Chapter 2: Configuring the Shell (variables)

Objectives: students should be able to interact with shells and commands using the command line. The objective assumes the Bash shell.

Key Knowledge Areas: Use single shell commands and one line command sequences to perform basic tasks on the command line; Use and edit command history; Invoke commands inside and outside the defined path; Use and modify the shell environment including defining, referencing and exporting environment variables.

Key Terms:

.bash_history
File used to store the current history list when the shell is closed.
env
Print a list of the current Environment variables or change to an alternate environment.
export
Makes an assigned variable available to sub-processes.
history
Print a list of previously executed commands or "re-execute" previously executed commands.
set
Display all variables (local and environment).
unset
Remove one or more variables.

And many more

2. Theory (Configuring the Shell):

2.1 Introduction

One key component of the Bash shell is shell *variables*. These variables are critical because they store vital system information and modify the behavior of the Bash shell, as well as many commands. This chapter will discuss in detail what shell variables are and how they can be used to configure the shell.

You will learn how the `PATH` variable affects how commands are executed and how other variables affect your ability to use the history of your commands. You will also see how to use initialization files to make shell variables *persistent*, so they will be created each time you log into the system.

Additionally, this chapter will cover how to use the command history and the files that are used to store and configure the command history.

2.2 Shell Variables

A *variable* is a name or identifier that can be assigned a value. The shell and other commands read the values of these variables, which can result in altered behavior depending on the contents (value) of the variable.

Variables typically hold a single value, like `0` or `bob`. Some may contain multiple values separated by spaces like `Joe Brown` or by other characters, such as colons; `/usr/bin:/usr/sbin:/bin:/usr/bin:/home/joe/bin`.

You assign a value to a variable by typing the name of the variable immediately followed by the equal sign = character and then the value. For example:

```
name="Bob Smith"
```

Variable names should start with a letter (alpha character) or underscore and contain only letters, numbers and the underscore character. It is important to remember that variable names are case sensitive; a and A are

different variables. Just as with arguments to commands, single or double quotes should be used when special characters are included in the value assigned to the variable to prevent shell expansion.

Valid Variable Assignments	Invalid Variable Assignments
a=1	1=a
_1=a	a-1=3
LONG_VARIABLE='OK'	LONG-VARIABLE='WRONG'
Name='Jose Romero'	'user name'=anything

The Bash shell and many commands make extensive use of variables. One of the main uses of variables is to provide a means to configure various preferences for each user. The Bash shell and commands can behave in different ways, based on the value of variables. For the shell, variables can affect what the prompt displays, the directories the shell will search for commands to execute and much more.

2.2.1 Local and Environment Variables

A *local variable* is only available to the shell in which it was created. An *environment variable* is available to the shell in which it was created, and it is passed into all other commands/programs started by the shell.

To set the value of a variable, use the following assignment expression. If the variable already exists, the value of the variable is modified. If the variable name does not already exist, the shell creates a new local variable and sets the value:

```
variable=value
```

In the example below, a local variable is created, and the `echo` command is used to display its value:

```
1033_clim_antonio@sop.ase.ro:~$ name="judy"
1033_clim_antonio@sop.ase.ro:~$ echo $name
judy
```

By convention, lowercase characters are used to create local variable names, and uppercase characters are used when naming an environment variable. For example, a local variable might be called `test` while an environment variable might be called `TEST`. While this is a convention that most people follow, it is not a rule. An environment variable can be created directly by using the `export` command.

```
export variable=value
```

In the example below, an environment variable is created with the `export` command:

```
1033_clim_antonio@sop.ase.ro:~$ export JOB=engineer
```

Recall that the `echo` command is used to display output in the terminal. To display the value of the variable, use a dollar sign `$` character followed by the variable name as an argument to the `echo` command:

```
1033_clim_antonio@sop.ase.ro:~$ echo $JOB
engineer
```

We can see the difference between local and environment variables by opening a new shell with the `bash` command:

```
1033_clim_antonio@sop.ase.ro:~$ bash
To run a command as administrator (user "root"), use "sudo" command.
See "man sudo_root" for details.
1033_clim_antonio@sop.ase.ro:~$ echo $name
1033_clim_antonio@sop.ase.ro:~$ echo $JOB
engineer
```

Notice that the local variable `name` is empty, while the environment variable `JOB` returns the value engineer. Also notice that if you return to the first shell by using the `exit` command, both variables are still available in the original shell:

```
1033_clim_antonio@sop.ase.ro:~$ exit
exit
1033_clim_antonio@sop.ase.ro:~$ echo $name

judy
1033_clim_antonio@sop.ase.ro:~$ echo $JOB

engineer
```

The local variable *name* is not available in the new shell because by default, when a variable is assigned in the Bash shell, it is initially set as a local variable. When you exit the original shell, only the environment variables will be available. There are several ways that a local variable can be made into an environment variable.

First, an existing local variable can be *exported* with the `export` command.

```
export variable
```

In the example below, the local variable `name` is exported to environment (with the standard convention of all caps):

```
1033_clim_antonio@sop.ase.ro:~$ NAME=judy
1033_clim_antonio@sop.ase.ro:~$ export NAME

1033_clim_antonio@sop.ase.ro:~$ echo $NAME

judy
```

Second, a new variable can be exported and assigned a value with a single command as demonstrated below with the variable `DEPARTMENT`:

```
1033_clim_antonio@sop.ase.ro:~$ export DEPARTMENT=science
1033_clim_antonio@sop.ase.ro:~$ echo $DEPARTMENT
science
```

Third, the `declare` or `typeset` command can be used with the *export* `-x` option to *declare* a variable to be an environment variable. These commands are synonymous and work the same way:

```
1033_clim_antonio@sop.ase.ro:~$ declare -x EDUCATION=masters

1033_clim_antonio@sop.ase.ro:~$ echo $EDUCATION
masters
1033_clim_antonio@sop.ase.ro:~$ typeset -x EDUCATION=masters
1033_clim_antonio@sop.ase.ro:~$ echo $EDUCATION
masters
```

The `env` command is used to run commands in a modified environment. It can also be used to temporarily create or change environment variables that are only passed to a single command execution by using the following syntax:

```
env [NAME=VALUE] [COMMAND]
```

For example, servers are often set to Coordinated Universal Time (UTC), which is good for maintaining consistent time on servers across the planet, but can be frustrating for practical use to simply tell the time:

```
1033_clim_antonio@sop.ase.ro:~$ date
Sun Mar 10 22:47:44 UTC 2020
```

To temporarily set the time zone variable, use the `env` command. The following will run the `date` command with the temporary variable assignment:

```
1033_clim_antonio@sop.ase.ro:~$ env TZ=EST date
Sun Mar 10 17:48:16 EST 2020
```

The `TZ` variable is set only in the environment of the current shell, and only for the duration of the command. The rest of the system will not be affected by this variable. In fact, running the `date` command again will verify that the `TZ` variable has reverted to UTC.

```
1033_clim_antonio@sop.ase.ro:~$ date
Sun Mar 10 22:49:46 UTC 2020
```

2.2.2 Displaying Variables

There are several ways to display the values of variables. The `set` command by itself will display all variables (local and environment) Here, we will pipe the output to the `tail` command so we can see some of the variables that were set in the previous section:

```
1033_clim_antonio@sop.ase.ro:~$ set | tail
DEPARTMENT=science
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
        SHLVL=1
TERM=xterm
UID=1001
USER=1033_clim_antonio
```

```
VISUAL=vi
_=set
name=judy
```

Note

The example above utilizes a *command line pipe*, represented by the `|` character. The pipe character can be used to send the output of one command to another.

The command line pipes will be covered in greater detail later in the course.

Note

The `tail` command is used to display only the last few lines of a file, (or, when used with a pipe, the output of a previous command). By default, the `tail` command displays ten lines of a file provided as an argument.

The `tail` command will be covered in greater detail later in the course.

To display only environment variables, you can use several commands that provide nearly the same output:

```
env
declare -x
typeset -x
export -p
```

```
1033_clim_antonio@sop.ase.ro:~$ env | tail
NAME=judy
MAIL=/var/mail/1033_clim_antonio

SHELL=/bin/bash
TERM=xterm
SHLVL=1
EDUCATION=masters
LOGNAME=1033_clim_antonio

PATH=/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
n:
/bin:/usr/games:/usr/local/games
LESSOPEN=| /usr/bin/lesspipe %s
_=/usr/bin/env
```

To display the value of a specific variable, use the `echo` command with the name of the variable prefixed by the `$` (dollar sign). For example, to display the value of the `PATH` variable, you would execute `echo $PATH`:

```
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n:/usr/games:/usr/local/games
```

Consider This

Variables can also be enclosed in the curly brace `{}` characters in order to delimit them from surrounding text. While the `echo ${PATH}` command would produce the same result as the `echo $PATH` command, the curly braces set the variable apart visually, making it easier to see in scripts in some contexts.

2.2.3 Unsetting Variables

If you create a variable and then no longer want that variable to be defined, use the `unset` command to delete it:

```
unset VARIABLE
```

```
1033_clim_antonio@sop.ase.ro:~$ example=12
1033_clim_antonio@sop.ase.ro:~$ echo $example
12
1033_clim_antonio@sop.ase.ro:~$ unset example
1033_clim_antonio@sop.ase.ro:~$ echo $example
```

Warning
Do not unset critical system variables like the `PATH` variable, as this may lead to a malfunctioning environment.

2.2.4 PATH Variable

The `PATH` variable is one of the most critical environment variables for the shell, so it is important to understand the effect it has on how commands will be executed.

The `PATH` variable contains a list of directories that are used to search for commands entered by the user. When the user types a command and then presses the **Enter** key, the `PATH` directories are searched for an executable file that matches the command name. Processing works through the list of directories from left to right; the first executable file that matches what is typed is the command the shell will try to execute.

Note
Before searching the `PATH` variable for the command, the shell will first determine if the command is an alias or function, which may result in the `PATH` variable not being utilized when that specific command is executed.

Additionally, if the command happens to be built-in to the shell, the `PATH` variable will not be utilized.
Using the `echo` command to display the current `$PATH` will return all the directories that files can be executed from. The following example displays a typical `PATH` variable, with directory names separated from each other by a colon `:` character:

```
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

The following table illustrates the purpose of some of the directories displayed in the output of the previous command:

Directory	Contents
`/home/1033_clim_antonio/bin`	A directory for the current user `1033_clim_antonio` to place programs. Typically used by users who create their own scripts.
`/usr/local/sbin`	Normally empty, but may have administrative commands that have been compiled from local sources.
`/usr/local/bin`	Normally empty, but may have commands that have been compiled from local sources.

Directory	Contents
/usr/sbin	Contains the majority of the administrative command files.
/usr/bin	Contains the majority of the commands that are available for regular users to execute.
/sbin	Contains the essential administrative commands.
/bin	Contains the most fundamental commands that are essential for the operating system to function.

To execute commands that are not contained in the directories that are listed in the PATH variable, several options exist:

- The command may be executed by typing the absolute path to the command.
- The command may be executed with a relative path to the command.
- The PATH variable can be set to include the directory where the command is located.
- The command can be copied to a directory that is listed in the PATH variable.

To demonstrate *absolute paths* and *relative paths* an executable script named my.sh is created in the home directory. Next, that script file is given "execute permissions" (permissions are covered in a later chapter):

```
1033_clim_antonio@sop.ase.ro:~$ echo 'echo Hello World!' > my.sh
1033_clim_antonio@sop.ase.ro:~$ chmod u+x my.sh
```

An *absolute path* specifies the location of a file or directory from the top-level directory through all of the subdirectories to the file or directory. Absolute paths always start with the / character representing the root directory. For example, /usr/bin/ls is an absolute path. It means the ls file, which is in the bin directory, which is in the usr directory, which is in the / (root) directory. A file can be executed using an absolute path like so:

```
1033_clim_antonio@sop.ase.ro:~$ /home/1033_clim_antonio/my.sh
Hello World!
```

A *relative path* specifies the location of a file or directory relative to the current directory. For example, in the /home/1033_clim_antonio directory, a relative path of test/newfile would actually refer to the /home/1033_clim_antonio/test/newfile file. Relative paths never start with the / character.

Note

Visualizing the directory structure using paths can be confusing at first, but it is a necessary skill for effectively navigating the filesystem.

*Absolute paths and relative paths are covered in greater detail in the **NDG Linux Essentials** course.*

Using a relative path to execute a file in the current directory requires the use of the . character, which symbolizes the current directory:

```
1033_clim_antonio@sop.ase.ro:~$ ./my.sh
Hello World!
```

Sometimes a user wants their home directory added to the PATH variable in order to run scripts and programs without using ./ in front of the file name. They might be tempted to modify the PATH variable like so:

```
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
1033_clim_antonio@sop.ase.ro:~$ pwd
/home/1033_clim_antonio
1033_clim_antonio@sop.ase.ro:~$ PATH=/home/1033_clim_antonio
```

Unfortunately, modifying a variable this way *overwrites* the contents. Therefore, everything that was previously contained in the `PATH` variable will be lost.

```
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
/home/1033_clim_antonio
```

Programs listed outside of the `/home/1033_clim_antonio` directory will now only be accessible by using their full path name. For example, assuming that the home directory has not yet been added to the `PATH` variable, the `uname -a` command would behave as expected:

```
1033_clim_antonio@sop.ase.ro:~$ uname -a
Linux localhost 3.15.6+ #2 SMP Wed Jul 23 01:26:02 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

After assigning the home directory to the `PATH` variable, the `uname -a` command will need its full path name to execute successfully:

```
1033_clim_antonio@sop.ase.ro:~$ PATH=/home/1033_clim_antonio
1033_clim_antonio@sop.ase.ro:~$ uname -a
-bash: uname: command not found
1033_clim_antonio@sop.ase.ro:~$ /bin/uname -a
Linux localhost 3.15.6+ #2 SMP Wed Jul 23 01:26:02 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Important

If you changed the `PATH` variable like we did in the previous example and want to reset it, simply use the `exit` command to logout:

```
1033_clim_antonio@sop.ase.ro:~$ exit
```

Once logged back in, the `PATH` variable will be reset to its original value.

It is possible to add a new directory to the `PATH` variable without overwriting its previous contents. Import the current value of the `$PATH` variable into the newly defined `PATH` variable by using it on both sides of the assignment statement:

```
1033_clim_antonio@sop.ase.ro:~$ PATH=$PATH
```

Finish it with the value of the additional home directory path:

```
1033_clim_antonio@sop.ase.ro:~$ PATH=$PATH:/home/1033_clim_antonio
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/home/1033_clim_antonio
```

Now, scripts located in the `/home/1033_clim_antonio` directory can execute without using a path:

```
1033_clim_antonio@sop.ase.ro:~$ my.sh
Hello World!
```

Warning
In general, it is a bad idea to modify the `$PATH` variable. If it were to change, administrators would view it as suspicious activity. Malicious forces want to gain elevated privileges and access to sensitive information residing on Linux servers. One way to do this is to write a script which shares the name of a system command, then change the `PATH` variable to include the administrator's home directory. When the administrator types in the command, it actually runs the malicious script!

2.3 Initialization Files

When a user opens a new shell, either during login or when they run a terminal that starts a shell, the shell is customized by files called *initialization (or configuration) files*. These initialization files set the value of variables, create aliases and functions, and execute other commands that are useful in starting the shell.

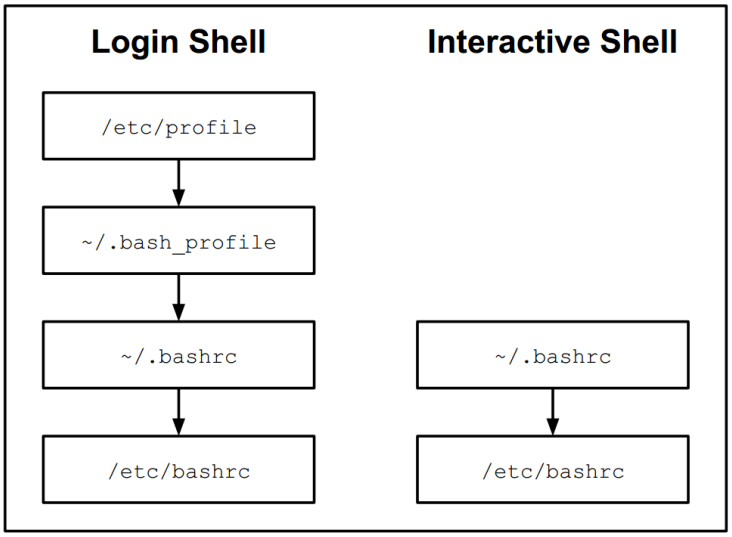
There are two types of initialization files: *global initialization files* that affect all users on the system and *local initialization files* that are specific to an individual user.

The global configuration files are located in the `/etc` directory. Local configuration files are stored in the user's home directory.

BASH Initialization Files

Each shell uses different initialization files. Additionally, most shells execute different initialization files when the shell is started via the login process (called a *login shell*) versus when a shell is started by a terminal (called a non-login shell or an *interactive shell*).

The following diagram illustrates the different files that are started with a typical login shell versus an interactive shell:



Note
The tilde `~` character represents the user's home directory.
Note
File names preceded by a period `.` character indicates hidden files. You can view these files in your home directory by using the `ls` command with the *all* `-a` option.

```
1033_clim_antonio@sop.ase.ro:~$ ls -a
.               .bashrc   .selected_editor  Downloads  Public
```

```
..              .cache Desktop         Music   Templates
.bash_logout   .profile  Documents      Pictures   Videos
```

When Bash is started as a login shell, the `/etc/profile` file is executed first. This file typically executes all files ending in `.sh` that are found in the `/etc/profile.d` directory.

The next file that is executed is usually `~/.bash_profile` (but some users may use `~/.bash_login` or `~/.profile` file). The `~/.bash_profile` file typically also executes the `~/.bashrc` file which in turn executes the `/etc/bashrc` file.

Consider This

Since the `~/.bash_profile` file is under the control of the user, the execution of the `~/.bashrc` and `/etc/bashrc` files are also controllable by the user.

When Bash is started as an interactive shell, it executes the `~/.bashrc` file, which may also execute the `/etc/bashrc` file, if it exists. Again, since the `~/.bashrc` file is owned by the user who is logging in, the user can prevent execution of the `/etc/bashrc` file.

With so many initialization files, a common question at this point is "what file am I supposed to use?" The following chart illustrates the purpose of each of these files, providing examples of what commands you might place in each file:

File	Purpose
`/etc/profile`	This file can only be modified by the administrator and will be executed by every user who logs in. Administrators use this file to create key environment variables, display messages to users as they log in, and set key system values.

`~/.bash_profile`
`~/.bash_login`
`~/.profile`
	Each user has their own `.bash_profile` file in their home directory. The purpose of this file is the same as the `/etc/profile` file, but having this file allows a user to customize the shell to their own tastes. This file is typically used to create customized environment variables.

`~/.bashrc`	Each user has their own `.bashrc` file in their home directory. The purpose of this file is to generate items that need to be created for each shell, such as local variables and aliases.
`/etc/bashrc`	This file may affect every user on the system. Only the administrator can modify this file. Like the `.bashrc` file, the purpose of this file is to generate items that need to be created for each shell, such as local variables and aliases.

2.3.1 Modifying Initialization Files

The way a user's shell operates can be changed by modifying that user's initialization files. Modifying global configuration requires administrative access to the system as the files under the `/etc` directory can only be modified by an administrator. A user can only modify the initialization files in their home directory.

The best practice is to create a backup before modifying configuration files as insurance against errors; a backup can always be restored if something should go wrong. The following example references CentOS, a community-supported Linux distribution based on source code from Red Hat Enterprise Linux.

In some distributions, the default `~/.bash_profile` file contains the following two lines which customize the `PATH` environment variable:

```
PATH=$PATH:$HOME/bin
export PATH
```

The first line sets the `PATH` variable to the existing value of the `PATH` variable with the addition of the `bin` subdirectory of the user's home directory. The `$HOME` variable refers to the user's home directory. For example, if the user logging in is `joe`, then `$HOME/bin` is `/home/joe/bin`.

The second line converts the local `PATH` variable into an environment variable.

The default `~/.bashrc` file executes `/etc/bashrc` using a statement like:

```
./etc/bashrc
```

The period `.` character used to *source* a file in order to execute it. The period `.` character also does have a synonym command, the `source` command, but the use of the period `.` character is more common than using the `source` command.

Sourcing can be an effective way to test changes made to initialization files, enabling you to correct any errors without having to log out and log in again. If you update the `~/.bash_profile` to alter the `PATH` variable, you could verify that your changes are correct by executing the following:

```
. ~/.bash_profile
echo $PATH
```

Warning

Our virtual environment is not persistent, therefore, any changes made to the configuration files will not be saved if the VM is reset.

2.3.2 BASH Exit Scripts

Just as Bash executes one or more files upon starting up, it also may execute one or more files upon exiting. As Bash exits, it will execute the `~/.bash_logout` and `/etc/bash_logout` files, if they exist. Typically, these files are used for "cleaning up" tactics as the user exits the shell. For example, the default `~/.bash_logout` executes the `clear` command to remove any text present in the terminal screen.

Consider This

When a new user is created, the files from the `/etc/skel` directory are automatically copied into the new user's home directory. As an administrator, you can modify the files in the`/etc/skel` directory to provide custom features to new users.

2.4 Command History

In a sense, the `~/.bash_history` file could also be considered to be an initialization file, since Bash also reads this file as it starts up. By default, this file contains a history of the commands that a user has executed within the Bash shell. When a user exits the Bash shell, it writes out the recent history to this file.

There are several ways that this command history is advantageous to the user:

- You can use the **Up** ↑ and **Down** ↓ **Arrow Keys** to review your history and select a previous command to execute again.
- You can select a previous command and modify it before executing it.
- You can do a reverse search through history to find a previous command to select, edit, and execute it. To start the search, press **Ctrl+R** and then begin typing a portion of a previous command.
- You execute a command again, based upon a number that is associated with the command.

2.4.1 Executing Previous Commands

The Bash shell will allow a user to use the **Up ↑ Arrow Key** to view previous commands. With each press of the up arrow, the shell displays one more command back in the history list.

If the user goes back too far, the **Down ↓ Arrow Key** can be used to return to a more recent command. Once the correct command is displayed, the **Enter** key can be pressed to execute it.

The **Left ←** and **Right → Arrow Keys** can also be used to position the cursor within the command.
The **Backspace** and **Delete** keys are used to remove text, and additional characters can be typed into the command line to modify it before pressing the **Enter** key to execute it.

There are more keys that can be used for editing a command on the command line. The following table summarizes some helpful editing keys:

Action	Key	Alternate Key Combination
Previous history item	↑	**Ctrl+P**
Next history item	↓	**Ctrl+N**
Reverse history search		**Ctrl+R**
Beginning of line	**Home**	**Ctrl+A**
End of line	**End**	**Ctrl+E**
Delete current character	**Delete**	**Ctrl+D**
Delete to the left of the cursor	**Backspace**	**Ctrl+X**
Move cursor left	←	**Ctrl+B**
Move cursor right	→	**Ctrl+F**

2.4.2 Changing Editing Keys

The keys that are available for editing a command are determined by the settings of a *library* called **Readline**. The keys are typically set to match the key assignments found within the `emacs` text editor (a popular Linux editor) by default.

To *bind* the keys to match the key assignments found with another popular text editor, the `vi` editor. The shell can be configured with the `set -o vi` command. To set the key bindings back to the `emacs` text editor, use the `set -o emacs` command.

Note

If you don't know how to use the `vi` editor yet, you may want to stick with the `emacs` keys as the `vi` editor has a bigger learning curve.

The `vi` editor will be covered in greater detail later in the course. Usually, tasks can be accomplished faster using nano editor.

To automatically configure the edit history options at login, edit the `~/.inputrc` file. If this file doesn't exist, the `/etc/inputrc` file is used instead. Key bindings are set differently in configuration files than on the command line; for example, to enable the `vi` key binding mode for an individual, add the following lines to the `~/.inputrc` file:

```
set editing-mode vi
set keymap vi
```

If the situation is reversed, perhaps due to the above two lines being added to the `/etc/inputrc` file, a user can enable the `emacs` mode by adding the following lines to the `~/.inputrc` file:

```
set editing-mode emacs
set keymap emacs
```

2.4.3 Using the history Command

The `history` command can be used to re-execute previously executed commands. When used with no arguments, the `history` command provides a list of previously executed commands:

```
1033_clim_antonio@sop.ase.ro:~$ history
        1  ls
        2  cd test
        3  cat alpha.txt
        4  ls -l
        5  cd ..
        6  ls
        7  history
```

Note that each command is assigned a number that a user can use to re-execute the command.

The `history` command has numerous options; the most common of these options are listed below:

Option	Purpose
`-c`	Clear the list
`-r`	Read the history file and replace the current history
`-w`	Write the current history list to the history file

As the `history` list commonly contains five hundred or more commands, it is often helpful to filter the list. The `history` command accepts a number as an argument to indicate how many commands to list. For example, executing the following command will only show the last three commands from your history.

```
1033_clim_antonio@sop.ase.ro:~$ history 3
        5  cd ..
        6  ls
        7  history
```

Note

The `grep` command is very useful for filtering the output of commands that produce copious output. For example, to view all of the commands in your history that contain the ls command, search for the `ls` pattern using the following command:

```
1033_clim_antonio@sop.ase.ro:~$ history | grep "ls"
    1  ls
    4  ls -l
    6  ls
```

```
    9  history | grep "ls"
```

The `grep` command (as weel as `sed`) will be covered in greater detail later in the course and are subjects for final exam!).

2.4.4 Configuring the history Command

When you close the shell program, it takes commands in the history list and stores them in the `~/.bash_history` file, also called the *history file*. By default, five hundred commands will be stored in the history file. The `HISTFILESIZE` variable will determine how many commands to write to this file.

If a user wants to store the history commands in a file that is different from `~/.bash_history`, then the user can specify an absolute path to the different file as the value for the `HISTFILE` local variable:

```
HISTFILE=/path/to/file
```

The `HISTSIZE` variable will determine how many commands to keep in memory for each Bash shell. If the size of `HISTSIZE` is greater than the size of `HISTFILESIZE`, then only the most recent number of commands specified by `HISTFILESIZE` will be written to the history file when the Bash shell exits.

Although it is not normally set to anything by default, you may want to take advantage of setting a value for the `HISTCONTROL` variable in an initialization file like the `~/.bash_profile` file. The `HISTCONTROL` variable could be set to any of the following features:

```
HISTCONTROL=ignoredups
```

Prevents duplicate commands that are executed consecutively.

```
HISTCONTROL=ignorespace
```

Any command that begins with a space will not be stored. This provides the user with an easy way to execute a command that won't go into the history list.

```
HISTCONTROL=ignoreboth
```

Consecutive duplicates *and* any commands that begin with a space will not be stored.

```
HISTCONTROL=erasedups
```

Commands that are identical to another command in your history will not be stored. (The previous entry of the command will be deleted from the history list.)

```
HISTCONTROL=ignorespace:erasedups
```

Includes the benefit of `erasedups` with the advantage of `ignorespace`.

Another variable that will affect what gets stored in the history of commands is the `HISTIGNORE` variable. Most users do not want the history list cluttered with basic commands like the `ls`, `cd`, `exit`, and `history` commands. The `HISTIGNORE` variable can be used to tell Bash not to store certain commands in the history list.

To have commands not included in the history list, include an assignment like the following in the `~/.bash_profile` file:

```
HISTIGNORE='ls*:cd*:history*:exit'
```

Note

The `*` character is used to indicate *anything else after the command*. So, `ls*` would match any `ls` command, such as `ls -l` or `ls` /etc.

***Globbing** will be covered in greater detail later in the course.*

2.4.5 Executing Previous Commands

The ! exclamation mark is a special character to the Bash shell to indicate the execution of a command within the history list. There are many ways to use the ! exclamation character to re-execute commands; for example, executing two exclamation characters will repeat the previous command:

```
1033_clim_antonio@sop.ase.ro:~$ ls

Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
1033_clim_antonio@sop.ase.ro:~$ !!

ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

The following table provides some examples of using the exclamation ! character:

History Command	Meaning
`!!`	Repeat the last command
`!-4`	Execute the command that was run four commands ago
`!555`	Execute command number `555`
`!ec`	Execute the last command that started with `ec`
`!?joe`	Execute the last command that contained `joe`

Another way to use the history list is to take advantage of the fact that the last argument of every command is stored. Often a user will type a command to refer to a file, perhaps to display some information about that file. Subsequently, the user may want to type another command to do something else with the same file. Instead of having to type the path again, a user can use a couple of keyboard shortcuts to recall that file path from the previous command line.

By pressing either **Esc+. (Escape+Period)** or **Alt+. (Alt+Period)**, the shell will bring back the last argument of the previous command. Repeatedly pressing one of these key combinations will recall, in reverse order, the last argument of each previous command.

2. Labs - Configuring the Shell - (steps from 1 to 26):

2.1 Introduction

One key component of the Bash shell is shell *variables*. These variables are critical because they store vital system information and modify the behavior of the Bash shell, as well as many commands. In this lab, you will learn how to configure shell features such as variables and command history. Understanding these shell features will make it much easier for you to work in the command line environment.

Step 1

A *local variable* is only available to the shell in which it was created. To set the value of a variable, use the following assignment expression:

```
variable=value
```

Create and display a local variable by executing the following commands:

```
name="bob"
echo $name
```

```
1033_clim_antonio@sop.ase.ro:~$ name="bob"
1033_clim_antonio@sop.ase.ro:~$ echo $name
bob
```

Step 2

An *environment variable* is available to the shell in which it was created, and it is passed into all other commands/programs started by the shell. An environment variable can be created directly by using the `export` command:

```
export variable=value
```

Create and display an environment variable by executing the following commands:

```
export AGE=25
echo $AGE
```

```
1033_clim_antonio@sop.ase.ro:~$ export AGE=25
1033_clim_antonio@sop.ase.ro:~$ echo $AGE
25
```

Step 3

To display the value of the variable, use a dollar sign `$` character followed by the variable name as an argument to the `echo` command. Recall that the `echo` command is used to display output in the terminal.

Open another shell and view the contents of the two variables that were created:

```
bash
echo $name
echo $AGE
```

```
1033_clim_antonio@sop.ase.ro:~$ bash
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.
1033_clim_antonio@sop.ase.ro:~$ echo $name
1033_clim_antonio@sop.ase.ro:~$ echo $AGE
25
```

Notice in the output of the commands above that the `$name` variable didn't exist in the new shell because local variables are not passed into new processes. However, the `$AGE` variable did exist in the new shell, since environment variables *are* passed into new processes.

Step 4

The `set` command by itself will display all variables (local and environment). To display variables in the original shell, exit the current shell and use the `set` command with a *pager*, along with the `less` command, in a *command pipeline* by using the pipe `|` character:

```
exit
set | less
```

```
1033_clim_antonio@sop.ase.ro:~$ exit
1033_clim_antonio@sop.ase.ro:~$ set | less
AGE=25
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote
:force_fignore:histappend:interactive_comments:login_shell:progcomp:promptvars:s
ourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_VERSINFO=([0]="2" [1]="8")
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="4" [2]="19" [3]="1" [4]="release" [5]="x86_64-pc-lin
ux-gnu")
BASH_VERSION='4.4.19(1)-release'
COLUMNS=80
DIRSTACK=()
EDITOR=vi
EUID=1001
GROUPS=()
HISTCONTROL=ignoreboth
HISTFILE=/home/1033_clim_antonio/.bash_history
HISTFILESIZE=2000
:
```

To exit the `less` command and return to the prompt, press **Q**.

Step 5

The `set` command used in the previous step displays all variables, both local and environment. To only display environment variables, execute the `env` command:

```
env
```

```
1033_clim_antonio@sop.ase.ro:~$ env


LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;
01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.t
ar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=
01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;3
1:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=0
1;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;3
1:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*
.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm
=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gi
f=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=0
1;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01
;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;
35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:
*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.g
l=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01
;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;3
6:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.
opus=00;36:*.spx=00;36:*.xspf=00;36:


LESSCLOSE=/usr/bin/lesspipe %s %s

AGE=25

HUSHLOGIN=FALSE

USER=1033_clim_antonio

PWD=/home/1033_clim_antonio

HOME=/home/1033_clim_antonio

MAIL=/var/mail/1033_clim_antonio

SHELL=/bin/bash

TERM=xterm

SHLVL=1

LOGNAME=1033_clim_antonio

PATH=/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
n:/bin:/usr/games:/usr/local/games

LESSOPEN=| /usr/bin/lesspipe %s

 _=/usr/bin/env

1033_clim_antonio@sop.ase.ro:~$
```

Note

The following commands will also display only environment variables and will provide similar output to the `env` command:

```
declare -x

typeset -x

export -p
```

Step 6

If it is no longer necessary for a variable to be defined, the `unset` command can be used to delete it:

```
unset VARIABLE
```

Execute the following commands to unset these variables and then verify that they are no longer set:

```
unset name
unset AGE
echo $name
echo $AGE
```

```
1033_clim_antonio@sop.ase.ro:~$ unset name
1033_clim_antonio@sop.ase.ro:~$ unset AGE
1033_clim_antonio@sop.ase.ro:~$ echo $name
1033_clim_antonio@sop.ase.ro:~$ echo $AGE
```

Important

Do not unset critical system variables like the `PATH` variable, as this may lead to a malfunctioning environment.

Step 7

The `PATH` variable contains a list of directories that are used to search for commands entered by the user. When the user types a command and then presses the **Enter** key, the `PATH` directories are searched for an executable file that matches the command name.

Display the value of the `PATH` variable by executing the following command:

```
echo $PATH
```

```
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

The following table illustrates the purpose of some of the directories displayed in the output of the previous command:

Directory	Contents
`/home/1033_clim_antonio/bin`	A directory for the current `1033_clim_antonio` user to place programs. Typica used by users who create their own scripts.
`/usr/local/sbin`	Normally empty, but may have administrative commands that have been compile from local sources.
`/usr/local/bin`	Normally empty, but may have commands that have been compiled from local sources.
`/usr/sbin`	Contains the majority of the administrative command files.
`/usr/bin`	Contains the majority of the commands that are available for regular users to execute.

Directory	Contents
/sbin	Contains the essential administrative commands.
/bin	Contains the most fundamental commands that are essential for the operating system to function.

Note

Recall that when you execute a command without specifying a complete path name, the shell uses the value of the PATH variable to determine where the command is stored.

Step 8

Using the `which` command, you can determine where a command resides in the PATH. Execute the following to determine where the `ls` command resides:

```
which ls
```

```
1033_clim_antonio@sop.ase.ro:~$ which ls
/bin/ls
```

Step 9

If the command doesn't reside in a directory that is specified in the PATH variable, no output is displayed:

```
which zzz
```

```
1033_clim_antonio@sop.ase.ro:~$  which zzz
1033_clim_antonio@sop.ase.ro:~$
```

Consider This

You can also use the `type` command, which more clearly indicates that the command is not located in the PATH:

```
type zzz
```

```
1033_clim_antonio@sop.ase.ro:~$  type zzz
-bash: type: zzz: not found
```

Step 10

Recall that when the user types a command and then presses the **Enter** key, the PATH directories are searched for an executable file that matches the command name. However, it is possible to execute commands that are not contained in the directories that are listed in the PATH variable using the following options:

- The command may be executed by typing the *absolute path* to the command.
- The command may be executed with a *relative path* to the command.
- The PATH variable can be set to include the directory where the command is located.
- The command can be copied to a directory that is listed in the PATH variable.

There are two types of paths commonly used when invoking a command or locating a file in the Linux filesystem: absolute paths and relative paths.

An absolute path specifies the location of a file or directory from the top-level directory through all of the subdirectories to the file or directory. Absolute paths always start with the `/` character representing the root directory. A relative path specifies the location of a file or directory relative to the current directory.

Note

If a command or file seems to be missing, often the reason is that it is not located within directories in the `PATH` variable, and the user is trying to access it with a relative path. While specifying an absolute path should remedy the problem, it can be cumbersome, especially when a file is located several layers deep into a directory structure.

A relative path is used for files that are within the directory structure you are currently located in. For example, users normally log in to their home directories, so changing to the `Documents` directory (located under the user's home directory) using the relative path is as simple as typing the following command:

```
cd Documents
```

```
1033_clim_antonio@sop.ase.ro:~$ cd Documents
1033_clim_antonio@sop.ase.ro:~/Documents$
```

If, however, you were in the `/etc` directory and you want to change to the `Documents` directory, just typing the command below from the `/etc` directory will return an error, because the `Documents` directory is not located within the `/etc` directory:

```
cd /etc
cd Documents
```

```
1033_clim_antonio@sop.ase.ro:~/Documents$ cd /etc
1033_clim_antonio@sop.ase.ro:/etc$ cd Documents
-bash: cd: Documents: No such file or directory
```

Instead, you would need to use the absolute path by typing the following command:

```
cd /home/1033_clim_antonio/Documents
```

```
1033_clim_antonio@sop.ase.ro:/etc$ cd /home/1033_clim_antonio/Documents
1033_clim_antonio@sop.ase.ro:~/Documents$
```

Step 11

In this step, you will set the `PATH` variable to include a directory where a command would be located. Make a directory and then add it to the `PATH` variable by executing the following commands:

```
mkdir mybin
PATH=$PATH:/home/1033_clim_antonio/mybin
```

```
1033_clim_antonio@sop.ase.ro:~/Documents$ cd
1033_clim_antonio@sop.ase.ro:~$ mkdir mybin
1033_clim_antonio@sop.ase.ro:~$ PATH=$PATH:/home/1033_clim_antonio/mybin
```

Verify that the `PATH` now includes the `/home/1033_clim_antonio/mybin` directory using the `echo` command:

```
echo $PATH
```

```
1033_clim_antonio@sop.ase.ro:~$ echo $PATH
```

```
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n:/usr/games:/usr/local/games:/home/1033_clim_antonio/mybin
```

Notice in the output above that the `/home/1033_clim_antonio/mybin` directory is appended to the `PATH` variable:

```
/home/1033_clim_antonio/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n:/usr/games:/usr/local/games: /home/1033_clim_antonio/mybin
```

Step 12

In the following steps, you will copy a command (executable program) to a directory that is listed in the `PATH` variable. Create a simple Bash shell program in your current directory by executing the following commands:

```
echo "echo hello" > hello.sh
chmod a+x hello.sh
```

```
1033_clim_antonio@sop.ase.ro:~$ echo "echo hello" > hello.sh
1033_clim_antonio@sop.ase.ro:~$ chmod a+x hello.sh
```

Note

The second command in the example above gives the `hello.sh` script file *execute* permissions, which allows for a file to be run as a process.

Permissions will be covered in a later chapter (ref. chmod in CLI + octal & trough WinSCP properties).

Step 13

Execute the following command to verify that the `hello.sh` program works correctly:

```
/home/1033_clim_antonio/hello.sh
```

```
1033_clim_antonio@sop.ase.ro:~$ /home/1033_clim_antonio/hello.sh
hello
```

Step 14

Note that you can't just execute `hello.sh`, you must include the path to the command. Execute the following to verify:

```
hello.sh
```

```
1033_clim_antonio@sop.ase.ro:~$ hello.sh
hello.sh: command not found
```

Step 15

However, if you place this script in the `/home/1033_clim_antonio/mybin` directory, which is included in the `PATH` variable, you can execute the command without specifying the full path. Use the `mv` command to move the `hello.sh` file to the `/home/1033_clim_antonio/mybin` directory, which is located in the `PATH` and then execute the `hello.sh` file alone to verify:

```
mv hello.sh /home/1033_clim_antonio/mybin
hello.sh
```

```
1033_clim_antonio@sop.ase.ro:~$ mv hello.sh /home/1033_clim_antonio/mybin
1033_clim_antonio@sop.ase.ro:~$ hello.sh
hello
```

Note

You could also verify this by executing the following command:

```
which hello.sh
```

```
1033_clim_antonio@sop.ase.ro:~$ which hello.sh
/home/1033_clim_antonio/mybin/hello.sh
```

Step 16

When a user opens a new shell, either during login or when they run a terminal that starts a shell, the shell is customized by files called *initialization (or configuration) files*. These initialization files set the value of variables, create aliases and functions, and execute other commands that are useful in starting the shell.

Each shell uses different initialization files. Additionally, most shells execute different initialization files when the shell is started via the login process (called a *login shell*) versus when a shell is started by a terminal (called a non-login shell or an *interactive shell*).

When Bash is started as an interactive shell, it executes the `~/.bashrc` file.

Use the `less` pager command to view your account's `.bashrc` file to see what customizations are set by this file:

```
less .bashrc
```

```
1033_clim_antonio@sop.ase.ro:~$ less .bashrc
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend
```

```
# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
```

To exit the `less` command and return to the prompt, press **Q**.

Step 17

Adding custom paths to the `PATH` environment variable provides users with a means to customize their shell. For example, if you wanted to create a custom shell script that could run from a directory other than where it was located, you could add the path to it by modifying the `PATH` environment variable. The `grep` command can be used to specifically see any changes made to the `PATH` variable in the `.bashrc` file:

```
grep PATH .bashrc
```

```
1033_clim_antonio@sop.ase.ro:~$ grep PATH .bashrc
PATH="$HOME/bin:$PATH"
```

Note

The `grep` command provides the ability to filter text and will only display lines that contain the argument passed to the command, in this case, `PATH`.

The `grep` command will be covered in greater detail later in the course.

Step 18

If you wanted to add more to the `PATH` variable, you could either edit the `.bashrc` file directly with a text editor (such as `vi`, `nano`, or others covered in a later chapter) or execute the following commands:

```
echo 'PATH=$PATH:/home/1033_clim_antonio/mybin' >> .bashrc
tail .bashrc
```

```
1033_clim_antonio@sop.ase.ro:~$ echo 'PATH=$PATH:/home/1033_clim_antonio/mybin' >> .bas
hrc
1033_clim_antonio@sop.ase.ro:~$ tail .bashrc
# add user bin to path
PATH="$HOME/bin:$PATH"


# set editor and visual envs to vim
EDITOR=vi
VISUAL=vi


# change to home dir
cd ~
PATH=$PATH:/home/1033_clim_antonio/mybin
```

In the example above, the >> characters are used to redirect the output of the `echo` command and append it to the bottom of the `.bashrc` file. The `tail` command displays the last ten lines of the `.bashrc` file so you can verify the file was successfully edited.

Note

Redirection and the `tail` command will be covered in greater detail later in the course.

Step 19

The `~/.bash_history` file could also be considered to be an initialization file since Bash also reads this file as it starts up. By default, this file contains a history of the commands that a user has executed within the Bash shell.

The command history will allow a user to use the **Up ↑ Arrow Key** to view previous commands. With each press of the up arrow, the shell displays one more command back in the history list.

If the user goes back too far, the **Down ↓ Arrow Key** can be used to return to a more recent command. Once the correct command is displayed, the **Enter** key can be pressed to execute it.

The **Left ← and Right → Arrow Keys** can also be used to position the cursor within the command.
The **Backspace** and **Delete** keys are used to remove text, and additional characters can be typed into the command line to modify it before pressing the **Enter** key to execute it. The following steps will demonstrate how to use the command history.

Execute the following command, which will result in an error message:

```
cp /etc/host .
```

```
1033_clim_antonio@sop.ase.ro:~$ cp /etc/host .
cp: cannot stat '/etc/host': No such file or directory
```

Step 20

The file name should be `hosts`, not `host`. To fix this, press the **Up Arrow ↑ Key** one time, use your **Left Arrow ← Keys** to move over to the space after `host`, type an `s`, and then press the **Enter** key:

```
Up Arrow Key
Left Arrow Key
Left Arrow Key
s
Enter Key
```

```
1033_clim_antonio@sop.ase.ro:~$ cp /etc/hosts .
```

Step 21

Execute the `history` command to view previously executed commands:

```
history
```

```
1033_clim_antonio@sop.ase.ro:~$ history
    1  bash
    2  name="bob"
    3  echo $name
```

```
    4  export AGE=25
    5  echo $AGE
    6  bash
    7  set
    8  bash
    9  env
   10  unset name
   11  unset AGE
   12  echo $name
   13  echo $AGE
   14  echo $PATH
   15  which ls
   16  which zzz
   17  type zzz
   18  mkdir mybin
   19  PATH=$PATH:/home/1033_clim_antonio/mybin
   20  echo "echo hello" > hello.sh
   21  chmod a+x hello.sh
   22  /home/1033_clim_antonio/hello.sh
   23  hello.sh
   24  mv hello.sh /home/1033_clim_antonio/mybin
   25  hello.sh
   26  which hello.sh
   27  more .bashrc
   28  grep PATH .bashrc
   29  echo 'PATH=$PATH:/home/1033_clim_antonio/mybin' >> .bashrc
   30  tail .bashrc
   31  cp /etc/host .
   32  cp /etc/hosts .
   33  history
```

Step 22

The ! exclamation mark is a special character to the Bash shell to indicate the execution of a command within the history list. There are many ways to use the ! exclamation character to re-execute commands; for example, executing two exclamation characters will repeat the previous command.

Execute the `history` command again by executing the command history shortcut `!!`:

```
!!
```

```
1033_clim_antonio@sop.ase.ro:~$ !!
```

```
 1  bash
 2  name="bob"
 3  echo $name
 4  export AGE=25
 5  echo $AGE
 6  bash
 7  set
 8  bash
 9  env
10  unset name
11  unset AGE
12  echo $name
13  echo $AGE
14  echo $PATH
15  which ls
16  which zzz
17  type zzz
18  mkdir mybin
19  PATH=$PATH:/home/1033_clim_antonio/mybin
20  echo "echo hello" > hello.sh
21  chmod a+x hello.sh
22  /home/1033_clim_antonio/hello.sh
23  hello.sh
24  mv hello.sh /home/1033_clim_antonio/mybin
25  hello.sh
26  which hello.sh
27  more .bashrc
28  grep PATH .bashrc
29  echo 'PATH=$PATH:/home/1033_clim_antonio/mybin' >> .bashrc
30  tail .bashrc
31  cp /etc/host .
32  cp /etc/hosts .
33  history
```

Step 23

Execute the previous `tail` command by typing an exclamation `!` character followed by the number before the `tail` command in the history list. For example, given the previous output, you would execute the following:

The number in the command history in our virtual environment may be different from the example.

```
!15
```

```
1033_clim_antonio@sop.ase.ro:~$  !15
which ls
/bin/ls
```

Step 24

Execute the following commands to have the history feature ignore duplicate commands:

```
HISTCONTROL=ignoredups
ls
ls
history | tail
```

```
1033_clim_antonio@sop.ase.ro:~$ HISTCONTROL=ignoredups
1033_clim_antonio@sop.ase.ro:~$ ls
Desktop    Downloads  Pictures  Templates  hosts
Documents  Music      Public    Videos     mybin
1033_clim_antonio@sop.ase.ro:~$ ls
Desktop    Downloads  Pictures  Templates  hosts
Documents  Music      Public    Videos     mybin
1033_clim_antonio@sop.ase.ro:~$ history | tail
   37  history
   38  bash
   39  env
   40  history
   41  echo $AGE
   42  history
   43  which ls
   44  HISTCONTROL=ignoredups
   45  ls
   46  history | tail
```

Notice that only one of the `ls` commands was recorded. The `history | tail` command line takes the output of the `history` command and only displays the last ten lines of that output.

Step 25

To prevent the history feature from recording commands that begin with a space character, execute the following commands. (Note that the cal command used in the example below has a space in front of it.):

```
HISTCONTROL=ignorespace
date
```

```
 cal
history | tail
```

```
1033_clim_antonio@sop.ase.ro:~$ HISTCONTROL=ignorespace
1033_clim_antonio@sop.ase.ro:~$ date
Tue Apr  9 15:26:35 UTC 2019
1033_clim_antonio@sop.ase.ro:~$  cal
     April 2019
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8   10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

1033_clim_antonio@sop.ase.ro:~$ history | tail
   40  history
   41  echo $AGE
   42  history
   43  which ls
   44  HISTCONTROL=ignoredups
   45  ls
   46  history | tail
   47  HISTCONTROL=ignorespace
   48  date
   49  history | tail
```

Step 26

To clear the history list, use the `-c` option with the `history` command:

```
history -c
```

```
1033_clim_antonio@sop.ase.ro:~$ history -c
1033_clim_antonio@sop.ase.ro:~$ history
    1  history
```

Now: ONLINE TEST from this material