followed by the desired alias name and the command it should run. For example, to create an alias for ls -l, you can use the following command:

```
alias ll='ls -l'
```

how to list aliases

To list all of your currently defined aliases, you can use the alias command without any arguments:

```
alias
```

This will display a list of all the aliases currently defined in your shell.

how to temporarily disable an alias

To temporarily disable an alias, you can use the \ character before the alias command. For example, to temporarily disable the ll alias we created earlier, you can run:

```
\ll
```

This will run the original ls command instead of the `ls -l` command that the ll alias runs. Note that this only temporarily disables the alias for the duration of that command. The alias will still be active for future commands.

To permanently remove an alias, you can use the unalias command followed by the alias name. For example, to remove the ll alias we created earlier, you can run:

```
unalias ll
```

This will remove the ll alias from your shell.

printenv

The printenv command is used to print out the values of all or some of the environment variables in your current shell session.
To print out the values of all the environment variables, you can simply run the printenv command without any arguments. For example:

```
printenv
```

To print out the value of a specific environment variable, you can pass its name as an argument to the printenv command. For example, to print out the value of the PATH variable, you can run:

```
printenv PATH
```

You can also pipe the output of the printenv command to another command for further processing.

```
printenv | grep PATH
```

set

The set command is used to set or unset options and positional parameters in the current shell session. It can also be used to display the current settings of options and variables.

To display all the variables and functions that are currently defined in your shell session, you can simply run the set command without any arguments. For example:

```
set
```

To set the value of a variable, you can use the = operator followed by the value you want to assign. For example, to set the MYVAR variable to the value "hello", you can run:

```
MYVAR=hello
```

set

To set a shell option, you can use the set command followed by the name of the option. For example, to enable the -x option, which causes the shell to print each command it executes (enable debugging mode in a shell script), you can run:

```
set -x
```

To unset a shell option, you can use the set command followed by the name of the option and the - sign. For example, to disable the -x option, you can run:

```
set +x
```

unset

To unset a variable, you can use the unset command followed by the name of the variable. For example, to unset the MYVAR variable, you can run:

```
unset MYVAR
```

Unset multiple variables at once:

```
VAR1="Hello"
VAR2="World"
VAR3="!"

unset VAR1 VAR2 VAR3

echo $VAR1 $VAR2 $VAR3
```

export

The export command is used to set an environment variable.

Unset a variable:

```
# set the MYVAR variable
export MYVAR="Hello World!"

# unset the variable
unset MYVAR
```

Set the PATH variable:

```
# add a directory to the PATH
export PATH="$PATH:/path/to/my/directory"

# call a command that is in the new directory
echo $PATH
```

. (dot)

The . (dot) command is used to execute commands from a specified file in the current shell environment. It is also known as the "source" command.

Suppose you have a file called myscript.sh that contains the following code:

```
#!/bin/bash
myvar="Hello, world!"
echo $myvar
```

Don't forget to give it execute permission

```
chmod +x myscript.sh
```

To execute this script and make the variable myvar available in the current shell environment, you can use the . command like this:

```
. myscript.sh
```

source

The . (dot) command is used to execute commands from a specified file in the current shell environment. It is also known as the "source" command.

Suppose you have a file called myscript.sh that contains the following code:

```
#!/bin/bash
myvar="Hello, world!"
echo $myvar
```

Don't forget to give it execute permission

```
chmod +x myscript.sh
```

To execute this script and make the variable myvar available in the current shell environment, you can use the . command like this:

```
source myscript.sh
```

printf

printf is a command that is used to print formatted output to the standard output (usually the terminal). It is similar to the echo command, but provides more control over the output format.

Here's a basic syntax of printf:

```
printf FORMAT [ARGUMENT]
```

```
printf "Hello, %s! \n" "World“
```

In this example, %s is a format specifier that is used to print a string. The string "World" is passed as an argument to the printf command, and is inserted into the %s format specifier.

printf

Here are some other common format specifiers that you can use with printf:

%d: prints a decimal integer.
%f: prints a floating-point number.
%c: prints a single character.
%s: prints a string.
%x: prints a hexadecimal integer.

Formatting a string using placeholders:

```
printf "My name is %s and I am %d years old. \n" "John" 25
```

printf

Yocan also use modifiers to control the output format. For example:

%-10s: left-justifies a string in a field width of 10 characters.

%10d: right-justifies a decimal integer in a field width of 10 characters.

%05d: pads a decimal integer with leading zeros to a field width of 5 characters.

```
printf "|%-10s| \n" "Hello"
printf "|%10d| \n" 42
```

Formatting a string with leading zeros:

```
printf "The number is: %04d \n" 42
```

printf

The decimal point rule with printf refers to the precision specifier used with floating point values. The precision specifier controls the number of digits to be printed after the decimal point.

In printf, the precision specifier is specified using a period (.) followed by an integer value. For example, %.3f specifies that a floating point value should be printed with 3 digits after the decimal point.

```
printf "%.2f \n" 3.14159        will output 3.14
printf "%.4f \n" 6.02214076     will output 6.0221
printf "%.0f \n" 42.9999        will output 43
```

printf

In printf, %o is a format specifier used to print an integer in octal (base 8) format. Here's an example:

```
printf "%o\n" 15
```

will output 17

In this example, the decimal number 15 is converted to its octal equivalent, which is 17. The \n at the end of the printf command is used to print a newline character after the output.

shell arithmetic expansion converts

Binary: base 2 (digits 0 and 1)

To convert binary to decimal: `echo $((2#binary_number))`

To convert decimal to binary: `echo "obase=2; decimal_number" | bc`

Octal: base 8 (digits 0 to 7)

To convert octal to decimal: `echo $((8#octal_number))`

To convert decimal to octal: `echo "obase=8; decimal_number" | bc`

Decimal: base 10 (digits 0 to 9)

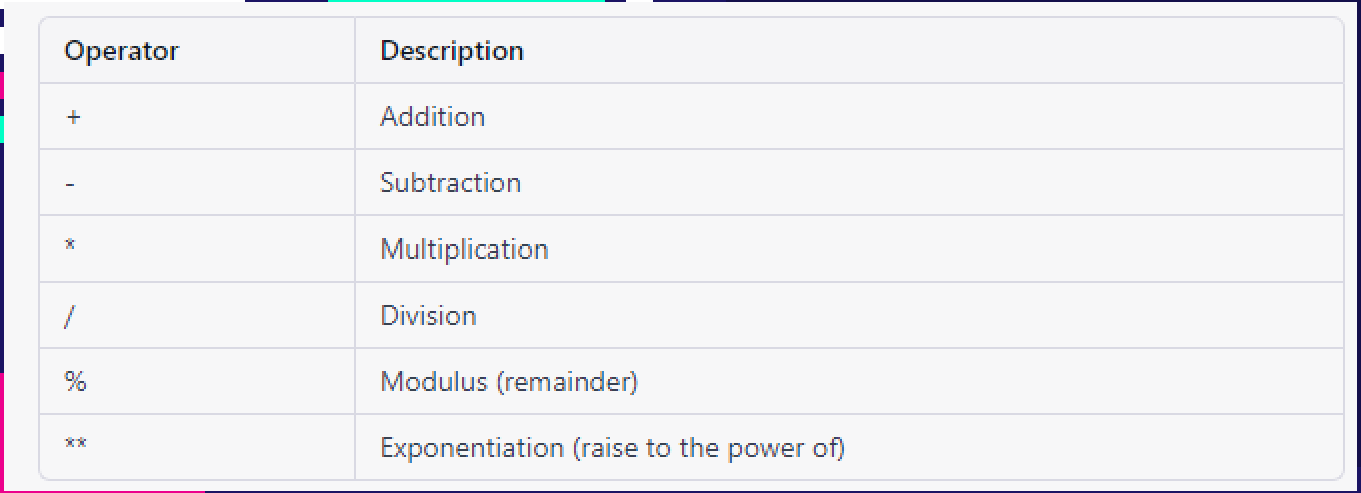
To convert decimal to hexadecimal: `echo "obase=16; decimal_number" | bc`

Hexadecimal: base 16 (digits 0 to 9 and A to F)

To convert hexadecimal to decimal: `echo $((16#hexadecimal_number))`

shell arithmetic expansion converts

bc stands for "basic calculator" and is a command-line calculator that is often used for performing calculations in shell scripts or in the terminal. It can perform basic arithmetic operations, as well as more advanced functions such as trigonometry, logarithms, and exponentiation. It is also capable of working with different numerical bases, such as binary, octal, decimal, and hexadecimal.



Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
**	Exponentiation (raise to the power of)

04

Hands on lab Practice

Have a Question
Leave a Comment!

Subscribe

To stay updated with latest videos

Share

To let the others know more

Thanks