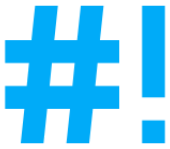
Linux Commands and Bash Scripting



This is a hands-on training on Linux commands and Bash scripting, with lots of step-by-step examples, and with concise (not very detailed) explanations. To get the maximum benefit from it, you should try these commands and examples yourself, rather than reading or just skimming through.

> **INFO**
>
> These examples are mostly based on the wonderful book The Linux Command Line by William Shotts. So, for more detailed and complete explanations, and for a deeper understanding, I recommend downloading and reading the PDF version of this book. I cannot recommend it highly enough.

You need a Linux terminal to try these examples. I would recommend using a virtual machine or a container with Ubuntu or Debian.

> **CAUTION**
>
> If you already use Linux on your personal machine (laptop), it is still **NOT recommended** to try the examples directly on it. They are are not harmful, but a mistyping or some other mistake might have unexpected results.

> **TIP**
>
> There are lots of ways for running a virtual machine. Using a podman container is one of the simplest ones.

> **INFO**
>
> These lessons have been used previously on this online course. The video recordings for the scripting part are published on YouTube:

> **INFO**

There is also an italian version of these lessons, translated by Claudio Cavalli.

INFO

A PDF version of these lessons is also available for download.

Lesson 1: Intro

In this lesson we will get started with some simple commands and their options, will learn how to navigate and explore the system, etc.:

- Trying some simple commands (`date`, `cal`, `df`, `free`, `exit`).
- Navigation (`pwd`, `ls`, `cd`).
- Relative and absolute filenames.
- Exploring the system (`ls`, `file`, `less`).
- Long directory listing.
- Directory structure on Linux systems.

DOWNLOAD INTRO.CAST

DOWNLOAD LESSON01/INTRO.CAST

1. First commands

1. Display the current date and time:

```
date
```

```
date +%Y-%m-%d
```

2. Display a calendar of the current month:

```
cal
```

   For another month:

```
cal 5 2020
```

3. Check how much free space there is on the disk drives:

```
df
```

```
df -h
```

```
df -h /
```

4. Display the amount of the free memory:

```
free
```

```
free -h
```

DOWNLOAD LESSON01/PART1.CAST

2. Navigation

The command `cd` (change directory) is used to move from one directory to another. The command `pwd` (print working directory) shows the current location. The command `ls` (list) shows the content of the current working directory.

1. Display the current working directory with `pwd` (print working directory):

```
pwd
```

2. List the contents of a directory:

```
ls /
```

```
ls /usr
```

```
ls -l /usr
```

3. Change the current working directory:

```
cd /usr
```

```
pwd
```

```
ls
```

```
cd /usr/bin
```

```
pwd
```

The path `/usr/bin` is called *absolute*, since it shows the full path, starting from the *root* (`/`).

4. Go to the directory one level up:

```
cd ..
```

```
pwd
```

Two dots (`..`) represent the *parent* of the current directory.

By the way, a single dot (`.`) represents the current directory:

```
cd .
```

```
pwd
```

5. Use a relative path:

```
cd bin
```

```
pwd
```

The directory `bin` is relative to the current one (in this case `/usr`).

6. Go to the previous current directory:

```
cd /var/log
```

```
cd -
```

```
cd -
```

7. Go to the home directory:

```
cd
```

```
cd ~
```

The tilde (`~`) represents the home directory of the current user.

DOWNLOAD LESSON01/PART2.CAST

3. Command options

Let's see some options of the command `ls`.

1. List only some files:

   ```
   ls /bin
   ```

   ```
   ls /bin/b* /bin/c*
   ```

   We are listing only those files that start with `b` and those that start with `c`, on the directory `/bin`.

2. Long listing:

   ```
   ls -l /bin/b* /bin/c*
   ```

   The option `-l` stands for *long* listing, where each file is printed on its own line, with more details.

3. Long and short options.

   Notice that the middle column shows the size of the file (in bytes). To make the size more readable we can use the option `--human-readable`:

   ```
   ls -l --human-readable /bin/b* /bin/c*
   ```

   Instead of this long option we can use its short equivalent `-h`, which is more convenient to write:

   ```
   
   ```

```
ls -l -h /bin/b* /bin/c*
```

Tip: In order to modify the previous command, you can use the *up-arrow* key on the keyboard to display the previous command, use *left-arrow* and *right-arrow* keys to locate the cursor, modify the command, and then press [Enter].

4. Merging short options.

   We can also merge the short options like this:

   ```
   ls -lh /bin/b* /bin/c*
   ```

   By default files are listed alphabetically, but we can sort them by modification time, using the option `-t`:

   ```
   ls -lht /bin/b* /bin/c*
   ```

5. With the option `--reverse` or `-r` we can reverse the order of display:

   ```
   ls -lt --reverse /bin/b* /bin/c*
   ```

   ```
   ls -lh --reverse /bin/b* /bin/c*
   ```

   ```
   ls -ltr /bin/b* /bin/c*
   ```

   ```
   ls -lhr /bin/b* /bin/c*
   ```

Usually the options have a long version (like `--reverse` or `--human-readable`) and a short one (like `-r` or `-h`). But not all of them. For example the options `-l` or `-t` don't have a long version.

It seems like the short options are more convenient when writing commands. In your opinion, why do we have long options as well? Why they might be useful?

> **DOWNLOAD LESSON01/PART3.CAST**

4. Exploring the system

To explore the system we use these steps:

1. Use `cd` to go to a directory.
2. List the directory contents with `ls -l`.
3. If you see an interesting file, determine its contents with the command `file`.
4. If it looks like it might be text, try viewing it with `less`.

Let's try some of these:

1. Go to `/bin` and list its content:

```
cd /bin
```

```
ls -l
```

```
ls -l b*
```

```
ls -l bzless
```

2. Check the type of some files and their contents:

```
file bzless
```

The file `bzless` is a *symbolic link*, a kind of shortcut, or alias, or a reference to another file. There are also *hard links* which we will see later.

```
ls -l bzmore
```

```
file bzmore
```

The file `bzmore` is a shell script and actually a text file, so we can read its content:

```
less bzmore
```

Press [Space] a couple of times, and then quit with `q`.

Shell scripts are like programs and contain Linux commands.

The command `less` displays the contents of a **text** file page-by-page.

Note: The command `less` is an improved replacement of an earlier Unix command that was called `more`. So, sometimes it is said that: `less` is `more`. Or: `less` is more or less `more`.

3. Let's check another file:

```
ls -lh bash
```

```
file bash
```

The file `bash` is an executable program, and a **binary** (non-text) file. Let's try to read its content:

```
less bash
```

Exit with `q`.

As you see, *text* files have a content that is readable by humans, *non-text* files (or *binary* files) have a content that is not readable by humans (but it may be read and interpreted by some programs).

4. Let's check `/etc`:

```
file /etc
```

```
ls -l /etc/passwd
```

```
file /etc/passwd
```

   It is plain text. Let's check its content:

```
less /etc/passwd
```

   This file contains the accounts of the system.

   The files on `/etc` are usually configuration files, and almost all of them are text files (readable and writable by humans).

5. In contrast, the files on `/bin` are programs or commands and they are mostly binary files or shell scripts. The same goes for `/sbin`, `/usr/bin`, `/usr/sbin`, `/usr/local/bin`, etc.

```
ls /sbin
```

```
ls /usr/bin
```

```
ls /usr/sbin
```

```
ls /usr/local/bin
```

6. Some other important directories are:

```
ls /boot
```

```
ls /boot/grub
```

Contains the Linux kernel, initial RAM disk image, the boot loader, etc.

```
ls /dev
```

```
file /dev/tty
```

```
file /dev/pts/1
```

Contains device nodes.

```
ls /home
```

Contains home directories of the users.

```
ls /lib
```

```
ls /usr/lib
```

Contains shared libraries.

```
ls /proc
```

```
less /proc/cpuinfo
```

This is a special directory that exposes the settings and the state of the kernel itself.

```
ls /var
```

```
ls /var/log
```

Contains data that are likely to change frequently (like log files).

```
ls /tmp
```

Temporary data which might be erased on each reboot.

DOWNLOAD LESSON01/PART4.CAST

📄 Intro

- Manipulating files and directories (cp, mv, mkdir, rm,

📄 1. Manipulating files and directories

To work with files and directories we can use these commands:

📄 2. Some commands about commands

1. The command type displays a command's type:

📄 3. Command history

1. The command history can be used to display the history of the

📄 4. Keyboard tricks

In the previous section we saw that we can search the command history

Intro

- Manipulating files and directories (`cp`, `mv`, `mkdir`, `rm`, `ln`). Wildcards. Symbolic and hard links.
- Working with commands (`type`, `which`, `help`, `man`, `apropos`, `info`, `whatis`, `alias`).
- Command history and keyboard tricks (`clear`, `history`, `Ctrl+a`, `Ctrl+e`, `Ctrl+r`).

DOWNLOAD LESSON02/INTRO.CAST

1. Manipulating files and directories

To work with files and directories we can use these commands:

- `cp` - Copy files and directories
- `mv` - Move/rename files and directories
- `mkdir` - Create directories
- `rm` - Remove files and directories
- `ln` - Create hard and symbolic links

Let's use them in some examples.

1. Creating directories:

```
cd
```

```
mkdir playground
```

```
cd playground
```

```
mkdir dir1 dir2
```

```
ls -l
```

2. Copying files:

```
cp /etc/passwd .
```

```
ls -l
```

Notice that `.` is the current working directory.

```
cp -v /etc/passwd .
```

The option `-v` makes the command verbose.

```
cp -i /etc/passwd .
```

The option `-i` makes the command *interactive*. This means that it asks you first, before doing any potentially destructive actions. Press `y` or `n` to continue.

3. Moving and renaming files:

```
mv passwd fun
```

```
ls -l
```

```
mv fun dir1
```

```
ls -l
```

```
ls -l dir1
```

```
mv dir1/fun dir2
```

```
```

```
ls -l dir1
```

```
ls -l dir2
```

```
mv dir2/fun .
```

```
tree
```

```
mv fun dir1
```

```
mv dir1 dir2
```

```
tree
```

```
ls -l dir2/dir1
```

```
mv dir2/dir1 .
```

```
mv dir1/fun .
```

```
tree
```

4. Creating hard links:

```
ln fun fun-hard
```

```
ln fun dir1/fun-hard
```

```
ln fun dir2/fun-hard
```

```
ls -lR
```

Notice that the second field in the listing of `fun` and `fun-hard` is `4`, which shows the number of the links for the file. Hard links are like different names for the same file content.

To make sure that all four of them are the same file, let's try the option `-i`:

```
ls -lRi
```

You may notice that the number on the first column is the same for all the files. This is called the *inode* number of a file, and can be thought as the address where the file is located. Since it is the same for all the files, this shows that they are actually the same file.

5. Creating symbolic links:

```
ln -s fun fun-sym
```

```
ls -l
```

Symbolic links are a special type of file that contains a text pointer to the target file or directory. They were created to overcome two disadvantages of hard links:

i. hard links cannot span physical devices
ii. hard links cannot reference directories, only files

```
ln -s ../fun dir1/fun-sym
```

```
ln -s ../fun dir2/fun-sym
```

```
tree
```

These two examples might seem a bit difficult to understand what is going on. But remember that when we create a symbolic link, we are creating a text description of where the target file is, relative to the symbolic link.

We can also use absolute file names when creating symbolic links:

```
ln -sf /home/user1/playground/fun dir1/fun-sym
```

```
ls -l dir1/
```

However, in most cases, using relative pathnames is more desirable, because it allows a directory tree containing symbolic links and their referenced files to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
ln -s dir1 dir1-sym
```

```
ls -l
```

6. Removing files and directories.

   Let's clean up the playground a little bit. First let's delete one of the hard links:

```
```

```
rm fun-hard
```

```
ls -l
```

Notice that the link count for `fun` is reduced from 4 to 3 (as indicated in the second field of the directory listing).

```
rm -i fun
```

Press `y`

```
ls -l
```

```
less fun-sym
```

The symbolic link now is broken.

```
rm fun-sym dir1-sym
```

```
ls -l
```

When we remove a symbolic link the target is not touched.

```
rm -r dir1/
```

```
cd ..
```

```
```

```
rm -rf playground/
```

DOWNLOAD LESSON02/PART1.CAST

2. Some commands about commands

1. The command `type` displays a command's type:

```
type type
```

```
type ls
```

```
type cp
```

   The command `cp` is an executable program located on `/usr/bin/cp`.

2. The command `which` displays the location of an executable:

```
which ls
```

```
which cd
```

   The command `cd` is not an executable but a shell builtin command.

```
type cd
```

3. The command `help` displays a help page for the shell builtin commands:

```
help cd
```

```
help mkdir
```

The command `mkdir` is not a shell builtin.

4. The option `--help` displays usage information:

```
mkdir --help
```

5. The command `man` displays the manual page of a program:

```
man ls
```

Manual pages are organized into different sections, where section 1 for example is about user commands, and section 5 is about file formats. So, these two commands will display different manual pages:

```
man passwd
```

```
man 5 passwd
```

6. The command `info` is another way to display manual pages:

```
info coreutils
```

```
info passwd
```

7. The command `apropos` displays appropriate commands:

```
apropos passwd
```

This is the same as:

```
```

```
man -k passwd
```

It makes a simple search on man pages for the term "passwd".

8. The command `whatis` displays a very brief description of a command:

```
whatis ls
```

9. The command `alias` is used to create new commands.

```
alias --help
```

```
alias
```

```
type alias
```

```
type ls
```

```
cd /usr; ls; cd -
```

```
type foo
```

```
alias foo="cd /usr; ls; cd -"
```

```
type foo
```

```
```

```
foo
```

```
unalias foo
```

```
type foo
```

DOWNLOAD LESSON02/PART2.CAST

3. Command history

1. The command `history` can be used to display the history of the typed commands:

```
history
```

```
history | less
```

```
history | tail
```

```
history | tail -n 20
```

```
history | grep man
```

2. The history is kept of in the file `~/.bash_history`:

```
echo $HISTFILE
```

```
ls $HISTFILE
```

```
tail $HISTFILE
```

Notice that the latest commands are not there. Bash maintains the list of commands internally in memory while it's running, and writes it to the history file on exit. Let's tell Bash to append the command list to the history file now:

```
history -a
```

```
tail $HISTFILE
```

3. We can re-run a previous command like this:

   ```
   !67
   ```

   Rerun the command which has the given number.

   ```
   !ls
   ```

   Rerun the last command that *starts* with `ls`.

   ```
   !?passwd
   ```

   Rerun the last command that *contains* `passwd`.

   ```
   history | grep passwd
   ```

4. We can recall the previous commands also by pressing the *up-arrow* multiple times.

5. However the most useful way to rerun previous commands is searching interactively, with keyboard shortcuts.

   For example typing "Ctrl-r" will start a reverse incremental search. It is "reverse" because it searches backwards in the history list, starting from the last command. While we start typing the search text it will display the last command that matches it. If we are happy with the search result we can just press enter to rerun it, or we can use the left and right arrows to edit it first and then press

[Enter] to run it. Otherwise we can keep pressing "Ctrl-r" to get the next matching command, and so on.

Let's try it:

i. Press "Ctrl-r".

ii. Type "pass".

iii. Press "Ctrl-r" again.

iv. Press "Ctrl-r" again.

v. Press "Enter".

DOWNLOAD LESSON02/PART3.CAST

4. Keyboard tricks

In the previous section we saw that we can search the command history and recall one of the previous commands by pressing "Ctrl-r". We can also use the *up-arrow* and *down-arrow* to select one of the previous commands, and *left-arrow* and *right-arrow* to move the cursor while editing a command.

Some other key combinations that can be useful while editing commands are:

- "Ctrl-a" -- Move cursor to the beginning of the line.
- "Ctrl-e" -- Move cursor to the end of the line.
- "Alt-f" -- Move cursor forward one word.
- "Alt-b" -- Move cursor backward one word.
- "Ctrl-l" -- Clear the screen and move the cursor to the top. Same as the `clear` command.
- "Ctrl-k" -- Kill (cut) text from the cursor location to the end of the line.
- "Ctrl-u" -- Cut text from the cursor location to the beginning of the line.
- "Ctrl-d" -- Delete the character at the cursor location.
- "Alt-d" -- Cut text from the cursor location to the end of the current word.
- "Ctrl-y" -- Yank (paste) text from the kill-ring and insert it at the cursor location.

Write a command and try to use some of these key combinations.

The shell can also help us with *completion*, if we press the TAB key while typing a command. For example try:

- `echo /etc/n*`

- Now type `cat /etc/n` (without pressing Enter) and press TAB. Press TAB a second time. The shell shows us possible completions of the command that we are typing.
- Continue to add additional letters, and pressing TAB (maybe twice or more) after each letter. When the completion is unique, press Enter.

> **DOWNLOAD LESSON02/PART4.CAST**

📄 Intro

Linux, like UNIX, is a multi-tasking and a multi-user system. This

📄 1. Ownership and permissions

In the Unix security model, a user may own files and

📄 2. Adding user accounts

1. Let's create a new user:

📄 3. Example with permissions

In this example we will set up a shared directory between the users

📄 4. Processes

A process is a program that is being executed by the system. Linux

Intro

Linux, like UNIX, is a multi-tasking and a multi-user system. This means that the system can run multiple programs at the same time, and it can be used by more than one user at the same time. One of the challenges of such a system is that it needs to identify users and protect them from each-other.

- Users and groups, permissions:
  `useradd`, `passwd`, `id`, `chmod`, `umask`, `su`, `sudo`, `chown`, `chgrp`
- Processes and signals:
  `ps`, `pstree`, `top`, `jobs`, `bg`, `fg`, `kill`, `killall`, `shutdown`

INFO

The commands in the first and second parts need to be given as user `root`. Usually you can switch to this user with the command:

```
sudo su
```

DOWNLOAD LESSON03/INTRO.CAST

1. Ownership and permissions

In the Unix security model, a user may *own* files and directories. When a user owns a file or a directory, he has control over its access (he decides who can access it). To facilitate granting permissions, users may belong to one or more groups. If the owner of a file grants permissions to a group, then all the members of the group have access to this file. Besides granting access to a group, an owner may grant some access rights to everybody, which in Unix terms is referred to as *others*.

1. When you use the command `ls -l`, the first column of the output (the one which has some dashes) shows the *attributes* of the file.

   ```
   > foo.txt
   ```

   ```
   ls -l foo.txt
   ```

   The first char of attributes shows the file type. If this char is a `-` it is a regular file, `d` is for a directory, `l` for a symbolic link, `c` for a *character special file* (for example a keyboard or network card), and `d` for *block special file* (like a hard drive or RAM).

   The remaining 9 characters show the access rights for the file's owner, the file's group, and the rest of the users. They are `rwx` for the user, `rwx` for the group, and `rwx` for the others, where `r` stands for *reading* (viewing the content of the file), `w` is for *writing* (modifying the content of the file), and `x` is for *executing* (running the file like a program or a script). If there is a minus (or a dash) instead of `r`, `w` or `x`, it means that the corresponding right is missing.

   For a directory, the `x` attribute allows a directory to be entered (e.g. `cd directory`). The `r` attribute allows a directory's content to be listed (with `ls`), but only if the `x` attribute is also set. And the `w` attribute allows files within a directory to be created, deleted, and renamed, if the `x` attribute is also set.

2. We can change the permissions of a file or directory with `chmod`. Only the owner and the superuser can change the permissions of a file or directory.

```
ls -l foo.txt
```

```
chmod 600 foo.txt
```

```
ls -l foo.txt
```

In this case we are using *octal* notation for telling `chmod` what permissions to set. For example **7** (`111`) is for `rwx`, **6** (`110`) is for `rw-`, **5** (`101`) is for `r-x`, **4** (`100`) is for `r--`, and **0** is for `---` (no permissions).

We can also use *symbolic* notation with `chmod`, where **u** (user) represents the owner, **g** represents the group, and **o** (others) represents the world. There is also the symbol **a** (all) which is a combination of **u**, **g** and **o**.

- Add the *execute* permission to the user:

  ```
  chmod u+x foo.txt
  ```

  ```
  ls -l foo.txt
  ```

- Remove the *execute* permission from the user:

  ```
  chmod u-x foo.txt
  ```

  ```
  ls -l foo.txt
  ```

- Add *execute* to user. Group and others should have only *read* and *execute*:

```
chmod u+x,go=rx foo.txt
```

```
ls -l foo.txt
```

- Remove the *execute* permission from all:

```
chmod ugo-x foo.txt
```

```
chmod a-x foo.txt
```

```
chmod -x foo.txt
```

```
ls -l foo.txt
```

3. The `umask` command controls the default permissions given to a file when it is created:

```
umask
```

This octal notation tells which bits will be *masked* (removed) from the attributes of a file:

```
rm -f foo.txt
> foo.txt
ls -l foo.txt
```

The reason that the others don't have a `w` permission is because of the mask. Remember that the number **2** in octal is written as `010`, so the permissions expressed by it are `-w-`. This means that the `w` permission for the others will be removed from the attributes.

Let's change the mask and try again:

```
rm foo.txt
```

```
umask 0000
umask
```

```
> foo.txt
ls -l foo.txt
```

Restore the normal umask:

```
umask 0022
umask
```

DOWNLOAD LESSON03/PART1.CAST

2. Adding user accounts

1. Let's create a new user:

```
useradd --help
```

```
useradd -m -s /bin/bash test1
```

Only the superuser can create new accounts, so let's use `sudo`:

```
sudo useradd -m -s /bin/bash test1
```

The option `-m` tells it to create a home directory for the user, which is by default located at `/home/`, and the option `-s` tells it what shell to use for this user.

```
ls /home/
```

```
ls -al /home/test1/
```

Users normally cannot access the content of each other. Superuser can access everything.

```
sudo ls -al /home/test1/
```

We should also set a password for `test1`:

```
sudo passwd test1
```

2. Let's switch to this user and try some commands:

```
sudo su -l test1
```

`su` means: 'switch user'

```
pwd
```

```
whoami
```

```
id
```

When a user account is created, the system assigns it a number called *user ID* or *uid*, which is mapped to a username for the sake of humans. Each user is also assigned a primary group ID (or *gid*) and may belong to additional groups.

```
exit
```

Back to the first user.

```
pwd
whoami
id
```

3. User accounts are defined in `/etc/passwd` and groups in `/etc/group`. However the passwords of the users are stored in `/etc/shadow`:

```
ls -l /etc/passwd
```

```
```

```
file /etc/passwd
```

```
less /etc/passwd
```

You can see that besides the normal users there are also some system users, including the superuser (or *root*), with uid=0.

```
ls -l /etc/group
```

```
file /etc/group
```

```
less /etc/group
```

```
ls -l /etc/shadow
```

```
file /etc/shadow
```

```
less /etc/shadow
```

You don't have permission to see the content of this file.

```
sudo less /etc/shadow
```

4. The command `chown` can be used to change the owner and/or the group of a file. Let's try it:

```
chown root: foo.txt
```

```
whoami
```

Superuser privileges are required to use it. Let's try with `sudo`.

```
sudo chown root: foo.txt
```

```
ls -l foo.txt
```

```
chown test1:root foo.txt
```

```
ls -l foo.txt
```

DOWNLOAD LESSON03/PART2.CAST

3. Example with permissions

In this example we will set up a shared directory between the users "bill" and "karen", where they can store their music files.

1. Most of the commands in this part need *root* permissions, so let's switch first to the superuser:

```
sudo su
whoami
```

2. Let's create the users "bill" and "karen":

```
useradd -m -s /bin/bash bill
```

```
ls /home/
```

```
useradd -m -s /bin/bash karen
```

```
ls /home/
```

```
tail /etc/passwd
```

3. We also need to create a group for these two users:

```
groupadd music
```

```
```

```
tail /etc/group
```

```
adduser bill music
```

```
adduser karen music
```

```
tail /etc/group
```

4. Now let's create a directory:

```
mkdir -p /usr/local/share/Music
```

```
ls -ld /usr/local/share/Music
```

To make this directory shareable we need to change the group ownership and the group permissions to allow writing:

```
chown :music /usr/local/share/Music
```

```
chmod 775 /usr/local/share/Music
```

```
ls -ld /usr/local/share/Music
```

Now we have a directory that is owned by `root` and allows read and write access to the group `music`. Users `bill` and `karen` are members of the group `music`, so they can create files in this directory. Other users can only list the contents of the directory but cannot create files there.

5. But we still have a problem. Let's try to create a file as user `bill`:

```
su -l bill
```

Now let's create an empty file, just for testing:

```
> /usr/local/share/Music/test
```

```
ls -l /usr/local/share/Music
```

The group of the created file is `bill` (which is the primary group of the user `bill`). Actually we want the group of the created file to be `music`, otherwise `karen` won't be able to access it properly.

We can fix it by giving this command (as root):

```
exit
```

```
chmod g+s /usr/local/share/Music
```

```
ls -ld /usr/local/share/Music
```

When we talked about permissions we did not mention the special permission **s**. When we give this permission to the group of a directory, the files that are created on this directory will belong to the same group as the directory.

Let's try this by creating another test file as user `bill`:

```
su -l bill
```

```
> /usr/local/share/Music/test_1
```

```
ls -al /usr/local/share/Music
```

Notice that the second file belongs to the group `music`.

```
exit
whoami

exit
whoami
```

DOWNLOAD LESSON03/PART3.CAST

4. Processes

A *process* is a program that is being executed by the system. Linux is a *multitasking* system, which means that it can run many processes at the same time. Actually, if there is only one processor, only one process can be executed at a certain time. However the Linux kernel can switch quickly between different processes, allowing each of them to run for a short time, and because this happens very fast, it gives the impression that all the programs are running in parallel.

A process in Linux is started by another process, so each process has a parent and may have some children. Only the `init` process does not have a parent because it is the first process that is started by the kernel after it is loaded.

1. We can use the command `ps` to list processes:

   ```
   ps
   ```

   It shows only the processes associated with the current terminal session. TTY is short for *Teletype* and refers to the terminal of the process. Each process has a PID (process ID number).

   ```
   ps a
   ```

   Shows all the processes associated with any terminal. TIME shows the amount of CPU time consumed by a process. STAT shows the state of the process, where `S` is for sleeping, `R` is for running, etc.

   ```
   ps au
   ```

   Displays also the user (the owner of the process). It also shows what percentage of RAM and CPU a process is using.

```
ps --help
```

```
ps --help simple
```

```
ps aux | less
```

This shows all the processes. Notice that the **process number 1** is `/sbin/init` or `/lib/systemd/systemd`. Try also:

```
ls -l /sbin/init
```

2. Another command for listing processes is `pstree`:

```
pstree
```

With `-p` shows also the PIDs:

```
pstree -p | less
```

With a username shows only the processes of that user:

```
pstree -p user1
```

With a PID shows only the branch of processes that start at that process:

```
pstree -p 700
```

3. The command `top` shows a dynamic view of the processes, which is refreshed periodically:

```
top
```

The first part of the display shows a system summary, and the second part shows a list of processes, with the most active ones at the top (those that consume more RAM, CPU and other resources).

Press `q` to quit the program.

4. When we give a command in terminal, the shell starts a new process, and waits until that process is done, before returning the prompt. For example, let's start a process that takes a long time:

```
sleep 100
```

It is going to wait for about 100 seconds. To interrupt a command that is taking too long you can press **Ctrl-c**.

If we append an ampersand (**&**) to the command, the shell we run this command in *background*. This means that we are not going to wait until the command is done, but we will get the prompt immediately, so that we can run other commands:

```
sleep 200 &
```

```
ps
```

```
jobs
```

A command in background is called a job. We can move one of the jobs in foreground with the command `fg`:

```
```

```
fg %1
```

If we don't give a job number as argument, it assumes the first job.

Now that the `sleep` job is running in terminal, we can interrupt it with Ctrl-c.

If a command is taking too long, we can also stop it with **Ctrl-z** and then start it in background. For example:

```
sleep 200
```

Now stop it with Ctrl-z. Then move it to background with `bg`:

```
jobs
```

```
bg %1
```

```
jobs
```

5. We can send signals to a process with the command `kill`:

```
sleep 100 &
```

```
ps
```

```
kill $!
```

```
ps
```

The special variable `$!` contains the PID of the last background process.

By default, `kill` sends the signal *terminate* (15 or TERM), which asks the process politely to terminate himself.

The signal *interrupt* (2 or INT) is the same signal that is sent by Ctrl-c. The program usually will terminate.

```
sleep 100 &
```

```
ps
```

```
kill -2 $!
```

```
ps
```

The signal *stop* (19 or STOP) is not delivered to the process. Instead the kernel pauses the process, without terminating it (same as Ctrl-z for a foreground process).

```
sleep 300 &
```

```
jobs
```

```
kill -STOP $!
```

```
jobs
```

The signal *continue* (18 or CONT) will restore a process after it has been paused with STOP.

```
kill -CONT $!
```

```
jobs
```

The signal *kill* (9 or KILL) is also not delivered to the process. Instead, the kernel terminates the process immediately, no questions asked. This is usually used as a last resort if the process is not responding to the other signals.

```
kill -SIGKILL $!
```

```
jobs
```

There are many other signals as well:

```
kill -l
```

6. The command `killall` can send a signal to multiple processes that match a given program or username:

```
sleep 100 &
```

```
sleep 200 &
```

```
sleep 300 &
```

```
```

```
jobs
```

```
ps
```

```
killall sleep
```

```
jobs
```

```
ps
```

7. If you have superuser permissions, you can also try these commands to shutdown or reboot the system:

   `halt`

   `poweroff`

   `reboot`

   `shutdown -h now`

   `shutdown -r now`

> **DOWNLOAD LESSON03/PART4.CAST**

📄 Intro

Commands and programs in Linux usually produce some output. This

📄 1. Redirecting stdout and stderr

1. To redirect standard output to a file we can use the ">"

📄 2. Redirecting standard input

A command that makes use of standard input is cat (which is a

📄 3. Pipelines and filters

1. Using the pipe operator "|" (vertical bar), the standard output

Intro

Commands and programs in Linux usually produce some output. This output is of two types:

1. The program's results, that is the data that the program is designed to produce
2. Status and error messages, which tell how the program is getting along.

Programs usually output the results to the *standard output* (or **stdout**) and the status messages to the *standard error* (**stderr**). By default, both *stdout* and *stderr* are linked to the screen (computer display).

In addition, many programs take input from a facility called *standard input* (**stdin**), which is by default attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection we can change that.

We can also chain several commands together in a pipeline, where the output of a command is sent as the input of another. This is a powerful feature that allows us to perform complex operations on data by combining simple utilities.

DOWNLOAD LESSON04/INTRO.CAST

1. Redirecting stdout and stderr

1. To redirect standard output to a file we can use the "`>`" redirection operator.

```
ls -l /usr/bin
```

```
ls -l /usr/bin > ls-output.txt
```

```
ls -l ls-output.txt
```

```
less ls-output.txt
```

2. Let's try the same example with a directory that does not exist:

```
ls -l /bin/usr
```

```
ls -l /bin/usr > ls-output.txt
```

`ls` does not send its error messages to standard output.

```
ls -l ls-output.txt
```

The file has zero length.

```
less ls-output.txt
```

The redirection operator `>` has erased the previous content of the file. In fact, if we ever need to truncate (erase the content of) a file, or to create a new empty file, we can use a trick like this:

```
> ls-output.txt
```

3. To actually append the redirected output to the existing content of the file, instead of overwriting it, we can use the operator "`>>`":

```
ls -l /usr/bin >> ls-output.txt
```

```
ls -lh ls-output.txt
```

```
ls -l /usr/bin >> ls-output.txt
```

```
ls -lh ls-output.txt
```

```
ls -l /usr/bin >> ls-output.txt
```

```
ls -lh ls-output.txt
```

Notice that the size of the file is growing each time.

4. To redirect stderr we can use the operators "`2>`" and "`2>>`". In Linux, the standard output has the *file descriptor* (file stream number) **1**, and the standard error has the *file descriptor* **2**. So, hopefully this syntax makes sense to you and is similar to that of redirecting stdout.

```
ls -l /bin/usr 2> ls-error.txt
```

```
ls -l ls-error.txt
```

```
less ls-error.txt
```

5. We can redirect both stdout and stderr to the same file, like this:

```
ls -l /bin/usr > ls-output.txt 2>&1
```

The redirection `2>&1` redirects the file descriptor `2` (stderr) to the file descriptor `1` (stdout). But before that we redirected the stdout to `ls-output.txt`, so both stdout and stderr will be directed to this file.

Notice that the order of the redirections is significant. Let's try it like this:

```
ls -l /bin/usr 2>&1 >ls-output.txt
```

In this case we redirect file descriptor `2` (stderr) to file descriptor `1`, which is already the screen, and then we redirect the file descriptor `1` (stdout) to the file. So, the error messages will still be sent to the screen and not to the file.

A shortcut for redirecting both stdout and stderr to the same file is using "`&>`":

```
ls -l /bin/usr &> ls-output.txt
```

For appending to the file we can use "`&>>`":

```
ls -l /bin/usr &>> ls-output.txt
```

```
ls -lh ls-output.txt
```

```
ls -l /bin/usr &>> ls-output.txt
```

```
ls -lh ls-output.txt
```

6. To throw away the output or the error messages of a command, we can send them to `/dev/null`:

```
ls -l /bin/usr 2> /dev/null
```

> **DOWNLOAD LESSON04/PART1.CAST**

2. Redirecting standard input

A command that makes use of standard input is `cat` (which is a shortening for concatenate). It usually takes one or more files as arguments and outputs their contents to the screen, joined together.

```
cat ls-output.txt ls-error.txt
```

However if no files are given as arguments it just reads lines from the standard input (keyboard by default) and writes them to the standard output (screen by default). Let's try it:

```
cat
```

Type a couple of lines and thes press "Ctrl-d" to tell `cat` that it has reached the *end of file* (EOF) on standard input.

If we redirect the standard output to a file, then it can be used to create short files. For example try these:

```
cat > lazy_dog.txt
```

```
The quick brown fox jumped over the lazy dog.
```

Press "Ctrl-d" at the end.

```
cat lazy_dog.txt
```

To redirect the *standard input* (**stdin**) we can use the redirection operator "`<`", like this:

```
cat < lazy_dog.txt
```

We have changed the source of standard input from the keyboard to the file `lazy_dog.txt`.

> **DOWNLOAD LESSON04/PART2.CAST**

3. Pipelines and filters

1. Using the pipe operator "`|`" (vertical bar), the standard output (stdout) of a command can be *piped* to the standard input (stdin) of another command. This is a powerful feature that allows us to perform complex operations on data by combining simple utilities. Let's see some examples:

```
ls -l /usr/bin
```

```
ls -l /usr/bin | less
```

2. We can sort the data with `sort`:

```
ls /bin /usr/bin
```

```
ls /bin /usr/bin | sort | less
```

3. `uniq` can omit or report repeated lines:

```
ls /bin /usr/bin | sort | uniq | less
```

   If we want to see the list of duplicates instead we can use the option `-d`:

```
ls /bin /usr/bin | sort | uniq -d | less
```

4. `wc` counts the lines, words, and bytes of the input:

```
wc ls-output.txt
```

```
cat ls-output.txt | wc
```

If we want it to show only the lines we can use the option `-l`:

```
ls /bin /usr/bin | sort | wc -l
```

```
ls /bin /usr/bin | sort -u | wc -l
```

```
ls /bin /usr/bin | sort | uniq -d | wc -l
```

5. `grep` prints the lines that match a given pattern:

```
ls /bin /usr/bin | sort -u | grep zip
```

```
ls /bin /usr/bin | sort -u | grep zip | wc -l
```

The option `-v` shows the lines that do not match the pattern:

```
ls /bin /usr/bin | sort -u | grep -v zip
```

```
ls /bin /usr/bin | sort -u | grep -v zip | wc -l
```

The option `-i` can be used if we want grep to ignore case when searching (case in-sensitive search).

6. `head` / `tail` print the top or the last lines of input:

```
ls /usr/bin > ls-output.txt
```

```
head ls-output.txt
```

```
tail ls-output.txt
```

```
tail -n 5 ls-output.txt
```

```
tail -5 ls-output.txt
```

```
ls /usr/bin | head -n 5
```

```
tail /var/log/dpkg.log -n 20
```

```
tail /var/log/bootstrap.log -f
```

The option `-f` makes it follow the latest changes of the file in real time. Press "Ctrl-c" to terminate it.

7. `tee` sends its input both to stdout and to files:

```
ls /usr/bin | tee ls.txt | grep zip
```

```
ls -l ls.txt
```

```
less ls.txt
```

> **DOWNLOAD LESSON04/PART3.CAST**

📄 Intro

- Each time we type a command and press the Enter key, bash does

📄 1. Shell expansions

1. Wildcard expansions:

📄 2. Shell quotes

1. Using quotes in a command affects the spaces:

📄 3. The Environment

We have seen previously some environment variables. These are

Intro

- Each time we type a command and press the Enter key, `bash` does some kind of processing upon the text before executing our command. For example if there is a "`*`" in the command, it is replaced by the names of the matching files. This replacement is called an **expansion**.
- Quoting is used to control how the shell splits the input into parts. It also disables some types of expansion. Both single and double quotes can be used, and there are some differences between them.
- The shell environment (printenv, set, export). Customizing the prompt.

DOWNLOAD LESSON05/INTRO.CAST

1. Shell expansions

1. Wildcard expansions:

```
echo this is a test
```

`echo` displays all the arguments that are passed to it.

```
cd /usr
```

```
ls
```

```
echo *
```

The "`*`" character means match any characters in a filename. The shell expands the "`*`" before executing the command `echo`.

```
echo lib*
```

```
echo *bin
```

```
echo lib*32
```

```
echo */share
```

```
echo /*/*/bin
```

A question mark "`?`" matches any single character:

```
echo lib??
```

```
echo lib???
```

Character sets are enclosed in square brackets:

```
echo lib[123456789]?
```

```
echo lib[xyz][123456789]?
```

Character ranges:

```
echo lib[1-9][1-9]
```

```
echo lib[a-z][1-9][1-9]
```

```
echo lib[1-9][!2]
```

Character classes:

```
echo lib[[:digit:]][[:digit:]]
```

```
echo lib[[:alpha:]][[:digit:]][[:digit:]]
```

```
echo lib[[:alnum:]][[:alnum:]][[:alnum:]]
```

```
echo lib[![:digit:]]*
```

```
echo lib[36[:lower:]]*
```

```
echo /etc/[[:upper:]]*
```

2. The tilde character ("`~`") expands to the home directory:

```
echo ~
```

```
echo ~root
```

3. Arithmetic expansion:

```
echo $((2 + 2))
```

```
echo $(($((5**2)) * 3))
```

```
echo $(((5**2) * 3))
```

```
echo Five divided by two equals $((5/2))
```

```
echo with $((5%2)) left over
```

4. Brace expansion:

```
```

```
echo Front-{A,B,C}-Back
```

```
echo Number_{1..5}
```

```
echo {01..15}
```

```
echo {001..15}
```

```
echo {Z..A}
```

```
echo a{A{1,2},B{3,4}}b
```

```
cd
```

```
mkdir Photos
```

```
cd Photos
```

```
mkdir {2017..2019}-{01..12}
```

```
ls
```

5. Variable expansion:

```
echo $USER
```

```
printenv | less
```

```
echo $SUER
```

When the variable does not exist, it is expanded to the empty string.

6. Command substitution:

```
echo $(ls)
```

```
ls -l $(which cp)
```

```
echo "$(cal)"
```

> **DOWNLOAD LESSON05/PART1.CAST**

2. Shell quotes

1. Using quotes in a command affects the spaces:

```
echo this is a      test
```

```
echo "this is a      test"
```

When the shell parses the first command it finds 4 arguments: "this", "is", "a", "test". Then it calls `echo` passing it these arguments.

In the second case the quotes let the shell know that there is a single argument: "this is a test", and it passes to `echo` only one argument.

2. Double quotes do not prevent variable expansion, but single quotes do:

```
echo The total is $100.0
```

```
echo "The total is $100.0"
```

```
echo 'The total is $100.0'
```

Bash reckognizes `$1` as a special variable and tries to replace its value (which is empty). Using double quotes does not prevent Bash from expanding variables, we need single quotes for that.

3. Double quotes do not prevent the shell expansions that start with a "`$`", but prevents the others:

```
```

```
echo ~/*.txt {a,b} $(echo foo) $((2 + 2))
```

```
echo "~/*.txt {a,b} $(echo foo) $((2 + 2))"
```

```
echo '~/*.txt {a,b} $(echo foo) $((2 + 2))'
```

They are useful for preserving the spaces, for example:

```
echo $(cal)
```

```
echo "$(cal)"
```

4. We can also escape "`$`" by preceding it with "`\`":

```
echo The balance is: $5.00
```

```
echo The balance is: \$5.00
```

5. The option `-e` of `echo` will also enable other escape sequences like `\n` (for a new line) and `\t` (for a tab):

```
echo "a\nb"
```

```
echo -e "a\nb"
```

```
echo "a\tb"
```

```
echo -e "a\tb"
```

DOWNLOAD LESSON05/PART2.CAST

3. The Environment

We have seen previously some *environment variables*. These are variables that are maintained by the shell and are used to store some settings. They can also be used by some programs to get configuration values.

1. We can display a list of environment variables with `printenv` or `set`:

   ```
   printenv | less
   ```

   ```
   set | less
   ```

   The list displayed by `set` is longer because it displays also shell variables and some functions defined in the shell.

   ```
   printenv USER
   ```

   ```
   echo $USER
   ```

   The `USER` variable basically keeps the value that is displayed by the command `whoami`.

2. Some other interesting environment vars are these:

   ```
   echo $HOME
   ```

   ```
   echo $PWD
   ```

   ```
   ```

```
echo $SHELL
```

```
echo $LANG
```

```
echo $PATH
```

`PATH` is used by shell to find a program. For example when we call `ls`, shell is looking for it in the first directory of the PATH, then in the second, and so on. The command `which ls` shows us where the shell finds the program `ls`.

3. The environment variables are declared in some configuration files that the shell loads when it starts. There are two kinds of shells: a *login* shell session, which is started when we are prompted for a username and password, and a *non-login* shell session, which is started when we launch a terminal.

   The configuration scripts loaded by a *login* shell:

   ```
   nano /etc/profile
   ```

   ```
   nano ~/.profile
   ```

   Note: Type **Ctrl-x** to exit from `nano`.

   The configuration scripts loaded by a *non-login* shell:

   ```
   nano /etc/bash.bashrc
   ```

   ```
   nano ~/.bashrc
   ```

However the non-login shell inherits the environment variables from the parent process, usually a login shell, and the config scripts of a login shell usually include the config scripts of a non-login shell. So, if we want to make some changes to the environment, the right place to edit is the file `~/.bashrc`.

4. Let's say that we want to modify the variable HISTSIZE, which keeps the size of the command history.

```
echo $HISTSIZE
```

```
nano ~/.bashrc
```

Add this line at the end of the file:

```
export HISTSIZE=2000
```

Press Ctrl-o and Enter to save the changes. Then Ctrl-x to exit.

With `HISTSIZE=2000` we are giving a new value to the variable, and the command `export` actually saves it to the environment of the shell.

Next time that we will start a shell it will load `~/.bashrc` and a new value will be set to HISTSIZE. But we can also load `~/.bashrc` with the command `source`, so that the changes are applied right now:

```
echo $HISTSIZE
```

```
source ~/.bashrc
```

```
echo $HISTSIZE
```

5. One of the environment variables is `PS1`, which defines the prompt:

```
echo $PS1
```

Let's try to play with it, but first let's backup the current value:

```
ps1_old="$PS1"
```

```
echo $ps1_old
```

If we need to restore we can do it like this:

```
PS1="$ps1_old"
```

Let's try some other prompts:

```
PS1="--> "
```

```
ls -al
```

```
PS1="\$ "
```

```
ls -al
```

```
PS1="\A \h \$ "
```

```
ls -al
```

"\A" displays the time of day and "\h" displays the host.

```
PS1="<\u@\h \W>\$ "
```

```
ls -al
```

"\u" displays the user and "\W" displays the name of the current directory.

To save this prompt for future sessions of the shell, we should append this line to `~/.bashrc`:

```
export PS1="<\u@\h \W>\$ "
```

DOWNLOAD LESSON05/PART3.CAST

📄 Intro

In this tutorial we will learn about:

📄 1. Searching for files

1. We can make a quick search for files with locate:

📄 2. More examples with find

1. Let's create some test files:

📄 3. Regular expressions

Regular expressions are symbolic notations used to identify patterns

📄 4. More regex examples

1. Suppose that we are solving a crossword puzzle and we need a five

Intro

In this tutorial we will learn about:

- Searching for files (locate, find).
- Regular expressions (grep).

Regular expressions are symbolic notations used to identify patterns in text. They are supported by many command line tools and by most of programming languages to facilitate the solution of text manipulation problems.

DOWNLOAD LESSON06/INTRO.CAST

1. Searching for files

1. We can make a quick search for files with `locate`:

   ```
   locate bin/zip
   ```

   If the search requirement is not so simple, we can combine `locate` with other tools, like `grep`:

   ```
   locate zip | grep bin
   ```

   ```
   locate zip | grep bin | grep -v unzip
   ```

2. While `locate` searches are based only on the file name, with `find` we can also make searches based on other attributes of files.

   It takes as arguments one or more directories that are to be searched:

   ```
   ls -aR ~
   ```

   ```
   find ~
   ```

   To find only directories we can use the option `-type d` and to find only files we can use `-type f`:

   ```
   find . -type d
   ```

   ```
   find . -type f
   ```

```
find . -type d | wc -l
```

```
find . -type f | wc -l
```

```
find . | wc -l
```

3. We can also search by filename and file size:

```
sudo find /etc -type f | wc -l
```

```
sudo find /etc -type f -name "*.conf" | wc -l
```

We enclose the search pattern in double quotes to prevent shell from expanding "`*`".

```
sudo find /etc -type f -name "*.conf" -size -2k | wc -l
```

```
sudo find /etc -type f -name "*.conf" -size 2k | wc -l
```

```
sudo find /etc -type f -name "*.conf" -size +2k | wc -l
```

`-2k` matches the files whose size is less than 2 Kilobytes, `2k` those who are exactly 2 Kilobytes, and `+2k` those who are more than 2 Kilobytes. Besides `k` we can also use `M` for Megabytes, `G` for Gigabytes, etc.

4. `find` supports many other tests on files and directories, like the time of creation or modification, the ownership, permissions, etc. These tests can be combined with *logical operators* to create more complex logical relationships. For example:

```
find ~ \( -type f -not -perm 0600 \) -or \( -type d -not -perm 0700 \)
```

This looks weird, but if we try to translate it to a more understandable language it means: find on home directory (files with bad permissions) -or (directories with bad permissions). We have to escape the parentheses to prevent shell from interpreting them.

5. We can also do some actions on the files that are found. The default action is to `-print` them to the screen, but we can also `-delete` them:

```
touch test{1,2,3}.bak
```

```
ls
```

```
find . -type f -name '*.bak' -delete
```

```
ls
```

```
touch test{1,2,3}.bak
```

```
find . -type f -name '*.bak' -print -delete
```

```
ls
```

We can also execute custom actions with `-exec`:

```
touch test{1,2,3}.bak
```

```
ls
```

```
find . -name '*.bak' -exec rm '{}' ';'
```

```
ls
```

Here `{}` represents the pathname that is found and `;` is required to indicate the end of the command. Both of them have been quoted to prevent shell from interpreting them.

If we use `-ok` instead of `-exec` then each command will be confirmed before being executed:

```
touch test{1,2,3}.bak
```

```
ls
```

```
find . -name '*.bak' -ok rm '{}' ';'
```

6. Another way to perform actions on the results of `find` is to pipe them to `xargs`, like this:

```
touch test{1,2,3}.bak
```

```
ls
```

```
find . -name '*.bak' | xargs echo
```

```
find . -name '*.bak' | xargs ls -l
```

```
find . -name '*.bak' | xargs rm
```

```
ls
```

`xargs` gets input from stdin and converts it into an argument list for the given command.

> **DOWNLOAD LESSON06/PART1.CAST**

2. More examples with `find`

1. Let's create some test files:

```
mkdir -p test/dir-{001..100}
```

```
touch test/dir-{001..100}/file-{A..Z}
```

The command `touch` in this case creates empty files.

```
ls test/
```

```
ls test/dir-001/
```

```
ls test/dir-002/
```

2. Find all the files named `file-A`:

```
find test -type f -name 'file-A'
```

```
find test -type f -name 'file-A' | wc -l
```

3. Create a timestamp file:

```
touch test/timestamp
```

This creates an empty file and sets its modification time to the current time. We can verify this with `stat` which shows everything that the system knows about a file:

```
stat test/timestamp
```

```
touch test/timestamp
```

```
stat test/timestamp
```

We can see that after the second `touch` the times have been updated.

4. Next, let's use `find` to update all files named `file-B`:

```
find test -name 'file-B' -exec touch '{}' ';'
```

5. Now let's use `find` to identify the updated files by comparing them to the reference file `timestamp`:

```
find test -type f -newer test/timestamp
```

```
find test -type f -newer test/timestamp | wc -l
```

The result contains all 100 instances of `file-B`. Since we did a `touch` on them after updating `timestamp`, they are now "newer" than the file `timestamp`.

6. Let's find again the files and directories with bad permissions:

```
find test \( -type f -not -perm 0600 \)     \
      -or \( -type d -not -perm 0700 \)
```

```
find test \( -type f -not -perm 0600 \)     \
      -or \( -type d -not -perm 0700 \)     \
  | wc -l
```

7. Let's add some actions to the command above in order to fix the permissions:

```
find test    \
    \(       \
        -type f -not -perm 0600     \
        -exec chmod 0600 '{}' ';'   \
    \) -or   \
    \(       \
        -type d -not -perm 0700     \
        -exec chmod 0700 '{}' ';'   \
    \)
```

```
find test \( -type f -not -perm 0600 \)     \
      -or \( -type d -not -perm 0700 \)
```

```
find test \( -type f -perm 0600 \)    \
      -or \( -type d -perm 0700 \)
```

```
find test \( -type f -perm 0600 \)    \
      -or \( -type d -perm 0700 \)    \
  | wc -l
```

Note: This example is a bit complex just to illustrate the logical operators and parantheses, however we could have done it in two simpler steps, like this:

```
find test -type f -not -perm 0600  \
          -exec chmod 0600 '{}' ';'
```

```
```

```
find test -type d -not -perm 0700  \
          -exec chmod 0700 '{}' ';'
```

8. Let's try some more tests:

   Find files or directories whose *contents or attributes* were modified more than 1 minute ago:

   ```
   find test/ -cmin +1 | wc -l
   ```

   Less than 10 minutes ago:

   ```
   find test/ -cmin -10 | wc -l
   ```

   Find files or directories whose *contents* were modified more than 1 minute ago:

   ```
   find test/ -mmin +1 | wc -l
   ```

   Less than 10 minutes ago:

   ```
   find test/ -mmin -10 | wc -l
   ```

   Find files or directories whose *contents or attributes* were modified more than 7 days ago:

   ```
   find test/ -ctime +7 | wc -l
   ```

   Find files or directories whose *contents* were modified less than 7 days ago:

   ```
   find test/ -mtime -7 | wc -l
   ```

Find empty files and directories:

```
find test/ -empty | wc -l
```

DOWNLOAD LESSON06/PART2.CAST

3. Regular expressions

Regular expressions are symbolic notations used to identify patterns in text. They are supported by many command line tools and by most of programming languages to facilitate the solution of text manipulation problems.

1. We will test regular expressions with `grep` (which means "global regular expression print"). It searches text files for the occurrence of text matching a specified regular expression and outputs any line containing a match to standard output.

   ```
   ls /usr/bin | grep zip
   ```

   In order to explore `grep`, let's create some text files to search:

   ```
   ls /bin > dirlist-bin.txt
   ```

   ```
   ls /usr/bin > dirlist-usr-bin.txt
   ```

   ```
   ls /sbin > dirlist-sbin.txt
   ```

   ```
   ls /usr/sbin > dirlist-usr-sbin.txt
   ```

   ```
   ls dirlist*.txt
   ```

   We can do a simple search on these files like this:

   ```
   grep bzip dirlist*.txt
   ```

If we are interested only in the list of files that contain matches, we can use the option `-l`:

```
grep -l bzip dirlist*.txt
```

Conversely, if we want to see a list of files that do not contain a match, we can use `-L`:

```
grep -L bzip dirlist*.txt
```

2. While it may not seem apparent, we have been using regular expressions in the searches we did so far, albeit very simple ones. The regular expression "bzip" means that a line will match if it contains the letters "b", "z", "i", "p" in this order and without other characters in between.

   Besides the *literal characters*, which represent themselves, we can also use *metacharacters* in a pattern. For example a *dot* (`.`) matches any character:

   ```
   grep -h '.zip' dirlist*.txt
   ```

   The option `-h` suppresses the output of filenames.

   Notice that the `zip` program was not found because it has only 3 letters and does not match the pattern.

3. The caret (`^`) and dollar sign (`$`) are treated as *anchors* in regular expressions. This means that they cause the match to occur only if the regular expression is found at the beginning of the line (`^`) or at the end of the line (`$`):

   ```
   grep -h '^zip' dirlist*.txt
   ```

   ```
   grep -h 'zip$' dirlist*.txt
   ```

```
grep -h '^zip$' dirlist*.txt
```

Note that the regular expression '`^$`' will match empty lines.

4. Using *bracket expressions* we can match a single character from a specified set of characters:

```
grep -h '[bg]zip' dirlist*.txt
```

If the first character in a bracket expression is a caret (`^`), then any character will be matched, except for those listed:

```
grep -h '[^bg]zip' dirlist*.txt
```

The caret character only invokes negation if it is the first character within the bracket expression; otherwise it loses its special meaning and becomes an ordinary character in the set:

```
grep -h '[b^g]zip' dirlist*.txt
```

5. If we want to find all lines that start with an uppercase letter, we can do it like this:

```
grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXYZ]' dirlist*.txt
```

We can do less typing if we use a range:

```
grep -h '^[A-Z]' dirlist*.txt
```

If we want to match any alphanumeric character (all the letters and digits), we can use several ranges, like this:

```
grep -h '^[A-Za-z0-9]' dirlist*.txt
```

However the dash (`-`) character in this example stands for itself, does not make a range:

```
grep -h '^[-AZ]' dirlist*.txt
```

Besides ranges, another way to match groups of characters is using POSIX character classes:

```
grep -h '^[[:alnum:]]' dirlist*.txt
```

```
ls /usr/sbin/[[:upper:]]*
```

Other character classes are: `[:alpha:]`, `[:lower:]`, `[:digit:]`, `[:space:]`, `[:punct:]` (for punctuation characters), etc.

6. With a vertical bar (`|`) we can define alternative matching patterns:

```
echo "AAA" | grep AAA
```

```
echo "BBB" | grep BBB
```

```
echo "AAA" | grep 'AAA\|BBB'
```

```
echo "BBB" | grep -E 'AAA|BBB'
```

```
echo "CCC" | grep -E 'AAA|BBB'
```

```
echo "CCC" | grep -E 'AAA|BBB|CCC'
```

The option `-E` tells `grep` to use *extended* regular expressions. With extended regular expressions the vertical bar (`|`) is a metacharacter (used for alternation) and we need to escape it (with `\`) to use it as a literal character. With *basic* regular expressions (without the option `-E`) the vertical bar is a literal character and we need to escape it (with `\`) if we want to use it as a metacharacter.

7. Other metacharacters that are recognized by extended regular expressions, and which behave similar to `|` are: `(`, `)`, `{`, `}`, `?`, `+`. For example:

```
grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

Note that this is different from:

```
grep -Eh '^bz|gz|zip' dirlist*.txt
```

In the first example all the patterns are matched at the beginning of the line. In the second one only `bz` is matched at the beginning.

DOWNLOAD LESSON06/PART3.CAST

4. More regex examples

1. Suppose that we are solving a crossword puzzle and we need a five letter word whose third letter is "j" and last letter is "r". Let's try to use `grep` and regex to solve this.

   Fist of all make sure that we have a dictionary of words installed:

   ```
   sudo apt install wbritish
   ```

   ```
   ls /usr/share/dict/
   ```

   ```
   less /usr/share/dict/words
   ```

   ```
   cat /usr/share/dict/words | wc -l
   ```

   Now try this:

   ```
   grep -i '^..j.r$' /usr/share/dict/words
   ```

   The option `-i` is used to ignore the case (uppercase, lowercase).

   The regex pattern `'^..j.r$'` will match lines that contain exactly 5 letters, where the third letter is `j` and the last one is `r`.

2. Let's say that we want to check a phone number for validity and we consider a phone number to be valid if it is in the form `(nnn) nnn-nnnn` or in the form `nnn nnn-nnnn` where `n` is a digit. We can do it like this:

   ```
   ```

```
echo "(555) 123-4567" | \
    grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9]
[0-9]$'
```

```
echo "555 123-4567" | \
    grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9]
[0-9]$'
```

```
echo "AAA 123-4567" | \
    grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9]
[0-9]$'
```

Since we are using the option `-E` (for extended), we have to escape the parentheses `\(` and `\)` so that they are not interpreted as metacharacters.

If we use basic regular expressions (without `-E`), then we don't need to escape the parentheses, but in this case we will have to escape the question marks (`\?`) so that they are interpreted as metacharacters:

```
echo "(555) 123-4567" | \
    grep '^(\?[0-9][0-9][0-9])\? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-
9]$'
```

The question mark as a metacharacter means that the parentheses before it can be zero or one time.

3. Using the metachars `{}` we can express the number of required matches. For example:

```
echo "(555) 123-4567" | \
    grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
```

The expression `{3}` matches if the preceding element occurs exactly 3 times.

We could also replace `?` by `{0,1}`, or `{,1}`:

```
echo "(555) 123-4567" | \
    grep -E '^\({0,1}[0-9]{3}\){,1} [0-9]{3}-[0-9]{4}$'
```

```
echo "555 123-4567" | \
    grep -E '^\({0,1}[0-9]{3}\){,1} [0-9]{3}-[0-9]{4}$'
```

In general, `{n,m}` matches if the preceding element occurs at least `n` times, but no more than `m` times. These are also valid: `{n,}` (at least `n` times), and `{,m}` (at most `m` times).

4. Similar to `?` which is equivalent to `{0,1}`, there is also `*` which is equivalent to `{0,}` (zero or more occurrences), and `+` which is equivalent to `{1,}` (one or more, at least one occurrence):

   Let's say that we want to check if a string is a sentence. This means that it starts with an uppercase letter, then contains any number of upper and lowercase letters and spaces, and finally ends with a period. We could do it like this:

   ```
   echo "This works." | grep -E '[A-Z][A-Za-z ]*\.'
   ```

   ```
   echo "This Works." | grep -E '[A-Z][A-Za-z ]*\.'
   ```

   ```
   echo "this does not" | grep -E '[A-Z][A-Za-z ]*\.'
   ```

   Or like this:

   ```
   echo "This works." | grep -E '[[:upper:]][[:upper:][:lower:] ]*\.'
   ```

Note: In all these cases we have to escape the period (`\.`) so that it matches itself instead of any character.

5. Here is a regular expression that will only match lines consisting of groups of one or more alphabetic characters separated by single spaces:

```
echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
```

```
echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
```

```
echo "a b  c" | grep -E '^([[:alpha:]]+ ?)+$'
```

Does not match because there are two consecutive spaces.

```
echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
```

Does not match because there is a non-alphabetic character.

6. Let's create a list of random phone numbers for testings:

```
echo $RANDOM
```

```
echo $RANDOM
```

```
echo ${RANDOM:0:3}
```

```
for i in {1..10}; do \
    echo "${RANDOM:0:3} ${RANDOM:0:3}-${RANDOM:0:4}" >>
phonelist.txt; \
done
```

```
cat phonelist.txt
```

```
for i in {1..100}; do \
    echo "${RANDOM:0:3} ${RANDOM:0:3}-${RANDOM:0:4}" >>
phonelist.txt; \
done
```

```
less phonelist.txt
```

```
cat phonelist.txt | wc -l
```

You can see that some of the phone numbers are malformed. We can display those that are malformed like this:

```
grep -Ev '^[0-9]{3} [0-9]{3}-[0-9]{4}$' phonelist.txt
```

The option `-v` makes an inverse match, which means that `grep` displays only the lines that do not match the given pattern.

7. Regular expressions can be used with many commands, not just with `grep`.

   For example let's use them with `find` to find the files that contain bad characters in their name (like spaces, punctuation marks, etc):

```
touch "bad file name!"
```

```
ls -l
```

```
find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Different from `grep`, `find` expects the pattern to match the whole filename, that's why we are appending and prepending `.*` to the pattern.

We can use regular expressions with `locate` like this:

```
locate --regex 'bin/(bz|gz|zip)'
```

We can also use them with `less`:

```
less phonelist.txt
```

We can press `/` and write a regular expression, and `less` will find and highlight the matching lines. For example:

```
/^[0-9]{3} [0-9]{3}-[0-9]{4}$
```

The invalid lines will not be highlighted and will be easy to spot.

Regular expressions can also be used with `zgrep` like this:

```
cd /usr/share/man/man1
```

```
zgrep -El 'regex|regular expression' *.gz
```

It will find man pages that contain either "regex" or "regular expression". As we can see, regular expressions show up in a lot of programs.

DOWNLOAD LESSON06/PART4.CAST

📄 Intro

Linux relies heavily on text files for data storage, so it makes sense

📄 1. sort

1. Let's try and compare these commands:

📄 2. cut

1. The command cut extracts a certain column (field) from the input,

📄 3. paste

The paste command does the opposite of cut. Rather than extracting

📄 4. join

The command join, like paste, adds columns to a file. However it

📄 5. Comparing text

1. Let's create two test files:

📄 6. Editing on the fly

1. The tr program is used to transliterate characters:

📄 7. aspell

1. Plain text files:

Intro

Linux relies heavily on text files for data storage, so it makes sense that there are many tools for manipulating text. Some of these tools are:

- `cat` -- Concatenate files and print on the standard output
- `sort` -- Sort lines of text files
- `uniq` -- Report or omit repeated lines
- `cut` -- Extract sections from each line of files
- `paste` -- Merge lines of files
- `join` -- Join lines of two files on a common field
- `comm` -- Compare two sorted files line by line
- `diff` -- Compare files line by line
- `patch` -- Apply a diff file to an original
- `tr` -- Translate or delete characters
- `sed` -- Stream editor for filtering and transforming text
- `aspell` -- Interactive spell checker

DOWNLOAD LESSON07/INTRO.CAST

1. sort

1. Let's try and compare these commands:

```
du -s /usr/share/* | less
```

```
du -s /usr/share/* | sort | less
```

```
du -s /usr/share/* | sort -r | less
```

```
du -s /usr/share/* | sort -nr | less
```

```
du -s /usr/share/* | sort -nr | head
```

The command `du` gets the size (disk usage) of the files and directories of `/usr/share`, and `head` filters the top 10 results.

Then we try to sort them with `sort` and `sort -r` (reverse), but it does not seem to work as expected (sorting results by the size). This is because `sort` by default sorts the first column alphabetically, so `2` is bigger than `10` (because `2` comes after `1` on the character set).

With the option `-n` we tell sort to do a *numerical* sort. So, the last command returns the top 10 biggest files and directories on `/usr/share`.

2. This example works because the numerical values happen to be on the first column of the output. What if we want to sort a list based on another column? For example the result of `ls -l`:

```
```

```
ls -l /usr/bin | head
```

Ignoring for the moment that `ls` can sort its results by size, we could use `sort` to sort them like this:

```
ls -l /usr/bin | sort -nr -k 5 | head
```

The option `-k5` tells `sort` to use the fifth field as the key for sorting. By the way, `ls` like most of the commands, separates the fields of its output by a TAB.

3. For testing we are going to use the file distros.txt, which is like a history of some Linux distributions (containing their versions and release dates).

```
wget https://linux-cli.fs.al/examples/lesson07/distros.txt
cat distros.txt
```

```
cat -A distros.txt
```

The option `-A` makes it show any special characters. The tab character is represented by `^I`, and the `$` shows the end of line.

4. Let's try to sort it:

```
sort distros.txt
```

The result is almost correct, but Fedora version numbers are not in the correct order (since `1` comes before `5` in the character set).

To fix this we are going to sort on multiple keys. We want an alphabetic sort on the first field and a numeric sort on the second field:

```
sort --key=1,1 --key=2n distros.txt
```

```
sort -k 1,1 -k 2n distros.txt
```

```
sort -k1,1 -k2n distros.txt
```

Notice that if we don't use a range of fields (like `1,1`, which means start at field 1 and end at field 1), it is not going to work as expected:

```
sort -k 1 -k 2n distros.txt
```

This is because in this case it starts at field 1 and goes up to the end of the line, ignoring thus the second key.

The modifier `n` stands for *numerical sort*. Other modifiers are `r` for *reverse*, `b` for *ignore blanks*, etc.

5. Suppose that we want to sort the list in reverse chronological order (by release date). We can do it like this:

```
sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
```

The `--key` option allows specification of offsets within fields. So `3.7` means start sorting from the 7-th character of the 3-rd field, which is the year. The modifier `n` makes it a numerical sort, `r` does reverse sorting, and with `b` we are suppressing any leading spaces of the third field.

In a similar way, the second sort key `3.1` sorts by the month, and the third key `3.4` sorts by day.

6. Some files don't use tabs and spaces as delimiters, for example the file `/etc/passwd`:

```
head /etc/passwd
```

In this case we can use the option `-t` to define the field separator character. For example to sort `/etc/passwd` on the seventh field (the account's default shell), we could do this:

```
sort -t ':' -k 7 /etc/passwd | head
```

DOWNLOAD LESSON07/PART1.CAST

2. cut

1. The command `cut` extracts a certain column (field) from the input, for example:

```
cut -f 3 distros.txt
```

```
cut -f 1,3 distros.txt
```

```
cut -f 1-2,3 distros.txt
```

2. If we want to extract only the year, we can do it like this:

```
cut -f 3 distros.txt | cut -c 7-10
```

   The option `-c` tell `cut` to extract from the line characters, instead of fields (as if each character is a field).

```
cut -f 3 distros.txt | cut -c 7-10,1-2,4-5
```

3. Another way to get the year would be like this:

```
expand distros.txt | cut -c 23-
```

   The command `expand` replaces tabs by the corresponding number of spaces, so that the year would always start at the position 23.

4. When working with fields, it is possible to specify a different field delimiter, instead of the tab. For example:

```
head /etc/passwd
```

```
cut -d ':' -f 1 /etc/passwd | head
```

Here we extract the first field from `/etc/passwd`.

DOWNLOAD LESSON07/PART2.CAST

3. paste

The `paste` command does the opposite of `cut`. Rather than extracting a column of text from a file, it adds one or more columns of text to a file.

To demonstrate how `paste` operates, we will perform some surgery on our `distros.txt` file to produce a chronological list of releases.

1. First let's sort distros by date:

```
head distros.txt
```

```
sort -k 3.7nbr -k 3.1nbr -k 3.4nbr \
    distros.txt > distros-by-date.txt
```

```
head distros-by-date.txt
```

2. Next, let's use `cut` to extract the first two fields/columns from the file (the distro name and version):

```
cut -f 1,2 distros-by-date.txt > distros-versions.txt
```

```
head distros-versions.txt
```

   Let's also extract the release dates:

```
cut -f 3 distros-by-date.txt > distros-dates.txt
```

```
```

```
head distros-dates.txt
```

3. To complete the process, let's use `paste` to put the column of dates ahead of distro names and versions, thus creating a chronological list:

```
paste distros-dates.txt distros-versions.txt \
    > distros-chronological.txt
```

```
head distros-chronological.txt
```

> **DOWNLOAD LESSON07/PART3.CAST**

4. join

The command `join`, like `paste`, adds columns to a file. However it does it in a way that is similar to the *join* operation in *relational databases*. It joins data from multiple files based on a shared key field.

1. To demonstrate `join` let's make a couple of files with a shared key. The first file will contain the release dates and the release names:

```
cut -f 1,1 distros-by-date.txt > distros-names.txt
```

```
paste \
    distros-dates.txt \
    distros-names.txt \
    > distros-key-names.txt
```

```
head distros-key-names.txt
```

2. The second file will contain the release dates and the version numbers:

```
cut -f 2,2 distros-by-date.txt > distros-vernums.txt
```

```
paste \
    distros-dates.txt \
    distros-vernums.txt \
    > distros-key-vernums.txt
```

```
head distros-key-vernums.txt
```

3. Both of these files have the release date as a common field. Let's join them:

```
join \
    distros-key-names.txt \
    distros-key-vernums.txt \
    | head
```

It is important that the files must be sorted on the key field for `join` to work properly.

DOWNLOAD LESSON07/PART4.CAST

5. Comparing text

1. Let's create two test files:

```
cat > file1.txt <<EOF
a
b
c
d
EOF
```

```
cat > file2.txt <<EOF
b
c
d
e
EOF
```

2. We can compare them with `comm`:

```
comm file1.txt file2.txt
```

```
comm -12 file1.txt file2.txt
```

   In this case we are suppressing the columns 1 and 2.

3. A more complex tool is `diff`:

```
diff file1.txt file2.txt
```

   With context:

```
diff -c file1.txt file2.txt
```

Unified format is more concise:

```
diff -u file1.txt file2.txt
```

4. To create a *patch* file usually the options `-Naur` are used:

```
diff -Naur file1.txt file2.txt > patchfile.txt
```

```
cat patchfile.txt
```

We can use the command `patch` to apply a patch file:

```
patch file1.txt patchfile.txt
```

```
cat file1.txt
```

Now `file1.txt` has the same content as `file2.txt`.

> **DOWNLOAD LESSON07/PART5.CAST**

6. Editing on the fly

1. The `tr` program is used to transliterate characters:

   ```
   echo "lowercase letters" | tr a-z A-Z
   ```

   Multiple characters can be converted to a single character:

   ```
   echo "lowercase letters" | tr [:lower:] A
   ```

   With the option `-d` it can delete characters:

   ```
   echo "lowercase letters" | tr -d e
   ```

   With the option `-s` it can squeeze repeated characters:

   ```
   echo "aaabbbccc" | tr -s ab
   ```

   ```
   echo "abcabcabc" | tr -s ab
   ```

2. As another example, let's use `tr` to implement ROT13 encoding (where each character is moved 13 places up the alphabet):

   ```
   echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
   ```

   To decode, perform the same translation a second time:

   ```
   echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
   ```

3. The program `sed` means *stream editor*.

```
echo "front" | sed 's/front/back/'
```

The command `s` stands for *substitute*.

```
echo "front" | sed 's_front_back_'
```

The character immediately after `s` becomes the delimiter.

The commands in `sed` can be preceded by an address:

```
echo "front" | sed '1s/front/back/'
```

```
echo "front" | sed '2s/front/back/'
```

4. Let's try some more examples on `distros.txt`:

```
sed -n '1,5p' distros.txt
```

The option `-n` does not print lines by default, and the `p` command prints only the lines in the given range.

```
sed -n '/SUSE/p' distros.txt
```

Prints only the lines that match the given regular expression.

```
sed -n '/SUSE/!p' distros.txt
```

Prints only the lines that do not match.

5. The command `s` substitutes by default only the first occurrence on a matching line:

   ```
   echo "aaabbbccc" | sed 's/b/B/'
   ```

   We can append the modifier `g` (global) to replace all the occurrences:

   ```
   echo "aaabbbccc" | sed 's/b/B/g'
   ```

6. Let's change the date format from MM/DD/YYYY to YYYY-MM-DD on `distros.txt`:

   ```
   sed -E 's#([0-9]{2})/([0-9]{2})/([0-9]{4})$#\3-\1-\2#' distros.txt
   ```

   First, we are using the option `-E, --regexp-extended` because there are lots of special characters like `(`, `)`, `{`, `}` in the regular expression and escaping all of them by a `\` would make it really messy and unreadable.

   Then, we are enclosing in parentheses the parts of the regexp that match the month (`[0-9]{2}`), the day (`[0-9]{2}`) and the year (`[0-9]{4}`).

   The strings that are matched by a subexpression can be used in the replacement like this: `\n`, where `n` is the number of the matching subexpression (pair of parentheses).

7. It is possible to give several commands to the same `sed` program, like this:

   ```
   echo "aaabbbccc" | sed -e 's/b/B/g' -e 's/a/A/g'
   ```

   However, sometimes it is preferable to list these commands in a `sed` script file, and call this script instead. For example:

   ```
   
   ```

```
cat <<'EOF' > distros.sed
#sed script to produce a distro report

1 i\
\
Linux Distribution Report\

s#([0-9]{2})/([0-9]{2})/([0-9]{4})$#\3-\1-\2#
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
EOF
```

<details>
<summary>download</summary>

```
wget https://linux-cli.fs.al/examples/lesson07/distros.sed
```

</details>

```
cat distros.sed
```

```
sed -E -f distros.sed distros.txt
```

The first line is a comment.

Then, the `i` command inserts something before the first line.

The `s` command changes the date format.

Finally, the `y` command transliterates the lower case characters to upper case, similar to the command `tr`. However, unlike `tr`, it does not recognize character ranges or character classes, so we have to list all the letters of the alphabet.

DOWNLOAD LESSON07/PART6.CAST

7. aspell

1. Plain text files:

```
echo 'The quick brown fox jimped over the lazy dog.' > foo.txt
cat foo.txt
```

```
aspell check foo.txt
```

```
cat foo.txt
```

2. HTML files:

```
cat <<EOF > foo.html
<html>
  <head>
    <title>Mispelled HTML file</title>
  </head>
  <body>
    <p>The quick brown fox jimped over the lazy dog.</p>
  </body>
</html>
EOF
```

download

```
wget https://linux-cli.fs.al/examples/lesson07/foo.html
```

```
cat foo.html
```

```
aspell check foo.html
```

```
cat foo.html
```

If the extenssion of the file is not `.html` we can force the html mode with the `-H` option.

```
aspell --help | less
```

DOWNLOAD LESSON07/PART7.CAST

📄 Intro

- Archiving (gzip, bzip2, tar, zip, rsync)

📄 1. Archiving and backup

1. We can use gzip and bzip2 to compress one or more files:

📄 2. Networking

1. Basic tools:

📄 3. Networking: ssh

1. Another network tool is ssh, which can be used to login to a

📄 4. Filesystems

In this section we will try an example with a CoW (Copy-on-Write)

📄 5. Package Management

<AsciinemaWidget

Intro

- Archiving (gzip, bzip2, tar, zip, rsync)
- Networking (ping, traceroute, ip, netstat, wget, curl, ssh, scp, sftp)
- Filesystems (mount, umount, fdisk, mkfs, dd)
- Package Management (apt)

DOWNLOAD LESSON08/INTRO.CAST

1. Archiving and backup

1. We can use `gzip` and `bzip2` to compress one or more files:

```
ls -l /etc > foo.txt
```

```
ls -lh foo.*
```

```
gzip foo.txt
```

```
ls -lh foo.*
```

```
gunzip foo.txt
```

```
ls -lh foo.*
```

```
ls -l /etc | gzip > foo.txt.gz
```

```
gunzip -c foo.txt.gz
```

```
zcat foo.txt.gz | less
```

```
zless foo.txt.gz
```

```
bzip2 foo.txt
```

```
ls -lh foo.*
```

```
bunzip2 foo.txt.bz2
```

```
ls -lh foo.*
```

2. We can use `tar` to archive files.

   Let's create a test directory:

   ```
   mkdir -p testdir/dir-{001..100}
   ```

   ```
   touch testdir/dir-{001..100}/file-{A..Z}
   ```

   ```
   ls testdir/
   ```

   ```
   ls testdir/dir-001/
   ```

   Create a tar archive of the entire directory:

   ```
   tar -c -f testdir.tar testdir
   ```

   ```
   tar -cf testdir.tar testdir
   ```

   ```
   ```

```
ls -lh
```

The option `-c` means `create`, and the option `-f` is for the filename of the archive.

The option `-t` is used to list the contents of the archive, and `-v` is for verbose:

```
tar -tf testdir.tar | less
```

```
tar -tvf testdir.tar | less
```

Now let's extract the archive in a new location:

```
mkdir foo
```

```
cd foo
```

```
tar -xf ../testdir.tar
```

```
ls
```

```
tree -C | less -r
```

```
cd .. ; rm -rf foo/
```

3. By default, `tar` removes the leading `/` from absolute filenames:

```
```

```
echo $(pwd)/testdir
```

```
tar cf testdir2.tar $(pwd)/testdir
```

```
tar tf testdir2.tar | less
```

```
mkdir foo
```

```
tar xf testdir2.tar -C foo/
```

```
tree foo -C | less -r
```

```
rm -rf foo
```

4. We can extract only some files from the archive (not all the files):

```
mkdir foo
```

```
cd foo
```

```
tar tf ../testdir.tar testdir/dir-001/file-A
```

```
tar xf ../testdir.tar testdir/dir-001/file-A
```

```
tree
```

```
tar xf ../testdir.tar testdir/dir-002/file-{A,B,C}
```

```
tree
```

We can also use `--wildcards`, like this:

```
tar xf ../testdir.tar --wildcards 'testdir/dir-*/file-A'
```

```
tree -C | less -r
```

```
cd .. ; rm -rf foo
```

5. Sometimes it is useful to combine `tar` with `find` and `gzip`:

```
find testdir -name 'file-A'
```

```
find testdir -name 'file-A' \
     -exec tar rf testdir3.tar '{}' '+'
```

```
tar tf testdir3.tar | less
```

```
find testdir -name 'file-B' \
     -exec tar rf testdir3.tar '{}' '+'
```

```
tar tf testdir3.tar | less
```

The option 'r' is for appending files to an archive.

```
find testdir -name 'file-A' \
    | tar cf - -T - \
    | gzip > testdir.tgz
```

The first `-` makes tar to send the output to *stdout* instead of a file. The option `-T` or `--files-from` includes in the archive only the files listed in the given file. In this case we are reading the list of files from `-`, which means the *stdin* and is the list of files coming from the command `find`. Then we are passing the output of `tar` to `gzip` in order to compress it.

We can also use the options `z` or `j` to compress the archive:

```
find testdir -name 'file-A' \
    | tar czf testdir.tgz -T -
```

```
find testdir -name 'file-A' \
    | tar cjf testdir.tbz -T -
```

```
ls -lh
```

The option `j` uses `bzip2` compression, instead of `bzip`.

6. The `zip` program is both a compression tool and an archiver:

```
zip -r testdir.zip testdir
```

```
ls -lh
```

The option `-r` is for recursion.

```
mkdir -p foo
```

```
cd foo
```

```
unzip ../testdir.zip
```

```
tree | less
```

```
unzip -l ../testdir.zip testdir/dir-007/file-*
```

```
rm -rf testdir
unzip ../testdir.zip testdir/dir-007/file-*
tree | less
```

```
cd .. ; rm -rf foo
```

7. We can use `rsync` to synchronize files and directories:

```
rsync -av testdir foo
```

```
ls foo
```

```
rsync -av testdir foo
```

Notice that in the second case no files are copied because `rsync` detects that there are no differences between the source and the destination.

```
touch testdir/dir-099/file-Z
```

```
rsync -av testdir foo
```

With the option `--delete` we can also delete the files on the destination directory that are not present on the source directory.

```
rm testdir/dir-099/file-Z
```

```
rsync -av testdir foo
```

```
ls foo/testdir/dir-099/file-Z
```

```
rsync -av --delete testdir foo
```

```
ls foo/testdir/dir-099/file-Z
```

`rsync` can be used over the network as well, usually combined with `ssh`.

DOWNLOAD LESSON08/PART1.CAST

2. Networking

1. Basic tools:

```
ip address
```

```
ip addr
```

```
ip a
```

```
ip addr show lo
```

```
ip route
```

```
ip r
```

```
ping -c 3 8.8.8.8
```

```
dig linuxcommand.org
```

```
dig linuxcommand.org +short
```

```
ping -c 3 linuxcommand.org
```

`traceroute linuxcommand.org`

`tracepath linuxcommand.org`

2. For downloading files we can use `wget` or `curl`:

```
wget http://linuxcommand.org/index.php
```

```
less index.php
```

```
wget -O index.html 'http://linuxcommand.org/index.php'
```

```
less index.html
```

```
curl http://linuxcommand.org/index.php
```

```
curl http://linuxcommand.org/index.php > index.html
```

3. Netcat is a simple tool for network communication.

   Let's use it to listen to the port `12345`:

   ```
   nc -l 12345
   ```

   Open another terminal and connect to the same port like this:

   ```
   nc localhost 12345    # on the second terminal
   ```

   Now, any line that you type here is sent and displayed to the other terminal:

   ```
   Hello network
   ```

```
The quick brown fox jumped over the internet
```

Check the other terminal.

Interrupt them with `Ctrl-c`.

This may not seem very impressive, but instead of `localhost` we could have used a real server name or IP and connect to it remotely. It may be used to check that the TCP port `12345` on the server is accessible from the client (in case that there is a firewall, for example).

For checking a UDP port we can add the option `-u` to both of these commands.

It can also be used as a simple tool for file transfer:

```
nc -l 12345 > file.txt    # on the first terminal
```

```
nc -w 3 localhost 12345 < /etc/passwd    # on the second terminal
```

```
ls file.txt    # on the first terminal
```

```
cat file.txt    # on the first terminal
```

As another example, let's combine it with `tar` to transfer a whole directory:

```
mkdir cptest
```

```
cd cptest
```

```

```

```
nc -l 12345 | tar xzpf -
```

Switch to the second terminal.

```
cd testdir
ls
```

```
tar czpf - . | nc -w 3 localhost 12345
```

Switch to the first terminal.

```
ls
```

```
cd ..
```

```
rm -rf cptest
```

DOWNLOAD LESSON08/PART2.CAST

3. Networking: ssh

1. Another network tool is `ssh`, which can be used to login to a remote system, execute commands remotely, and more.

   First let's create a user account:

   ```
   sudo useradd -m -s /bin/bash user01
   ```

   ```
   echo user01:pass01 | sudo chpasswd
   ```

   Let's login to it:

   ```
   ssh user01@localhost
   ```

   ```
   ls -al
   ```

   ```
   exit
   ```

   We can also use `ssh` to just run a command remotely:

   ```
   ssh user01@localhost ls -al
   ```

   ```
   ssh user01@localhost whoami
   ```

   ```
   ssh user01@localhost ls .*
   ```

```
ssh user01@localhost 'ls .*'
```

2. Writing a password each time that we use `ssh` quickly becomes tedious. We can use keys instead, which is easier and more secure.

   First let's generate a public/private key pair:

   ```
   ssh-keygen --help
   ```

   ```
   ssh-keygen -t ecdsa -q -N '' -f ~/.ssh/key1
   ```

   The option `-N ''` makes it generate a key that does not have a passphrase.

   ```
   ls -al ~/.ssh/key1*
   ```

   ```
   cat ~/.ssh/key1
   ```

   ```
   cat ~/.ssh/key1.pub
   ```

   In order to be able to login to the server with this key, we need to send the public part of it to the server:

   ```
   ssh-copy-id -i ~/.ssh/key1.pub user01@localhost
   ```

   Now let's try to login using the private key as an identity file:

   ```
   ssh -i ~/.ssh/key1 user01@localhost
   ```

   ```
   
   ```

```
ls -al
```

```
cat .ssh/authorized_keys
```

```
exit
```

```
cat ~/.ssh/key1.pub
```

You may notice that the public key has been appended to `.ssh/authorized_keys` on the server.

It gets even better. Let's add this configuration to `~/.ssh/config`:

```
cat <<EOF >> ~/.ssh/config
Host server1
    HostName 127.0.0.1
    User user01
    IdentityFile ~/.ssh/key1
EOF
```

```
cat ~/.ssh/config
```

Now we can just use `ssh` with the name `server1`, without having to specify the hostname (or IP) of the server, the username, the identity file etc. It will get them automatically from the config file.

```
ssh server1
```

```
exit
```

```
ssh server1 whoami
```

3. Using `scp`, `sftp`, `rsync` etc.

   All these tools use an SSH tunnel for a secure communication with the server. Now that we have an easy `ssh` access to the server, we can also use easily these tools:

```
touch foo.txt
```

```
scp foo.txt server1:
```

```
ssh server1 ls -l
```

```
ssh server1 touch bar.txt
```

```
ssh server1 ls -l
```

```
scp server1:bar.txt .
```

```
ls -l bar.txt
```

`sftp`:

```
sftp server1
```

```
ls
```

```
help
```

```
quit
```

`rsync`:

```
ls testdir
```

```
rsync -av testdir server1:
```

```
ssh server1 ls
```

```
ssh server1 ls testdir
```

> **DOWNLOAD LESSON08/PART3.CAST**

4. Filesystems

In this section we will try an example with a CoW (Copy-on-Write) filesystem. CoW filesystems have the nice property that it is possible to "clone" files instantly, by having the new file refer to the old blocks, and copying (possibly) changed blocks. This both saves time and space, and can be very beneficial in a lot of situations (for example when working with big files). In Linux this type of copy is called "reflink". We will see how to use it on the XFS filesystem.

1. Create a virtual block device

Linux supports a special block device called the loop device, which maps a normal file onto a virtual block device. This allows for the file to be used as a "virtual file system".

> **INFO**
> Loop devices are not available on a container, so we need a real machine or a VM, in order to try this example.

1. Create a file of size 1G:

```
fallocate -l 1G disk.img
```

```
ls -hl disk.img
```

```
du -hs disk.img
```

2. Create a loop device with this file:

```
sudo losetup -f disk.img
```

The option `-f` finds an unused loop device.

3. Find the name of the loop device that was created:

```
losetup -a
```

```
losetup -a | grep disk.img
```

```
lodevice=$(losetup -a | grep disk.img | cut -d: -f1)
```

2. Create an XFS filesystem

1. Make sure that the package `xfsprogs` is installed:

```
sudo apt install xfsprogs
```

2. Create an XFS filesystem on the image file:

```
mkfs.xfs -m reflink=1 -L test disk.img
```

The metadata `-m reflink=1` tells the command to enable reflinks, and `-L test` sets the label of the filesystem.

3. Create a directory:

```
mkdir mnt
```

4. Mount the loop device on it:

```
# sudo mount /dev/loop0 mnt
sudo mount $lodevice mnt
```

```
mount | grep mnt
```

5. Check the usage of the filesystem:

```
df -h mnt/
```

   Notice that only **40M** are used from it.

3. Copy files with '--reflink'

1. Create for testing a file of size 100 MB (with random data):

```
cd mnt/
sudo chown $(whoami): .
```

```
dd if=/dev/urandom of=test bs=1M count=100
```

2. Check that now there are **140M** of disk space used:

```
df -h .
```

3. Create a copy of the file (with reflinks enabled):

```
cp -v --reflink=always test test1
```

4. Check the size of each file:

```
ls -hsl
```

Each of them is **100M** and in total there are **200M** of data.

5. However if we check the disk usage we will see that both of them still take on disk the same amount of space as before **140M**:

```
df -h .
```

This shows the space-saving feature of reflinks. If the file was big enough, we would have noticed as well that the reflink copy takes no time at all, it is done instantly.

4. Clean up

1. Unmount and delete the test directory `mnt/`:

```
cd ..
sudo umount mnt/
rmdir mnt/
```

2. Delete the loop device:

```
losetup -a
# sudo losetup -d /dev/loop0
sudo losetup -d $lodevice
```

3. Remove the file that was used to create the loop device:

```
rm disk.img
```

> **DOWNLOAD LESSON08/PART4.CAST**

5. Package Management

```
sudo apt update
```

```
sudo apt upgrade
```

```
apt list 'emacs*'
```

```
apt show emacs
```

```
sudo apt install emacs
```

```
emacs
```

```
sudo apt remove emacs
```

DOWNLOAD LESSON08/PART5.CAST

🗃️ Lesson 9

5 items

🗃️ Lesson 10

6 items

🗃️ Lesson 11

5 items

🗃️ Lesson 12

6 items

🗃️ Lesson 13

5 items

🗃️ Lesson 14

4 items

🗃️ Lesson 15

4 items

📄 Intro

Before starting with bash scripting (in the next lesson), we are going

📄 1. Vim basics

1. Starting and quitting.

📄 2. More Vim commands

2.1 Deleting text

📄 3. Editing multiple files

1. It is often useful to edit more than one file at a time.

📄 4. Tutorials

1. There is a Vim tutorial that can be started with:

Intro

Before starting with bash scripting (in the next lesson), we are going to make a quick introduction to the editors Vim and Emacs.

DOWNLOAD LESSON09/INTRO.CAST

1. Vim basics

1. Starting and quitting.

   ```
   vim
   ```

   The tilde char (`~`) at the start of a line means that there is no text on that line.

   To quit press `:q`

2. Editing modes.

   Let's start it again, passing to it the name of a nonexistent file:

   ```
   rm -f foo.txt
   ```

   ```
   vim foo.txt
   ```

   Vim has a *command mode* and an *insert mode*. In the *command mode* the keys that we type are interpreted as commands. In the *insert mode* we can add text to the file.

   When it starts up, Vim is in the *command mode*. To switch the mode to *insert* let's give the command: `i`

   Notice the status `-- INSERT --` at the bottom.

   Now let's enter some text:

   ```
   The quick brown fox jumped over the lazy dog.
   ```

To exit the *insert* mode and return to *command* mode, press: ESC

To save the file type: `:w`

3. Moving the cursor around.

   While in the command mode, we can move with the keys:

   - `h` -- left
   - `l` -- right
   - `j` -- down
   - `k` -- up

   Press a few times `h` and `l`.

   We can also use:

   - `0` -- to the beginning of line
   - `$` -- to the end of line
   - `w` -- to the beginning of next word or punctuation char
   - `W` -- to the beginning of next word (ignore punctuation)
   - `b` -- backwards one word or punctuation char
   - `B` -- backwards one word (ignore punctuations)

   Try them a few times.

   If a command is prefixed by a number, it will be repeated that many times. For example `3w` is the same as pressing `w` 3 times.

   We can also use the arrows (left, right, up, down).

4. Basic editing.

   With the command `i` we start inserting text at the current position of the cursor. To append text after the current position we can use the command `a`.

   Type `A` to start appending text at the end of the line.

Now type: `. It was cool.`

Press Enter and continue by adding these lines:

```
Line 2
Line 3
Line 4
Line 5
```

Then press ESC to switch back to command mode, then `:w` to save (write to file).

- Go to the first line: `1G`
- Go to the last line: `G`
- Go to the third line: `3G`

Then:

- Open a new line below the current one: `o`
- Undo: ESC and `u`
- Open a new line above the current one: `0`
- Undo: ESC and `u`

DOWNLOAD LESSON09/PART1.CAST

2. More Vim commands

2.1 Deleting text

1. To delete text in Vim we can use the commands `x`, which deletes the character at the cursor, and `d`.

   `1G5w`

2. Press `x` a few times to delete a few characters, then press `u` several times to undo.

   Press `3x` to delete 3 chars, then `u` to undo.

3. Try also these and see what they do:

   - `dW` and `u`
   - `5dW` and `u`
   - `d$` and `u`
   - `d0` and `u`
   - `dd` and `u`
   - `3dd` and `u`
   - `dG` and `u`
   - `d4G` and `u`

2.2 Cut, copy and paste

1. When we delete some text, it is actually like *cut*, because the deleted part is placed on a *paste* buffer and can be placed somewhere else. To paste it after the cursor we can use the command `p`, to paste it before the cursor we can use capital `P`.

2. Try:

`5x` and `p`, then `uu`

`3x` and `P`, then `uu`

`5dw`, `$`, `p`, `uu`

`d$`, `0`, `p`, `u`, `P`, `uu`

`dd`, `p`, `u`, `P`, `uu`

3. Instead of `d` we can use the command `y` (yank) to copy text.

   `yw`, `p`, `u`, `P`, `u`

   `5yw`, `P`, `u`

   `yy`, `p`, `u`

   `3yy`, `P`, `u`, `p`, `u`

   `yG`, `P`, `u`

   `y$`, `0`, `P`, `u`

4. To join a line with the next one we can use `J`:

   `3G`, `J`, `J`, `uu`

2.3 Search and replace

1. To find a character in the current line, press `f` and the character:

   `1G`, `fa`, `;`

2. To move the cursor to the next occurrence of a word or phrase, the `/` command is used. Type a word or phrase after it and then Enter:

   `/` then type `Line`

To find the next match press `n`:

`n`, `n`, `n`

3. To replace (substitute) `Line` by `line` in the whole file use:

```
:% s/Line/line/g
```

It is similar to the `sed` command. It starts with a range of lines where to perform the substitution. In this example, `%` denotes the whole file and is the same as `1,$` (from the first line to the last one).

4. We can also ask for confirmation by adding the modifier `c` at the end:

```
:1,$ s/line/Line/gc
```

The confirmation options are:

- `y` -- yes
- `n` -- no
- `a` -- all
- `q` -- quit
- `l` -- last (replace this one and quit)
- `Ctrl-e` / `Ctrl-y` -- scroll down and up, to see the context

DOWNLOAD LESSON09/PART2.CAST

3. Editing multiple files

1. It is often useful to edit more than one file at a time.

   Quit from vim:

   ```
   :q!
   ```

   Let's create another test file:

   ```
   ls -l /usr/bin > ls-output.txt
   ```

   Start vim with both test files as argument:

   ```
   vim foo.txt ls-output.txt
   ```

   To see the list of buffers (opened files):

   ```
   :buffers
   ```

   To switch to the next buffer press `:bn`. Try it a few times

   To switch to the previous buffer press `:bp`. Try it a few times.

   We can also switch to another buffer like this:

   ```
   :buffer 2
   ```

   ```
   :buffer 1
   ```

2. It's also possible to add files to the current editing session.

```
:q!
```

```
vim foo.txt
```

```
:e ls-output.txt
```

```
:buffers
```

3. While editing multiple files, it is possible to copy a portion of one file into another file that we are editing. This is easily done using the usual yank and paste commands.

`:buffer 1`

`1G` and `yy`

`:buffer 2`

`1G` and `p`

`:q!`

4. It's also possible to insert an entire file into one that we are editing.

```
vim ls-output.txt
```

```
3G
```

```
:r foo.txt
```

```
:w foo1.txt
```

We saved the buffer to the file `foo1.txt`, but we are still editing the first file and any further modifications will be done to it.

```
:q!
```

DOWNLOAD LESSON09/PART3.CAST

4. Tutorials

1. There is a Vim tutorial that can be started with:

   ```
   vimtutor
   ```

2. For an Emacs tutorial, first start emacs with:

   ```
   emacs -nw
   ```

   Then press `Ctrl-h` and `t`. Or move (by the down arrow) to `Emacs tutorial` and then press Enter.

3. Some other online tutorials:

   - https://openvim.com/
   - https://www.gnu.org/software/emacs/tour/

DOWNLOAD LESSON09/PART4.CAST

📄 Intro

In this lesson we will see:

📄 1. Create and run a script

1. Let's create a script that prints "Hello World!".

📄 2. Starting a project

Before starting, let's get first some [example

📄 3. Variables and constants

1. Variables in bash don't have to be declared, we just use them:

📄 4. Here documents

A here document is an additional form of I/O redirection in which we

📄 5. Shell functions

Functions can be declared in one of these two forms, which are

Intro

In this lesson we will see:

- how to create and run a bash script
- how to output some text from a bash script
- bash script variables, constants and functions

DOWNLOAD LESSON10/INTRO.CAST

1. Create and run a script

1. Let's create a script that prints "Hello World!".

   ```
   vim hello.sh
   ```

   Press `i` then type this code:

   ```
   echo "Hello World!"
   echo "This is the first script."
   ```

   Press ESC then type `:wq` to save and quit.

   ```
   ls -l hello.sh
   ```

   ```
   cat hello.sh
   ```

   ```
   cat hello.sh | bash
   ```

   ```
   bash hello.sh
   ```

   We are sending it to `bash`, and bash is interpreting and executing the commands inside the script.

2. We can tell the shell to use Bash for interpreting this script by adding `#!/bin/bash` as the first line of the script.

   ```
   vim hello.sh
   ```

Press `1G`, `0`, and then type `#!/bin/bash`.

Press ESC, `:wq` and Enter, to save and quit.

```
cat hello.sh
```

The script now should look like this:

```
#!/bin/bash
echo "Hello World!"
echo "This is the first script."
```

<details>
<summary>The shebang</summary>

If it was a Python script, we would have used instead the line `#!/usr/bin/python3` to tell the shell that it should use Python for interpreting this script.

The character `#` is usually called *hash*, and `!` is usually called *bang*. Together they are called *shebang* and they are placed at the very beginning of a script (no empty lines and no empty spaces before them). After the *shebang* comes the program that the shell should use to interpret the script.
</details>

3. Let's try to execute it:

```
hello.sh
```

It says `command not found`. This is because the shell looks for this command in certain directories, which are listed on the environment variable `PATH`:

```
echo $PATH
```

There is no command `hello.sh` in any of these directories, so shell cannot find such a command.

To fix this problem, we can tell bash the path of the command, like this:

```
./hello.sh
```

When we give a path to the command (`./` in this case), the shell does not use the variable `PATH` but tries to find the command in the given path.

<details>
<summary>Modifying <code>PATH</code></summary>

Another way for fixing the problem is to add the current directory to the `PATH`, like this:

```
echo $PATH
PATH="$(pwd):$PATH"
echo $PATH
```

Then the shell will be able to find `hello.sh` even if we don't specify its path:

```
hello.sh
```

</details>

4. Now, when we try to execute the script, it gives the error message `Permission denied`, because the script is not executable. Let's fix this by giving it the `x` permission, and try again:

```
ls -l hello.sh
chmod +x hello.sh
ls -l hello.sh
```

```
./hello.sh
```

5. In bash, comments are denoted by a `#`. Everything after a `#` is considered a comment and is ignored:

```
ls -l hello.sh   # this is a comment and will be ignored

# This is also a comment.
```

Let's modify the script by adding some comments, and execute it again, to verify that the comments are just ignored by the interpreter.

```
vim hello.sh
```

Type `1G`, `o` and then enter `# This is a comment.` on the second line.

Press ESC, `j`, `A` and append `# this is another comment` on the third line.

Press ESC, then `:wq` and Enter, to save and quit.

```
cat hello.sh
```

The script now should look like this:

```
#!/bin/bash
# This is a comment.
echo "Hello World!" # this is another comment
echo "This is the first script."
```

Let's execute it and make sure that the comments are just ignored:

```
./hello.sh
```

DOWNLOAD LESSON10/PART1.CAST

2. Starting a project

> **NOTE**
>
> Before starting, let's get first some example files:

```
mkdir -p examples
cd examples/

wget https://linux-cli.fs.al/examples/lesson10.tgz

tar xfz lesson10.tgz

cd lesson10/
ls
```

We will start to write a script that generates a report about various status and statistics of the system. This report will be in the HTML format.

1. The first step is to write a program (script) that outputs a basic HTML page to the standard output. An example of a basic HTML page is on the file `page.html`:

   ```
   cat page.html
   ```

   ```
   lynx page.html
   ```

   Type `q` and `y` to quit.

   ```
   mv page.html sys_info.sh
   ```

   ```
   
   ```

```
vim sys_info.sh
```

```
:1,$ s/^/echo "/
```

```
:% s/$/"/
```

Type `1G`, then capital `O`, and write these lines:

```
#!/bin/bash

# Program to output a system information page
```

Press ESC and type `:wq`.

```
chmod +x sys_info.sh
```

```
./sys_info.sh
```

```
./sys_info.sh > sys_info.html
```

```
lynx sys_info.html
```

Type `q` and `y` to quit.

2. We can make this script more simple and clear by using a single `echo`:

```
vim sys_info.sh
```

```
```

```
:6,$ s/echo "//
```

```
:5,$-1 s/"$//
```

```
:wq
```

```
./sys_info.sh
```

A quoted string may contain newlines, and therefore contain multiple lines of text.

3. Let's put some data on the report:

```
vim sys_info.sh
```

```
:% s/Page Title/System Information Report/
```

```
:% s#Page body#<h1>System Information Report</h1>#
```

```
:wq
```

```
./sys_info.sh
```

4. We can use a variable to avoid the repetition of the text "System Information Report":

```
vim sys_info.sh
```

```
:% s/System Information Report/$title/
```

```
/echo
```

Press capital `O` to open a new line above and write:

```
title="System Information Report"
```

Press ESC and type `:wq`.

```
./sys_info.sh
```

> **DOWNLOAD LESSON10/PART2.CAST**

3. Variables and constants

1. Variables in bash don't have to be declared, we just use them:

```
foo="yes"
```

```
echo $foo
```

We have a shell expansion here, and it is the same as: `echo yes`

```
echo $foo1
```

It is the first time that the shell sees the variable `foo1`, however it does not complain but just creates it and gives it an empty value. This means that we should be careful with spelling the names of the variables, otherwise we may get strange results.

```
touch foo.txt
```

```
foo=foo.txt
```

```
foo1=foo1.txt
```

```
cp $foo $fool
```

We have misspelled the second argument, so shell expands it to an empty string and we get an error from `cp`.

2. To denote constants in bash, we use uppercase variable names, by convention:

```
vim sys_info.sh
```

```
:% s/$title/$TITLE/g
```

```
:% s/title=/TITLE=/
```

```
:/^TITLE=/ s/Report/Report for $HOSTNAME/
```

```
:wq
```

```
./sys_info.sh
```

We have also used the environment variable `HOSTNAME`. Environment variables are considered as constants, so they are in uppercase.

Actually, there is a way to make sure that a variable cannot be changed (is a constant), although it is not used frequently:

```
sed -i sys_info.sh -e 's/TITLE=/declare -r TITLE=/'
```

```
cat sys_info.sh
```

```
highlight -O xterm256 sys_info.sh
```

```
./sys_info.sh
```

The option `-r` of `declare` means "read-only". So, we cannot assign a value again to this variable.

3. When a value is assigned to a variable there should be no spaces around the equal sign:

```
a=z
```

```
echo $a
```

Shell expansions are applied to the value, before assignment:

```
b="a string"
```

```
c="a string and $b"
```

```
echo $c
```

```
d=$(ls -l foo.txt)
```

```
echo $d
```

```
e=$((5 * 7))
```

```
echo $e
```

```
f="\t\ta string\n"
```

```
echo $f
```

```
echo -e $f
```

```
help echo
```

Multiple assignments may be done on a single line:

```
a=5 b="a string"
```

```
echo $a $b
```

4. During expansion, variable names may be surrounded by curly braces `{}`, which are necessary in some cases. For example:

```
filename="myfile"
```

```
touch $filename
```

```
mv $filename $filename1
```

What we want is to rename the file to `myfile1`, but the shell is interpreting `filename1` as a variable, which of course has not been assigned yet and is empty. We should do it like this

```
mv $filename ${filename}1
```

```
```

```
ls -l myfile1
```

5. Let's add a timestamp to the report, using variables/constants:

```
date +"%x %r %Z"
```

```
echo $USER
```

```
vim sys_info.sh
```

```
/TITLE=
```

Type `o` and enter below:

```
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated on $CURRENT_TIME, by $USER"
```

Press ESC and then `:w` to save.

```
/<h1>
```

Type `Yp` to copy and paste the current line.

```
:s/h1/p/g
```

```
:s/TITLE/TIMESTAMP/
```

```
:wq
```

```
./sys_info.sh
```

6.

TIP

The script so far should look like this:

```
#!/bin/bash

# Program to output a system information page.

declare -r TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated on $CURRENT_TIME, by $USER"
echo "<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
    </body>
</html>"
```

The output of the script should look like this:

```
<html>
    <head>
        <title>System Information Report for linuxmint</title>
    </head>
    <body>
        <h1>System Information Report for linuxmint</h1>
        <p>Generated on 09/28/23 07:21:03 AM UTC, by
```

```
dashamir</p>
    </body>
</html>
```

DOWNLOAD LESSON10/PART3.CAST

4. Here documents

A *here document* is an additional form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command. It works like this:

```
command << token
. . . . .
text
. . . . .
token
```

where *command* is a command that accepts standard input and *token* is a string used to indicate the end of the embedded text. It should be at the beginning of the line and should have no trailing spaces.

1. Let's modify the script to use a *here document*:

   ```
   vim sys_info.sh
   ```

   ```
   /echo
   ```

   Press capital `O` and type:

   ```
   cat << _EOF_
   ```

   Press ESC and then `G` and `o` to go to the end of the buffer and open a new line. Then type:

   ```
   _EOF_
   ```

Press ESC and give this substitute command:

```
:%s/echo "//
```

```
Gk$x
```

```
:wq
```

The script now should look like this:

```
#!/bin/bash

# Program to output a system information page.

declare -r TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated on $CURRENT_TIME, by $USER"
cat << _EOF_
<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
    </body>
</html>
_EOF_
```

```
./sys_info.sh
```

Instead of using `echo`, the script now uses `cat` and a *here document*.

2. The advantage of a *here document* is that inside the text we can freely use single and double quotes, since they are not interpreted by the shell as delimiters of a string. For example:

```
foo="some text"
```

```
cat << EOF
$foo
"$foo"
'$foo'
\$foo
EOF
```

   The shell treats the quotation marks as ordinary characters.

3. We also notice that the variables inside the text are expanded. To prevent variable expansion we can enclose the token in quotes:

```
cat << "EOF"
$foo
"$foo"
'$foo'
\$foo
EOF
```

```
cat << 'EOF'
$foo
"$foo"
'$foo'
\$foo
EOF
```

4. *Here* documents can be used with any command that accepts standard input. For example we can use it with `ftp` to retrieve a file:

ftp1.sh

```
#!/bin/bash

# Script to retrive a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/bookworm/main/installer-
amd64/current/images/cdrom/
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

```
vim ftp1.sh
```

```
:q!
```

```
./ftp1.sh
```

If we change the redirection operator from `<<` to `<<-`, the shell will ignore the leading tab characters in the here document. This allows a here document to be indented, which can improve readability.

ftp2.sh

```
#!/bin/bash

# Script to retrive a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/bookworm/main/installer-
amd64/current/images/cdrom/
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
        open $FTP_SERVER
        user anonymous me@linuxbox
        cd $FTP_PATH
        get $REMOTE_FILE
        bye
        _EOF_

ls -l $REMOTE_FILE
```

```
vim ftp2.sh
```

```
:q!
```

```
./ftp2.sh
```

DOWNLOAD LESSON10/PART4.CAST

5. Shell functions

Functions can be declared in one of these two forms, which are equivalent:

```
function name {
    commands
    return
}
```

```
name () {
    commands
    return
}
```

1. A simple example of a function is in the script `fun.sh`:

    ▼ `fun.sh`

```
#!/bin/bash

# Shell function demo

function step2() {
    echo "Step 2"
    return
}

# Main program starts here

echo "Step 1"
step2
echo "Step 3"
```

```
vim fun.sh
```

```
:q!
```

```
./fun.sh
```

2. Inside a function we can use local variables:

local-vars.sh

```
#!/bin/bash

# Script to demonstrate local variables

foo=0    # global variable foo

funct_1 () {
```

```
    local foo    # variable foo local to func_1

    foo=1
    echo "funct_1: foo = $foo"
}

funct_2 () {
    local foo    # variable foo local to func_2

    foo=2
    echo "funct_2: foo = $foo"
}


echo "global:  foo = $foo"
funct_1
echo "global:  foo = $foo"
funct_2
echo "global:  foo = $foo"
```

```
vim local-vars.sh
```

```
:q!
```

```
./local-vars.sh
```

3. Let's display some additional info on the report page, using functions. We would like to display info about:

    - System uptime and load.
    - Disk space.
    - Home space.

    Let's define a function for each of these:

```
vim sys_info.sh
```

```
/cat
```

Press capital `O` and input:

```
report_uptime () {
    return
}

report_disk_space () {
    return
}

report_home_space () {
    return
}
```

Press ESC and `:w`

Let's call these functions inside the html body:

```
/TIMESTAMP
```

Press lowercase `o` and type:

```
$(report_uptime)
$(report_disk_space)
$(report_home_space)
```

Press ESC and `:wq`

▼ The script should look like this:

```
!/bin/bash

# Program to output a system information page.

declare -r TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated on $CURRENT_TIME, by $USER"

report_uptime() {
    return
}

report_disk_space() {
    return
}

report_home_space() {
    return
}

cat << _EOF_
<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
$(report_uptime)
$(report_disk_space)
$(report_home_space)
    </body>
</html>
_EOF_
```

```
./sys_info.sh
```

▼ The output should look like this:

```
<html>
    <head>
        <title>System Information Report for linuxmint</title>
    </head>
    <body>
        <h1>System Information Report for linuxmint</h1>
        <p>Generated on 09/28/23 03:01:28 PM UTC, by
dashamir</p>



    </body>
</html>
```

4. We see that each function is replaced by an empty line and we don't know what is going on. Let's display some feedback from each function:

```
vim sys_info.sh
```

```
/^report_uptime
```

Press `o` and insert this line:

```
echo "Function report_uptime executed."
```

Press ESC and then search:

```
/^report_disk_space
```

Press `o` and insert this line:

```
echo "Function report_disk_space executed."
```

Press ESC and then search:

```
/^report_home_space
```

Press `o` and insert this line:

```
echo "Function report_home_space executed."
```

Press ESC and `:wq`

<details>
<summary>The functions should look like this:</summary>

```
report_uptime() {
    echo "Function report_uptime executed."
    return
}

report_disk_space() {
    echo "Function report_disk_space executed."
    return
}

report_home_space() {
    echo "Function report_home_space executed."
    return
}
```

</details>

```
./sys_info.sh
```

<details>
<summary>The output should look like this:</summary>

```
<html>
    <head>
        <title>System Information Report for linuxmint</title>
    </head>
    <body>
        <h1>System Information Report for linuxmint</h1>
        <p>Generated on 09/28/23 03:13:29 PM UTC, by
dashamir</p>
Function report_uptime executed.
Function report_disk_space executed.
Function report_home_space executed.
    </body>
</html>
```

</details>

5. Now let's provide the real data:

```
uptime
```

```
df -h .
```

```
du -hs $HOME
```

```
vim sys_info.sh
```

```
/^report_uptime
```

Type: `j dd 0`.

```
cat <<- _EOF_
    <h2>System Uptime</h2>
    <pre>$(uptime)</pre>
    _EOF_
```

> **CAUTION**
>
> Use TAB to indent the lines below `cat`.

Press ESC.

```
/^report_disk_space
```

Type: `j dd 0`.

```
cat <<- _EOF_
    <h2>Disk Space Utilization</h2>
    <pre>$(df -h .)</pre>
    _EOF_
```

Press ESC.

```
/^report_home_space
```

Type: `j dd 0`.

```
cat <<- _EOF_
    <h2>Home Space Utilization</h2>
    <pre>$(du -hs $HOME)</pre>
    _EOF_
```

Press ESC and `:wq`

<details>
<summary>The functions should look like this:</summary>

```
report_uptime() {
    cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
        _EOF_
    return
}

report_disk_space() {
    cat <<- _EOF_
        <h2>Disk Space Utilization</h2>
        <pre>$(df -h .)</pre>
        _EOF_
    return
}

report_home_space() {
    cat <<- _EOF_
        <h2>Home Space Utilization</h2>
        <pre>$(du -hs $HOME)</pre>
        _EOF_
    return
}
```

</details>

```
./sys_info.sh
```

6. Check it in browser:

```
./sys_info.sh > sys_info.html
```

```
lynx sys_info.html
```

Quit with `qy`.

DOWNLOAD LESSON10/PART5.CAST

📄 Intro

In this lesson we will see:

📄 1. Branching with if

The if statement has the following syntax:

📄 2. More testing constructions

1. The compound command [[expression]]

📄 3. Reading keyboard input

1. The script test-integer2.sh, that we have seen previously, has

📄 4. Examples

1. Let's see an example program that validates its input:

Intro

In this lesson we will see:

- how to branch with `if`
- reading keyboard input

> **NOTE**
>
> Before starting, let's get first some examples:

```
mkdir -p examples  &&  cd examples/

wget https://linux-cli.fs.al/examples/lesson11.tgz

tar xfz lesson11.tgz

cd lesson11/  &&  ls
```

> **DOWNLOAD LESSON11/INTRO.CAST**

1. Branching with `if`

The if statement has the following syntax:

```
if commands; then
    #commands
    #...
elif commands; then
    #commands
    #...
else
    #commands
    #...
fi
```

The `elif` and `else` parts are optional. The `elif` part can be repeated more than once.

1. Commands (including the scripts and shell functions) return an *exit status*. By convention, an exit status of zero indicates success and any other value indicates failure.

```
ls -d /usr/bin
```

```
echo $?
```

```
ls -d /bin/usr
```

```
echo $?
```

The builtin commands `true` and `false` do nothing except returning an exit status:

```
true
```

```
echo $?
```

```
false
```

```
echo $?
```

2. The `if` statement evaluates the success or failure of the commands, based on their exit status:

```
if true; then echo "It's true."; fi
```

```
if false; then echo "It's true."; fi
```

If a list of commands follows `if`, the last command in the list is evaluated:

```
if false; true; then echo "It's true."; fi
```

```
if true; false; then echo "It's true."; fi
```

3. The command used most frequently with `if` is `test`, which performs a variety of checks and comparisons.

```
touch foo.txt
```

```
if test -e foo.txt; then echo "File exists"; fi
```

```
if [ -e foo.txt ]; then echo "File exists"; \
    else echo "File does not exist"; fi
```

The command `[` is equivalent to `test` (it requires `]` as the last argument).

```
rm -f foo.txt
```

```
if [ -e foo.txt ]; then echo "File exists"; \
    else echo "File does not exist"; fi
```

4. Let's see an example script that is testing files:

▼ `test-file.sh`

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
        echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is writable."
```

```
    fi
    if [ -x "$FILE" ]; then
        echo "$FILE is executable/searchable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi

exit
```

```
vim test-file.sh
```

Notice that the parameter `$FILE` is quoted within the expression. This is not required, but it is a defense against the parameter being empty or containing whitespace.

Notice also the `exit` command at the end. It can optionally take a number as an argument, which becomes the exit status of the script, indicating success or failure. Without an argument, the default is the exit status of the last command executed. If the command `exit` is not present at all, the exit status of the script will be the exit status of the last command executed.

```
./test-file.sh
```

Edit the script, change the variable `FILE` and execute it again. You can also try to set it to the name of a directory.

▾ Let's do some more testing

```
sed -i test-file.sh \
    -e '/^FILE=/c FILE=./test-file.sh'
```

```
head test-file.sh
./test-file.sh
```

```
sed -i test-file.sh \
    -e '/^FILE=/c FILE=~/examples/'
head test-file.sh
./test-file.sh
```

```
sed -i test-file.sh \
    -e '/^FILE=/c FILE="non existent file"'
head test-file.sh
./test-file.sh
```

5. Let's see a similar example that uses a function instead:

test-file-fun.sh

```
#!/bin/bash

FILE=~/.bashrc

# test-file: Evaluate the status of a file
test_file () {
    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
            echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
            echo "$FILE is writable."
```

```
        fi
        if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi
}

test_file
```

```
vim test-file-fun.sh
```

Notice that instead of `exit`, a function can use `return` to indicate the exit status of the function.

```
./test-file-fun.sh
```

▼ Let's do some more testing

```
sed -i test-file-fun.sh \
    -e '/^FILE=/c FILE=./test-file.sh'
grep '^FILE=' test-file-fun.sh
./test-file-fun.sh
```

```
sed -i test-file-fun.sh \
    -e '/^FILE=/c FILE=~/examples/'
grep '^FILE=' test-file-fun.sh
./test-file-fun.sh
```

```
sed -i test-file-fun.sh \
    -e '/^FILE=/c FILE="non existent file"'
grep '^FILE=' test-file-fun.sh
./test-file-fun.sh
```

6. An example with testing strings:

test-string.sh

```
#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then
    echo "There is no answer." >&2
    exit 1
fi

if [ "$ANSWER" = "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNOWN."
fi
```

```
vim test-string.sh
```

Notice that when there is an error (`ANSWER` is empty), we print the error message to `stderr` by redirecting the output of `echo` (`>&2`). We also return an exit code of 1 by `exit 1`.

```
./test-string.sh
```

<details>
<summary>Let's do some more testing</summary>

```
sed -i test-string.sh -e '/^ANSWER=/c ANSWER=yes'
./test-string.sh

sed -i test-string.sh -e '/^ANSWER=/c ANSWER=no'
./test-string.sh

sed -i test-string.sh -e '/^ANSWER=/c ANSWER=xyz'
./test-string.sh

sed -i test-string.sh -e '/^ANSWER=/c ANSWER='
./test-string.sh
```

</details>

7. A similar example with testing integers:

<details>
<summary><code>test-integer.sh</code></summary>

```
#!/bin/bash

# test-integer: evaluate the value of an integer.

INT=-5

if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
```

</details>

```
fi

if [ "$INT" -eq 0 ]; then
    echo "INT is zero."
else
    if [ "$INT" -lt 0 ]; then
        echo "INT is negative."
    else
        echo "INT is positive."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
        echo "INT is even."
    else
        echo "INT is odd."
    fi
fi
```

```
vim test-integer.sh
```

```
./test-integer.sh
```

Change the number that is assigned to `INT` and execute it again.

<details>
<summary>More testing</summary>

```
sed -i test-integer.sh -e '/^INT=/c INT=11'
./test-integer.sh

sed -i test-integer.sh -e '/^INT=/c INT='
./test-integer.sh

sed -i test-integer.sh -e '/^INT=/c INT=0'
./test-integer.sh
```
</details>

```
sed -i test-integer.sh -e '/^INT=/c INT=12'
./test-integer.sh
```

8. For more details about the available tests let's see the help:

```
help test | less
```

```
help [
```

```
type test
```

```
type [
```

DOWNLOAD LESSON11/PART1.CAST

2. More testing constructions

1. The compound command `[[ expression ]]`

   Modern versions of bash include a compound command that acts as an enhanced replacement for test: `[[ expression ]]` . It has also an operator for regular expression matching: `=~`.

   ▼ `test-integer2.sh`

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ "$INT" -eq 0 ]; then
        echo "INT is zero."
    else
        if [ "$INT" -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

```
vim test-integer2.sh
```

```
./test-integer2.sh
```

▼ More testing

```
sed -i test-integer2.sh -e '/^INT=/c INT=11'
./test-integer2.sh

sed -i test-integer2.sh -e '/^INT=/c INT='
./test-integer2.sh

sed -i test-integer2.sh -e '/^INT=/c INT=0'
./test-integer2.sh

sed -i test-integer2.sh -e '/^INT=/c INT=12'
./test-integer2.sh
```

Another added feature of `[[ ]]` is that the `==` operator supports pattern matching the same way pathname expansion does:

```
FILE=foo.bar
```

```
if [[ $FILE == foo.* ]]; \
    then echo "$FILE matches pattern 'foo.*'"; fi
```

2. The compound command `(( integer expression ))`

   In addition to the compound command `[[ ]]`, bash also provides the compound command `(( ))`, which is useful for operating on integers.

```
if ((1)); then echo "It is true."; fi
```

```
if ((0)); then echo "It is true."; fi
```

```
if ((2)); then echo "It is true."; fi
```

With this test command we can simplify a bit the previous example script:

test-integer2a.sh

```
#!/bin/bash

# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if ((INT == 0)); then
        echo "INT is zero."
    else
        if ((INT < 0)); then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if (( ((INT % 2)) == 0)); then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

```
vim test-integer2a.sh
```

```
./test-integer2a.sh
```

```
diff -u test-integer2.sh test-integer2a.sh \
```

```
    | highlight -S bash -O xterm256 2>/dev/null
```

Notice that we don't use a `$` sign to refer to variables inside `(( ))`. Also, instead of `-eq` we use the operator `==`, instead of `-lt` we use `<`, etc. This makes the syntax a bit more natural.

3. We can use logical operators to create complex expressions. For the `test` (and `[ ]`) command the logical operators are `-a` (AND), `-o` (OR) and `!` (NOT). For the commands `[[ ]]` and `(( ))` they are: `&&`, `||` and `!`.

`test-integer3.sh`

```
#!/bin/bash

# test-integer3: determine if an integer is within a
# specified range of values.

MIN_VAL=1
MAX_VAL=100
INT=50

if [[ ! "$INT" =~ ^-?[0-9]+$ ]]; then
    echo "INT is not an integer." >&2
    exit 1
fi

if [[ "$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL" ]]; then
    echo "$INT is within $MIN_VAL to $MAX_VAL."
else
    echo "$INT is out of range."
fi

echo -n "Using [[ ... ]]: "
if [[ ! ("$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL") ]];
then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
```

```
fi

echo -n "Using (( ... )): "
if (( ! (INT > MIN_VAL && INT < MAX_VAL) )); then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi

echo -n "Using  [ ... ] : "
if [ ! \( "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" \) ];
then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi
```

```
vim test-integer3.sh
```

```
./test-integer3.sh
```

The option `-n` of the command `echo` tells it to not print a newline after the string.

Notice that because `test` and `[` are treated as commands (unlike `[[` and `((` which are special shell constructs), each argument given to them has to be separated by a space. Also, the parentheses that group logical expressions have to be escaped like this: `\(` and `\)`, otherwise shell will interpret them as something else (they have a special meaning in shell).

Usually it is more convenient to use `[[` instead of `test` or `[`.

4. We can use the operators `&&` (AND) and `||` (OR) for conditional execution of a command. They can be used like this:

`command1 && command2`

First is executed `command1`. If (and only if) it is successful, the `command2` is executed as well.

`command1 || command2`

First is executed `command1`. If (and only if) it fails, the `command2` is executed as well.

For example:

```
mkdir temp && cd temp
```

```
[[ -d temp ]] || mkdir temp
```

The first one is equivalent to:

```
if mkdir temp; then cd temp; fi
```

The second one is equivalent to:

```
if [[ -d temp ]]; then : ; else mkdir temp; fi
```

The command `:` is a null command, which means "do nothing". Without it we would get a syntax error.

> **DOWNLOAD LESSON11/PART2.CAST**

3. Reading keyboard input

1. The script `test-integer2.sh`, that we have seen previously, has the value of `INT` hardcoded, so that we need to edit the script in order to test another value. We can make it more interactive by using the command `read`:

   ▼ `read-integer.sh`

```
#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "
read int

if [[ ! "$int" =~ ^-?[0-9]+$ ]]; then
    echo "Input value is not an integer." >&2
    exit 1
fi

if ((int == 0)); then
    echo "$int is zero."
else
    if ((int < 0)); then
        echo "$int is negative."
    else
        echo "$int is positive."
    fi

    if (( ((int % 2)) == 0)); then
        echo "$int is even."
    else
        echo "$int is odd."
    fi
fi
```

```
vim read-integer.sh
```

```
./read-integer.sh    # enter 0
```

```
./read-integer.sh    # enter 7
```

```
./read-integer.sh    # enter 4
```

```
./read-integer.sh    # enter -3
```

```
./read-integer.sh    # enter -8
```

The command read assigns the input to the variable `int`. If no variable name is given, then it assigns the input to the variable `REPLY`.

2. The command `read` can also get multiple variable names, as in this example:

`read-multiple.sh`

```
#!/bin/bash

# read-multiple: read multiple values from keyboard

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

```
vim read-multiple.sh
```

In this script, we assign and display up to five values.

```
```

```
./read-multiple.sh    # enter: a b c d e
```

```
./read-multiple.sh    # enter: a b
```

```
./read-multiple.sh    # enter: a b c d e f g
```

3. It can also get some options:

```
help read | less
```

With the `-p` option we can provide a prompt string:

```
read -p "Enter one or more values > "    # enter: a b c
```

```
echo "REPLY = '$REPLY'"
```

The option `-s` can be used for a silent input, and `-t` to set a timeout. Let's see them in an example that tries to read a password:

`read-user-pass.sh`

```
#!/bin/bash

# -e: use readline to get the input
# -p: display a prompt
# -i: provide a default reply
read -e -p "What is your user name > " -i $USER username
echo "Welcome '$username'"

# -t: timeout (in seconds)
# -s: silent (do not echo characters to the display as they are
```

```
typed)
# -p: display a prompt
if read -t 10 -sp "Enter your secret passphrase > " secret_pass
then
    echo -e "\nYour secret passphrase is '$secret_pass'"
else
    echo -e "\nInput timed out" >&2
    exit 1
fi
```

```
vim read-user-pass.sh
```

```
./read-user-pass.sh
```

If we don't type a password in 10 seconds, the `read` command will time out with an error exit code.

4. The input provided to `read` is split by the shell. There is a shell variable named `IFS` (Internal Field Separator) which contains a list of separators. By default it contains a space, a tab, and a newline character. Each of them can separate items from each-other.

   If we want to modify the way that the input is separated into fields, we can change the value of `IFS`.

   `read-ifs.sh`

   ```
   #!/bin/bash
   # read-ifs: read fields from a file

   read -p "Enter a username > " user_name

   file_info="$(grep "^$user_name:" /etc/passwd)"
   ```

```
if [ -z "$file_info" ]; then
    echo "No such user '$user_name'" >&2
    exit 1
fi

IFS=":" read user pw uid gid name home shell <<< "$file_info"

echo "User      = '$user'"
echo "UID       = '$uid'"
echo "GID       = '$gid'"
echo "Full Name = '$name'"
echo "Home Dir  = '$home'"
echo "Shell     = '$shell'"
```

```
vim read-ifs.sh
```

Notice that we set `IFS=":"` before calling `read`. The shell allows one or more variable assignments to take place immediately before a command. These assignments alter the environment for the command that follows. The effect of the assignment is temporary changing the environment, for the duration of the command.

It is the same as doing this, but more concise:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

The `<<<` operator indicates a *here string*. A here string is like a here document, only shorter, consisting of a single string. We need to use it because `read` does not work well with a pipe (for example: `echo "$file_info" | read ...`)

```
./read-ifs.sh    # enter: xyz
```

```
./read-ifs.sh    # enter: user1
```

DOWNLOAD LESSON11/PART3.CAST

4. Examples

1. Let's see an example program that validates its input:

validate.sh

```
#!/bin/bash

# read-validate: validate input

invalid_input () {
    echo "Invalid input '$REPLY'" >&2
    exit 1
}

read -p "Enter a single item > "

# input is empty (invalid)
[[ -z "$REPLY" ]] && invalid_input

# input is multiple items (invalid)
(( "$(echo "$REPLY" | wc -w)" > 1 )) && invalid_input

# is input a valid filename?
if [[ "$REPLY" =~ ^[-[:alnum:]\._]+$ ]]; then
    echo "'$REPLY' is a valid filename."
    if [[ -e "$REPLY" ]]; then
        echo "And file '$REPLY' exists."
    else
        echo "However, file '$REPLY' does not exist."
    fi

    # is input a floating point number?
    if [[ "$REPLY" =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
        echo "'$REPLY' is a floating point number."
    else
```

```
        echo "'$REPLY' is not a floating point number."
    fi

    # is input an integer?
    if [[ "$REPLY" =~ ^-?[[:digit:]]+$ ]]; then
        echo "'$REPLY' is an integer."
    else
        echo "'$REPLY' is not an integer."
    fi
else
    echo "The string '$REPLY' is not a valid filename."
fi
```

```
vim validate.sh
```

Try it a few times with different inputs:

```
./validate.sh
```

2. Let's see a menu driven example program:

menu.sh

```
#!/bin/bash

# read-menu: a menu driven system information program

clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
```

```
read -p "Enter selection [0-3] > "

if [[ ! "$REPLY" =~ ^[0-3]$ ]]; then
    echo "Invalid entry." >&2
    exit 1
fi

if [[ "$REPLY" == 0 ]]; then
    echo "Program terminated."
    exit
fi

if [[ "$REPLY" == 1 ]]; then
    echo "Hostname: $HOSTNAME"
    uptime
    exit
fi

if [[ "$REPLY" == 2 ]]; then
    df -h .
    exit
fi

if [[ "$REPLY" == 3 ]]; then
    if [[ "$(id -u)" -eq 0 ]]; then
        echo "Home Space Utilization (All Users)"
        du -sh /home/*
    else
        echo "Home Space Utilization ($USER)"
        du -sh "$HOME"
    fi
    exit
fi
```

```
vim menu.sh
```

Notice the use of the `exit` command in this script. It is used here to prevent the script from executing unnecessary code after an action has been carried out.

Try if a few times.

3. As an exercise, try to modify these examples so that instead of `[[... ]]` and `((...)`, they use the command `test`.

   Hint: Use grep to evaluate the regular expressions and evaluate the exit status.

DOWNLOAD LESSON11/PART4.CAST

📄 Intro

In this lesson we will see:

📄 1. Looping with while and until

The syntax of the while command is as follows:

📄 2. Branching with case

The command case is a multiple-choice command.

📄 3. Positional parameters

1. The shell provides a set of variables called _positional

📄 4. An example

Let's try to improve the program sys_info.sh, that we started to

📄 5. Testing the example

1. Let's see the usage of the program:

Intro

In this lesson we will see:

- looping with `while`/`until`
- branching with `case`
- positional parameters

> **NOTE**
>
> Let's get first some examples:
>
> ```
> mkdir -p examples  &&  cd examples/
>
> wget https://linux-cli.fs.al/examples/lesson12.tgz
>
> tar xfz lesson12.tgz
>
> cd lesson12/  &&  ls
> ```

> **DOWNLOAD LESSON12/INTRO.CAST**

1. Looping with while and until

The syntax of the `while` command is as follows:

```
while commands; do commands; done
```

1. A simple example:

   while-count.sh

   ```
   #!/bin/bash

   # while-count: display a series of numbers

   count=1

   while [[ "$count" -le 5 ]]; do
       echo "$count"
       count=$((count + 1))
   done
   echo "Finished."
   ```

   ```
   vim while-count.sh
   ```

   ```
   ./while-count.sh
   ```

2. We can use a `while` loop to improve the menu program from the previous lesson:

while-menu.sh

```
#!/bin/bash

# while-menu: a menu driven system information program

DELAY=3 # Number of seconds to display results

while [[ "$REPLY" != 0 ]]; do
    clear
    cat <<- _EOF_
        Please Select:
        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit
        _EOF_
    read -p "Enter selection [0-3] > "

    if [[ "$REPLY" =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep "$DELAY"
        fi
        if [[ "$REPLY" == 2 ]]; then
            df -h .
            sleep "$DELAY"
        fi
        if [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh "$HOME"
            fi
            sleep "$DELAY"
        fi
```

```
    else
        echo "Invalid entry."
        sleep "$DELAY"
    fi
done
echo "Program terminated."
```

```
vim while-menu.sh
```

By enclosing the menu in a while loop, we are able to have the program repeat the menu display after each selection. The loop continues as long as `REPLY` is not equal to `0` and the menu is displayed again, giving the user the opportunity to make another selection. At the end of each action, a `sleep` command is executed so the program will pause for a few seconds to allow the results of the selection to be seen before the screen is cleared and the menu is redisplayed. Once `REPLY` is equal to `0`, indicating the "quit" selection, the loop terminates and execution continues with the line following `done`.

```
./while-menu.sh
```

3. Inside a loop in bash we can use `break` and `continue`.

▼ `while-menu2.sh`

```
#!/bin/bash

# while-menu: a menu driven system information program

DELAY=3 # Number of seconds to display results

while true; do
    clear
    cat <<- _EOF_
        Please Select:
```

```
        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit
        _EOF_
    read -p "Enter selection [0-3] > "

    if [[ "$REPLY" == 0 ]]; then
        break
    fi
    if [[ ! "$REPLY" =~ ^[0-3]$ ]]; then
        echo "Invalid entry."
        sleep "$DELAY"
        continue
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        sleep "$DELAY"
        continue
    fi
    if [[ "$REPLY" == 2 ]]; then
        df -h .
        sleep "$DELAY"
        continue
    fi
    if [[ "$REPLY" == 3 ]]; then
        if [[ "$(id -u)" -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh "$HOME"
        fi
        sleep "$DELAY"
        continue
    fi
done
echo "Program terminated."
```

```
vim while-menu2.sh
```

```
./while-menu2.sh
```

4. The `until` loop is very similar to the `while` loop, but with a negated condition.

<details>
<summary>until-count.sh</summary>

```
#!/bin/bash

# until-count: display a series of numbers

count=1

until [[ "$count" -gt 5 ]]; do
    echo "$count"
    count=$((count + 1))
done
echo "Finished."
```

</details>

```
vim until-count.sh
```

```
./until-count.sh
```

5. We can also read the standard input with `while` and `until`:

```
cat distros.txt
```

<details>
<summary>while-read.sh</summary>
</details>

```
#!/bin/bash

# while-read: read lines from a file

while read distro version release
do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
        "$distro" \
        "$version" \
        "$release"
done < distros.txt
```

```
vim while-read.sh
```

The `while` loop will continue as long as the `read` command is successful getting input from stdin, and we redirect stdin to get data from the file `distros.txt` (by using the operator `<` at the end of the `while` command).

```
./while-read.sh
```

We can also use a pipe (`|`) to redirect the stdin:

`while-read2.sh`

```
#!/bin/bash

# while-read2: read lines from a file

sort -k 1,1 -k 2n distros.txt | \
    while read distro version release; do
        printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
            "$distro" \
```

```
            "$version" \
            "$release"
    done
```

```
vim while-read2.sh
```

Notice that we are breaking long commands by adding a `\` at the end of a line, in order to make the program more readable and clear.

```
./while-read2.sh
```

DOWNLOAD LESSON12/PART1.CAST

2. Branching with case

The command `case` is a multiple-choice command.

1. Let's see an example that implements a menu program with `case`:

`case-menu.sh`

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

case "$REPLY" in
    0)  echo "Program terminated."
        exit
        ;;
    1)  echo "Hostname: $HOSTNAME"
        uptime
        ;;
    2)  df -h .
        ;;
    3)  if [[ "$(id -u)" -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
```

```
            echo "Home Space Utilization ($USER)"
            du -sh "$HOME"
        fi
        ;;
    *)  echo "Invalid entry" >&2
        exit 1
        ;;
esac
```

```
vim case-menu.sh
```

We have seen this example before, implemented with `if` and it is clear that with `case` it is much simpler.

```
./case-menu.sh
```

`case` attempts a match against the specified patterns. When a match is found, the commands associated with the specified pattern are executed. After a match is found, no further matches are attempted.

2. The patterns used by case are the same as those used by pathname expansion. For example:

    - `a)` -- matches the character "a"
    - `[[:alpha:]])` -- matches any alphabetic character
    - `???)` -- matches 3 characters
    - `*.txt)` -- matches anything that ends in `.txt`
    - `*)` -- matches anything

    It is good practice to include `*)` as the last pattern in a `case` command, to catch any values that did not match a previous pattern.

    Let's see an example script with patterns:

---

case-patterns.sh

```
#!/bin/bash

read -p "enter word > "

case "$REPLY" in
    [[:alpha:]])  echo "it is a single alphabetic character" ;;
    [ABC][0-9])   echo "it is A, B, or C followed by a digit" ;;
    ???)          echo "it is three characters long" ;;
    *.txt)        echo "it is a word ending in '.txt'" ;;
    *)            echo "it is something else" ;;
esac
```

```
vim case-patterns.sh
```

```
./case-patterns.sh    # enter: x
```

```
./case-patterns.sh    # enter: B2
```

```
./case-patterns.sh    # enter: foo.txt
```

```
./case-patterns.sh    # enter: xyz
```

```
./case-patterns.sh    # enter: ab
```

3. It is also possible to combine multiple patterns using the vertical bar character as a separator. Let's see a modified menu program that uses letters instead of digits for menu selection:

`case-menu-l`.sh

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "

case "$REPLY" in
    q|Q)  echo "Program terminated."
          exit
          ;;
    a|A)  echo "Hostname: $HOSTNAME"
          uptime
          ;;
    b|B)  df -h .
          ;;
    c|C)  if [[ "$(id -u)" -eq 0 ]]; then
              echo "Home Space Utilization (All Users)"
              du -sh /home/*
          else
              echo "Home Space Utilization ($USER)"
              du -sh "$HOME"
          fi
          ;;
    *)    echo "Invalid entry" >&2
          exit 1
          ;;
esac
```

```
vim case-menu-l.sh
```

Notice how the new patterns allow for entry of both uppercase and lowercase letters.

```
./case-menu-l.sh
```

4. When a pattern is matched, the corresponding actions are executed, and `;;` makes sure that processing is stopped (without trying to match the following patterns). If we want instead to try matching them as well, we can use `;;&` instead, as in this example:

▼ case4.sh

```
#!/bin/bash
# case4: test a character

# -n 1:  read only one char and don't wait for enter to be
pressed
read -n 1 -p "Type a character > "
echo
case "$REPLY" in
    [[:upper:]])    echo "'$REPLY' is upper case."
;;&
    [[:lower:]])    echo "'$REPLY' is lower case."
;;&
    [[:alpha:]])    echo "'$REPLY' is alphabetic."
;;&
    [[:digit:]])    echo "'$REPLY' is a digit."
;;&
    [[:graph:]])    echo "'$REPLY' is a visible character."
;;&
    [[:punct:]])    echo "'$REPLY' is a punctuation symbol."
;;&
    [[:space:]])    echo "'$REPLY' is a whitespace character."
;;&
```

```
    [[:xdigit:]])   echo "'$REPLY' is a hexadecimal digit."
;;&
esac
```

```
vim case4.sh
```

```
./case4.sh    # enter: a
```

```
./case4.sh    # enter: X
```

```
./case4.sh    # enter: +
```

DOWNLOAD LESSON12/PART2.CAST

3. Positional parameters

1. The shell provides a set of variables called *positional parameters* that contain the individual words on the command line.

   Let's test them with a simple script:

   posit-param.sh

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
Number of arguments: $#
"
```

```
vim posit-param.sh
```

```
./posit-param.sh
```

```
$(pwd)/posit-param.sh
```

Notice that `$0` contains th first word of the command, which is the name and path of the command itself.

```
./posit-param.sh a b c d
```

The special variable `$#` contains the number of arguments.

If we need to use more than 9 arguments, then we can use `${10}`, `${11}` etc. to access them (with curly braces).

2. The `shift` command causes all the parameters to "move down one" each time it is executed.

<details>
<summary>posit-param2.sh</summary>

```
#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done
```

</details>

```
vim posit-param2.sh
```

In this example there is a loop that evaluates the number of arguments remaining and continues as long as there is at least one.

```
./posit-param2.sh a b c d
```

```
./posit-param2.sh a b c d e f g
```

3. Here is another example:

<details>
<summary>file-info.sh</summary>

```
#!/bin/bash

# file-info: simple file information program

PROGNAME="$(basename "$0")"

if [[ -e "$1" ]]; then
    echo -e "\nFile Type:"
    file "$1"
    echo -e "\nFile Status:"
    stat "$1"
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi
```

</details>

```
vim file-info.sh
```

This program displays the file type (determined by the `file` command) and the file status (from the `stat` command) of a specified file.

It checks the first argument, and if it does not exist, exits with an error message that shows how to use this script.

The command `basename` gets only the name of the file (discarding the path).

```
./file-info.sh
```

```
./file-info.sh posit-param2.sh
```

```
./file-info.sh .
```

```
./file-info.sh xyz
```

4. Positional parameters can be used with functions as well.

▼ `file-info-fun.sh`

```
#!/bin/bash

file_info () {
    if [[ -e "$1" ]]; then
        echo -e "\nFile Type:"
        file "$1"
        echo -e "\nFile Status:"
        stat "$1"
    else
        local PROGNAME="$(basename "$0")"
        echo "$PROGNAME: usage: $FUNCNAME file" >&2
        return 1
    fi
}

echo -e '\nCalling file_info without args: file_info'
file_info
```

```
FILE=$0
echo -e '\nCalling file_info with an argument: file_info $FILE'
file_info $FILE
```

```
vim file-info-fun.sh
```

Notice that `$0` always contains the full pathname of the first item on the command line (i.e., the name of the program), even inside a function.

Notice also that `FUNCNAME` is a variable that always contains the name of the current function.

```
./file-info-fun.sh
```

5. The shell provides two special variables that contain the list of all the positional parameters. They are `$*` and `$@`. Let's try an example that shows their differences:

posit-param3.sh

```
#!/bin/bash

# posit-params3: script to demonstrate $* and $@

print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
    echo
}

pass_params () {
```

```
    echo '-- $* --' ;    print_params $*
    echo '-- "$*" --' ;  print_params "$*"
    echo '-- $@ --' ;    print_params $@
    echo '-- "$@" --' ;  print_params "$@"
}

pass_params "word" "words with spaces"
```

```
vim posit-param3.sh
```

```
./posit-param3.sh
```

You see that both `$*` and `$@` give 4 parameters. `"$*"` gives a single parameter, and `"$@"` gives back the two original parameters. This happens because `$*` is a string list of all the parameters, while `$@` is an array of all the parameters.

Anyway, the most useful construct seems to be `"$@"` because it preserves the original list of the parameters, and this is what we want in most of the cases.

DOWNLOAD LESSON12/PART3.CAST

4. An example

Let's try to improve the program `sys_info.sh`, that we started to build in a previous lesson, by adding some parameters and option to it. We want to be able to:

- Tell it to save the output to a specific file (instead of sending it to stdout), by using the options `-f file` or `--file file`.
- Tell it to ask interactively for a filename for saving the output. This option should be specified by `-i` or `--interactive`.
- Use the options `-h` or `--help` to make the program output information about its usage.

1. There is a small git repo on the archive `sys_info.tgz`, let's open it:

```
tar xfz sys_info.tgz
```

```
cd sys_info/
```

```
ls -al
```

```
git log --oneline
```

```
git tag
```

2. Let's get first the initial version of the script (that was developed in a previous lesson):

```
git checkout -q 1.initial
```

```
git status
```

```
vim sys_info.sh
```

sys_info.initial.sh

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated on $CURRENT_TIME, by $USER"

report_uptime() {
   cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
        _EOF_
    return
}

report_disk_space() {
   cat <<- _EOF_
        <h2>Disk Space Utilization</h2>
        <pre>$(df -h .)</pre>
        _EOF_
    return
}

report_home_space() {
    if [[ "$(id -u)" -eq 0 ]]; then
        cat <<- _EOF_
                <h2>Home Space Utilization (All Users)</h2>
                <pre>$(du -hs /home/*)</pre>
                _EOF_
    else
```

```
        cat <<- _EOF_
                <h2>Home Space Utilization ($PWD)</h2>
                <pre>$(du -hs "$PWD")</pre>
                _EOF_
    fi
    return
}


cat << _EOF_
<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </body>
</html>
_EOF_
```

```
./sys_info.sh
```

3. Let's see some modifications and improvements to it:

   Enclose in a function the last part (that generates the HTML page):

```
git checkout -q 2.write_html_page
```

```
git diff 1.initial
```

▼ git diff 1.initial

```
diff --git a/sys_info.sh b/sys_info.sh
index 4f261db..6291299 100755
--- a/sys_info.sh
+++ b/sys_info.sh
@@ -37,18 +37,23 @@ report_home_space() {
     return
 }

+write_html_page () {
+    cat <<- _EOF_
+       <html>
+           <head>
+               <title>$TITLE</title>
+           </head>
+           <body>
+               <h1>$TITLE</h1>
+               <p>$TIMESTAMP</p>
+               $(report_uptime)
+               $(report_disk_space)
+               $(report_home_space)
+           </body>
+       </html>
+       _EOF_
+}
+
+# output html page
+write_html_page

-cat << _EOF_
-<html>
-    <head>
-        <title>$TITLE</title>
-    </head>
-    <body>
-        <h1>$TITLE</h1>
-        <p>$TIMESTAMP</p>
-        $(report_uptime)
-        $(report_disk_space)
-        $(report_home_space)
-    </body>
```

```
-</html>
-_EOF_
```

Add a function that displays the usage of the program:

```
git checkout -q 3.usage
```

```
git diff 2.write_html_page
```

▼ git diff 2.write_html_page

```
diff --git a/sys_info.sh b/sys_info.sh
index 6291299..b58cf06 100755
--- a/sys_info.sh
+++ b/sys_info.sh
@@ -1,7 +1,23 @@
 #!/bin/bash
-
 # Program to output a system information page

+usage () {
+    cat <<- _EOF_
+       $PROGNAME: usage:
+
+           $PROGNAME (-f | --file) <file>
+               Output the report to the given file.
+
+           $PROGNAME (-i | --interactive)
+               Get the output file interactively from the
keyboard.
+
+           $PROGNAME [-h | --help]
+               Display this help message.
+       _EOF_
+    return
```

```
+}
+
+PROGNAME="$(basename "$0")"
 TITLE="System Information Report For $HOSTNAME"
 CURRENT_TIME=$(date +"%x %r %Z")
 TIMESTAMP="Generated on $CURRENT_TIME, by $USER"
```

4. Add some code that reads the command line options:

```
git checkout -q 4.process_options
```

```
git diff 3.usage
```

▼ git diff 3.usage

```
diff --git a/sys_info.sh b/sys_info.sh
index b58cf06..37ddc1b 100755
--- a/sys_info.sh
+++ b/sys_info.sh
@@ -17,6 +17,30 @@ usage () {
     return
 }

+# process command line options
+interactive=''
+filename=''
+while [[ -n "$1" ]]; do
+    case "$1" in
+        -f | --file)
+            shift
+            filename="$1"
+            ;;
+        -i | --interactive)
+            interactive=1
+            ;;
```

```
+        -h | --help)
+            usage
+            exit
+            ;;
+        *)
+            usage >&2
+            exit 1
+            ;;
+    esac
+    shift
+done
+
 PROGNAME="$(basename "$0")"
 TITLE="System Information Report For $HOSTNAME"
 CURRENT_TIME=$(date +"%x %r %Z")
```

We use a `while` loop and `shift` to process all the options. Inside the loop we use `case` to match the option with one of those that are expected. If the option is `-f` (or `--file`), we interpret the next parameter as a filename and set it to the variable `filename`. If the option is `-i` (or `--interactive`), we set the variable `interactive` to `1` (otherwise it will remain empty).

Notice that the actions corresponding to `-h | --help)` and `*)` are very similar, they display the usage and exit the program. However the later case is considered an error, because there is an unknown/unsupported option, so the usage is sent to *stderr* (`>&2`) and the program exits with code `1` (error).

5. If the option `-i` or (`--interactive`) is supplied, the program should get a file name interactively (from the keyboard). Let's see the code that does that:

```
git checkout -q 5.interactive
```

```
git diff 4.process_options
```

▾ git diff 4.process_options

```
diff --git a/sys_info.sh b/sys_info.sh
index 37ddc1b..f8a2fba 100755
--- a/sys_info.sh
+++ b/sys_info.sh
@@ -41,6 +41,32 @@ while [[ -n "$1" ]]; do
     shift
 done

+# interactive mode
+if [[ -n "$interactive" ]]; then
+    while true; do
+        read -p "Enter name of output file: " filename
+        if [[ -e "$filename" ]]; then
+            read -p "'$filename' exists. Overwrite? [y/n/q] > "
+            case "$REPLY" in
+                Y|y)
+                   break
+                    ;;
+                Q|q)
+                   echo "Program terminated."
+                    exit
+                    ;;
+                *)
+                    continue
+                    ;;
+            esac
+        elif [[ -z "$filename" ]]; then
+            continue
+        else
+            break
+        fi
+    done
+fi
+
 PROGNAME="$(basename "$0")"
 TITLE="System Information Report For $HOSTNAME"
 CURRENT_TIME=$(date +"%x %r %Z")
```

This code is executed only if the global variable `interactive` is not empty. There is an infinite `while` loop that tries to read the name of the file into to global variable `filename`. We check that the given value is not empty and that such a file does not exist already. If there is already such a file, we ask again whether we can overwrite the file or not.

We use the loop so that we can ask again for another file name if the given one is not suitable, and we repeat until we have a suitable file name (stored in the variable `filename`).

6. Now let's see the code that outputs the HTML page:

```
git checkout -q 6.output_html_page
```

```
git diff 5.interactive
```

<details>
<summary>git diff 5.interactive</summary>

```
diff --git a/sys_info.sh b/sys_info.sh
index f8a2fba..8c3e576 100755
--- a/sys_info.sh
+++ b/sys_info.sh
@@ -121,5 +121,13 @@ write_html_page () {
 }

 # output html page
-write_html_page
-
+if [[ -n "$filename" ]]; then
+    if touch "$filename" && [[ -f "$filename" ]]; then
+        write_html_page > "$filename"
+    else
+        echo "$PROGNAME: Cannot write file '$filename'" >&2
+        exit 1
+    fi
```
</details>

```
+else
+    write_html_page
+fi
```

If the variable `filename` is empty, then the HTML page will be sent to the *stdout*, same as before. Otherwise the program will try to send it to the given file (using redirection). The program also makes sure that we can write to the file, by trying to create an empty file first.

7. Finally, let's study the latest version of the program.

```
git checkout master
```

```
vim sys_info.sh
```

`sys_info.final.sh`

```
#!/bin/bash
# Program to output a system information page

PROGNAME="$(basename "$0")"

usage () {
    cat <<- _EOF_
        Usage:
            $PROGNAME
                Output the report to the stdout.

            $PROGNAME (-f | --file) <file>
                Output the report to the given file.

            $PROGNAME (-i | --interactive)
                Get the output file interactively from the
keyboard.
```

```
            $PROGNAME (-h | --help)
                Display this help message.
        _EOF_
    return
}

main () {
    # global aux vars
    interactive=''
    filename=''

    process_options "$@"
    interactive_mode
    output_html_page
}

process_options () {
    # process command line options
    while [[ -n "$1" ]]; do
        case "$1" in
            -f | --file)
                shift
                filename="$1"
                ;;
            -i | --interactive)
                interactive=1
                ;;
            -h | --help)
                usage
                exit
                ;;
            *)
                usage >&2
                exit 1
                ;;
        esac
        shift
    done
}

interactive_mode () {
    # interactive mode
```

```
    if [[ -n "$interactive" ]]; then
        while true; do
            read -p "Enter name of output file: " filename
            if [[ -e "$filename" ]]; then
                read -p "'$filename' exists. Overwrite? [y/n/q]
> "
                case "$REPLY" in
                    Y|y)
                        break
                        ;;
                    Q|q)
                        echo "Program terminated."
                        exit
                        ;;
                    *)
                        continue
                        ;;
                esac
            elif [[ -z "$filename" ]]; then
                continue
            else
                break
            fi
        done
    fi
}

output_html_page () {
    # output html page
    if [[ -n "$filename" ]]; then
        if touch "$filename" && [[ -f "$filename" ]]; then
            write_html_page > "$filename"
        else
            echo "$PROGNAME: Cannot write file '$filename'" >&2
            exit 1
        fi
    else
        write_html_page
    fi
}

write_html_page () {
```

```
    local TITLE="System Information Report For $HOSTNAME"
    local CURRENT_TIME=$(date +"%x %r %Z")
    local TIMESTAMP="Generated on $CURRENT_TIME, by $USER"

    cat <<- _EOF_
        <html>
            <head>
                <title>$TITLE</title>
            </head>
            <body>
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
            </body>
        </html>
        _EOF_
}

report_uptime() {
   cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
        _EOF_
    return
}

report_disk_space() {
   cat <<- _EOF_
        <h2>Disk Space Utilization</h2>
        <pre>$(df -h .)</pre>
        _EOF_
    return
}

report_home_space() {
    if [[ "$(id -u)" -eq 0 ]]; then
        cat <<- _EOF_
                <h2>Home Space Utilization (All Users)</h2>
                <pre>$(du -hs /home/*)</pre>
                _EOF_
```

```
    else
        cat <<- _EOF_
                <h2>Home Space Utilization ($PWD)</h2>
                <pre>$(du -hs "$PWD")</pre>
                _EOF_
    fi
    return
}

# call the main function
main "$@"
```

```
:set tabstop=8
```

Notice that we have placed almost all the code inside a function. There is a function `main()` that calls some other functions, then these functions call some other ones, and so on.

The `main()` function is called at the very end of the program, like this:

`main "$@"`

This makes sure that all the parameters given to the program are passed to the main function. The main function in turn passes all of them to the function `process_options`, like this:

`process_options "$@"`

DOWNLOAD LESSON12/PART4.CAST

5. Testing the example

1. Let's see the usage of the program:

```
./sys_info.sh -h
```

```
./sys_info.sh --help
```

2. We see that we can also call it without any parameters. Let's try it:

```
./sys_info.sh
```

```
./sys_info.sh > report1.html
```

```
lynx report1.html
```

Exit with `qy`

3. We can write the output to a file using the option `-f` or `--file`:

```
./sys_info.sh -f report2.html
```

```
cat report2.html
```

```
./sys_info.sh --file report3.html
```

4. Let's also try the interactive method:

```
./sys_info.sh -i
```

- Enter the name of the output file as `report1.html`
- Press `n` for not overwriting the existing file.
- Enter the new name of the output file as `report2.html`
- Press `y` for overwriting the existing file.

```
./sys_info.sh --interactive
```

- Enter the name of the output file as `report3.html`
- Press `n` for not overwriting the existing file.
- Enter the new name of the output file as `report4.html`

```
lynx report4.html
```

Exit with `qy`

> **DOWNLOAD LESSON12/PART5.CAST**

📄 Intro

In this lesson we will see:

📄 1. Looping with for

1. With for we can loop a list of words:

📄 2. Variable expansions

1. We have already seen that sometimes we need to surround variable

📄 3. String operations

1. Length of the string: $

📄 4. Arithmetic evaluation and expansion

We have seen before $((expression)) where expression is an

Intro

In this lesson we will see:

- looping with `for`
- parameter expansion
- arithmetic evaluation and expansion

> **NOTE**
>
> Let's get first some examples:

```
mkdir -p examples  &&  cd examples/

wget https://linux-cli.fs.al/examples/lesson13.tgz

tar xfz lesson13.tgz

cd lesson13/  &&  ls
```

> **DOWNLOAD LESSON13/INTRO.CAST**

1. Looping with `for`

1. With `for` we can loop a list of words:

```
for i in A B C D; do echo $i; done
```

```
for i in {A..D}; do echo $i; done
```

```
for i in *.sh; do echo "$i"; done
```

In these cases it is using shell expansions.

The last file expansion may fail if there are no files like that, in which case shell will return just `*.sh` (instead of a list of matching files). To protect against this, we can rewrite the last example like this:

```
for i in *.sh; do [[ -e "$i" ]] && echo "$i"; done
```

2. Let's see an example that finds the longest word in a file:

▼ `longest-word.sh`

```
#!/bin/bash

# longest-word: find longest string in a file

while [[ -n "$1" ]]; do
    if [[ -r "$1" ]]; then
        max_word=
        max_len=0
```

```
        for i in $(strings "$1"); do
            len="$(echo -n "$i" | wc -c)"
            if (( len > max_len )); then
                max_len="$len"
                max_word="$i"
            fi
        done
        echo "$1: '$max_word' ($max_len characters)"
    fi
    shift
done
```

```
vim longest-word.sh
```

Actually it can take one or more files as parameters and process each of them. This is implemented by the `while` loop:

```
while [[ -n "$1" ]]; do
    if [[ -r "$1" ]]; then
        # . . . . .
        # . . . . .
    fi
    shift
done
```

The `if` checks that the file is readable (otherwise it is skipped).

The list of words of the file is produced by the command `strings "$1"`. We use a *command substitution* to use this list in the `for` statement:

```
for i in $(strings "$1"); do
```

Notice that we are not surrounding `$(strings "$1")` with double quotes, otherwise it will be treated as a single string, which is not what we want.

Let's try it:

```
echo *
```

```
./longest-word.sh *
```

3. Let's see a slightly modified version of the previous example:

`longest-word2.sh`

```
#!/bin/bash

# longest-word2: find longest string in a file

for i; do
    if [[ -r "$i" ]]; then
        max_word=
        max_len=0
        for j in $(strings "$i"); do
            len="$(echo -n "$j" | wc -c)"
            if (( len > max_len )); then
                max_len="$len"
                max_word="$j"
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done
```

```
vim longest-word2.sh
```

The modification consists on replacing the `while` loop with a `for` loop like this:

```
for i; do
    if [[ -r "$i" ]]; then
        # . . . . .
        # . . . . .
    fi
done
```

And since we are using the variable `i` for the outer loop, in the inner loop we have replaced `i` with `j`.

When we omit the list of words, `for` will use by default the positional parameters.

```
echo *
```

```
./longest-word2.sh *
```

4. `for` has also another form, which is similar to that of C (and many other languages):

```
for ((i=0; i<5; i=i+1)); do echo $i; done
```

As you know, the construct `((...))` is used for arithmetic expressions, and inside it we don't use a `$` in front of the variables.

5. Let's also make a small modification to the program `sys_info.sh` that we saw in the last lesson:

▼ `sys_info.sh`

```
#!/bin/bash
# Program to output a system information page
```

```
PROGNAME="$(basename "$0")"

usage () {
    cat <<- _EOF_
        Usage:
            $PROGNAME
                Output the report to the stdout.

            $PROGNAME (-f | --file) <file>
                Output the report to the given file.

            $PROGNAME (-i | --interactive)
                Get the output file interactively from the
keyboard.

            $PROGNAME (-h | --help)
                Display this help message.
        _EOF_
    return
}

main () {
    # global aux vars
    interactive=''
    filename=''

    process_options "$@"
    interactive_mode
    output_html_page
}

process_options () {
    # process command line options
    while [[ -n "$1" ]]; do
        case "$1" in
            -f | --file)
                shift
                filename="$1"
                ;;
            -i | --interactive)
                interactive=1
                ;;
```

```
            -h | --help)
                usage
                exit
                ;;
            *)
                usage >&2
                exit 1
                ;;
        esac
        shift
    done
}

interactive_mode () {
    # interactive mode
    if [[ -n "$interactive" ]]; then
        while true; do
            read -p "Enter name of output file: " filename
            if [[ -e "$filename" ]]; then
                read -p "'$filename' exists. Overwrite? [y/n/q]
> "
                case "$REPLY" in
                    Y|y)
                        break
                        ;;
                    Q|q)
                        echo "Program terminated."
                        exit
                        ;;
                    *)
                        continue
                        ;;
                esac
            elif [[ -z "$filename" ]]; then
                continue
            else
                break
            fi
        done
    fi
}
```

```
output_html_page () {
    # output html page
    if [[ -n "$filename" ]]; then
        if touch "$filename" && [[ -f "$filename" ]]; then
            write_html_page > "$filename"
        else
            echo "$PROGNAME: Cannot write file '$filename'" >&2
            exit 1
        fi
    else
        write_html_page
    fi
}

write_html_page () {
    local TITLE="System Information Report For $HOSTNAME"
    local CURRENT_TIME=$(date +"%x %r %Z")
    local TIMESTAMP="Generated on $CURRENT_TIME, by $USER"

    cat <<- _EOF_
        <html>
            <head>
                <title>$TITLE</title>
            </head>
            <body>
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
            </body>
        </html>
        _EOF_
}

report_uptime() {
   cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
        _EOF_
    return
}
```

```
report_disk_space() {
   cat <<- _EOF_
        <h2>Disk Space Utilization</h2>
        <pre>$(df -h .)</pre>
        _EOF_
    return
}

report_home_space () {
    local format="%8s%10s%10s\n"
    local i dir_list total_files total_dirs total_size user_name
    if [[ "$(id -u)" -eq 0 ]]; then
        dir_list=/home/*
        user_name="All Users"
    else
        dir_list="$PWD"
        user_name="$USER"
    fi
    echo "<h2>Home Space Utilization ($user_name)</h2>"
    for i in $dir_list; do
        total_files="$(find "$i" -type f | wc -l)"
        total_dirs="$(find "$i" -type d | wc -l)"
        total_size="$(du -sh "$i" | cut -f 1)"
        echo "<H3>$i</H3>"
        echo "<pre>"
        printf "$format" "Dirs" "Files" "Size"
        printf "$format" "----" "-----" "----"
        printf "$format" "$total_dirs" "$total_files"
"$total_size"
        echo "</pre>"
    done
    return
}

# call the main function
main "$@"
```

```
vim sys_info.sh
```

Only `report_home_space ()` (the last function) has been modified. It provides more detail for each user's home directory and includes the total number of files and subdirectories in each. We also use some local variables and use `printf` (instead of `echo`) to format some of the output.

```
./sys_info.sh > report.html
```

```
lynx report.html
```

DOWNLOAD LESSON13/PART1.CAST

2. Variable expansions

1. We have already seen that sometimes we need to surround variable names in braces, to avoid any confusion:

```
a="foo"
```

```
echo "$a_file"
```

```
echo "${a}_file"
```

2. Get a default value if the variable is unset (or empty): `${parameter:-word}`

```
foo=
```

```
echo ${foo:-"substitute value if unset"}
```

```
echo $foo
```

```
foo=bar
```

```
echo ${foo:-"substitute value if unset"}
```

```
echo $foo
```

3. Assign a default value if the variable is unset (or empty): `${parameter:=word}`

```
foo=
```

```
echo ${foo:="default value if unset"}
```

```
echo $foo
```

```
foo=bar
```

```
echo ${foo:="default value if unset"}
```

```
echo $foo
```

   Note: Positional and other special parameters cannot be assigned this way.

4. Exit with an error message if the parameter is unset or empty: `${parameter:?word}`

```
foo=
```

```
echo ${foo:?"parameter is empty"}
```

```
echo $?
```

```
foo=bar
```

```
echo ${foo:?"parameter is empty"}
```

```
echo $?
```

5. Return the given value only if the parameter is not empty: `${parameter:+word}`

```
foo=
```

```
echo ${foo:+"substitute value if set"}
```

```
foo=bar
```

```
echo ${foo:+"substitute value if set"}
```

6. Return the names of variables that begin with a prefix: `${!prefix*}` or `${!prefix@}`

```
echo ${!BASH*}
```

```
echo ${!BASH@}
```

> ⓘ **DOWNLOAD LESSON13/PART2.CAST**

3. String operations

1. Length of the string: `${#parameter}`

```
foo="This string is long."
```

```
echo "'$foo' is ${#foo} characters long."
```

However, `${#@}` and `${#*}` give the number of the positional parameters.

2. Extract a substring:
`${parameter:offset}`
`${parameter:offset:length}`

```
echo ${foo:5}
```

```
echo ${foo:5:6}
```

```
echo ${foo: -5}
```

```
echo ${foo: -5:2}
```

Notice that a space is needed before the `-` in order to avoid confusion with a default value.

3. Remove text from the beginning and from the end:
`${parameter#pattern}`
`${parameter##pattern}`

`${parameter%pattern}`
`${parameter%%pattern}`

```
foo=file.txt.zip
```

```
echo ${foo#*.}
```

```
echo ${foo##*.}
```

```
echo ${foo%.*}
```

```
echo ${foo%%.*}
```

4. Replace:
   `${parameter/pattern/string}`
   `${parameter//pattern/string}`
   `${parameter/#pattern/string}`
   `${parameter/%pattern/string}`

```
foo=XYZ.XYZ.XYZ
```

```
echo ${foo/XYZ/ABC}
```

```
echo ${foo//XYZ/ABC}
```

```
echo ${foo/%XYZ/ABC}
```

```
```

```
echo ${foo/#XYZ/ABC}
```

If the replacement is omitted, then the matched pattern will be deleted.

```
echo ${foo/XYZ}
```

```
echo ${foo//XYZ}
```

```
echo ${foo/%XYZ}
```

```
echo ${foo/#XYZ}
```

5. Let's modify the previous `longest-word` example to use `${#j}` instead of `$(echo -n "$j" | wc -c)` for getting the length of a word:

```
diff -u longest-word3.sh longest-word2.sh
```

▾ diff -u longest-word3.sh longest-word2.sh

```
--- longest-word2.sh 2023-06-28 01:48:25.000000000 +0000
+++ longest-word3.sh 2023-09-30 16:13:27.942258976 +0000
@@ -7,7 +7,7 @@
         max_word=
         max_len=0
         for j in $(strings "$i"); do
-            len="$(echo -n "$j" | wc -c)"
+            len="${#j}"
             if (( len > max_len )); then
                 max_len="$len"
                 max_word="$j"
```

```
vim longest-word3.sh
```

longest-word3.sh

```
#!/bin/bash

# longest-word2: find longest string in a file

for i; do
    if [[ -r "$i" ]]; then
        max_word=
        max_len=0
        for j in $(strings "$i"); do
            len="${#j}"
            if (( len > max_len )); then
                max_len="$len"
                max_word="$j"
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done
```

```
ls -l /usr/bin > dirlist-usr-bin.txt
```

```
./longest-word3.sh dirlist-usr-bin.txt
```

It is not only simpler, but also more efficient:

```
time ./longest-word3.sh dirlist-usr-bin.txt
```

```
time ./longest-word2.sh dirlist-usr-bin.txt
```

6. Case conversion:

```
```

```
foo=ABCD
```

```
echo ${foo,}
```

```
echo ${foo,,}
```

```
foo=abcd
```

```
echo ${foo^}
```

```
echo ${foo^^}
```

We can also declare a variable to keep only uppercase or lowercase content:

```
declare -u foo
```

```
foo=aBcD
```

```
echo $foo
```

```
declare -l foo
```

```
foo=aBcD
```

```
echo $foo
```

```
unset foo
```

```
foo=aBcD
```

```
echo $foo
```

DOWNLOAD LESSON13/PART3.CAST

4. Arithmetic evaluation and expansion

We have seen before `$((expression))` where *expression* is an arithmetic expression. It is related to the compound command `((...))` which is used for arithmetic evaluation (truth tests). Here we will see some more arithmetic operators and expressions.

1. By default numbers are treated as decimals (base 10). But we can also use octal numbers (base 8), hexadecimal numbers (base 16), etc.

```
echo $((99))    # decimal
```

```
echo $((077))    # octal
```

```
echo $((0xff))    # hexadecimal
```

```
echo $((2#11))    # binary
```

```
echo $((7#66))    # base 7
```

2. Arithmetic operators:

```
echo $((5 + 2))
```

```
echo $((5 - 2))
```

```
echo $((5 * 2))
```

```
echo $((5 ** 2))
```

```
echo $((5 / 2))
```

```
echo $((5 % 2))
```

An example:

```
vim modulo.sh
```

modulo.sh

```
#!/bin/bash

# modulo: demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( remainder == 0 )); then
        printf "<%d> " "$i"
    else
        printf "%d " "$i"
    fi
done
printf "\n"
```

```
./modulo.sh
```

3. Assignment:

```
foo=
```

```
echo $foo
```

```
if (( foo = 5 )); then echo "It is true."; fi
```

```
echo $foo
```

The `=` sign above makes an assignment, and this assignment is successful. To check for equality we can use `==`.

Other assignment operators are: `+=`, `-=`, `*=`, `/=`, `%=`.

There are also incremental/decremental operators: `++` and `--`

```
foo=1
echo $((foo++))
echo $foo
```

```
foo=1
echo $((++foo))
echo $foo
```

Let's see a modified version of the `modulo.sh` example:

```
vim modulo2.sh
```

▼ `modulo2.sh`

```
#!/bin/bash

# modulo: demonstrate the modulo operator

for ((i = 0; i <= 20; ++i)); do
    if ((i % 5 == 0)); then
        printf "<%d> " "$i"
    else
        printf "%d " "$i"
    fi
done
printf "\n"
```

```
diff -u modulo.sh modulo2.sh
```

diff -u modulo.sh modulo2.sh

```
--- modulo.sh    2023-06-28 01:48:25.000000000 +0000
+++ modulo2.sh   2023-06-28 01:48:25.000000000 +0000
@@ -2,9 +2,8 @@

 # modulo: demonstrate the modulo operator

-for ((i = 0; i <= 20; i = i + 1)); do
-    remainder=$((i % 5))
-    if (( remainder == 0 )); then
+for ((i = 0; i <= 20; ++i)); do
+    if ((i % 5 == 0)); then
         printf "<%d> " "$i"
     else
         printf "%d " "$i"
```

```
./modulo2.sh
```

4. There are also some operators that work at the bit level:

   - `~` -- Negate all the bits in a number.
   - `<<` -- Shift all the bits in a number to the left.
   - `>>` -- Shift all the bits in a number to the right.
   - `&` -- Perform an AND operation on all the bits in two numbers.
   - `|` -- Perform an OR operation on all the bits in two numbers.
   - `^` -- Perform an exclusive OR operation on all the bits in two numbers.

   There are also corresponding assignment operators (for example, `<<=`) for all but bitwise negation.

   Let's see an example that prints the powers of 2:

```
for ((i=0;i<8;++i)); do echo $((1<<i)); done
```

5. The compound command `((...))` supports also comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&` (logical AND), `||` (logical OR).

   It also supports the ternary operator: `expr1?expr2:expr3`. If expression `expr1` evaluates to be non-zero (arithmetic true), then `expr2`; else `expr3`.

   Logical expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The `((...))` compound command maps the results into the shell's normal exit codes.

```
if ((1)); then echo "true"; else echo "false"; fi
```

```
if ((0)); then echo "true"; else echo "false"; fi
```

The ternary operator is like a compact `if/then/else` statement:

```
a=0
```

```
((a<1?++a:--a))
```

```
echo $a
```

```
((a<1?++a:--a))
```

```
echo $a
```

```
a=$((a<1?a+1:a-1))
```

```
echo $a
```

6. Let's see a more complete example of using arithmetic operators in a script that produces a simple table of numbers.

```
vim arith-loop.sh
```

arith-loop.sh

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
```

```
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" "$a" "$b" "$c"
    ((a<10?++a:(finished=1)))
done
```

```
./arith-loop.sh
```

7. For complex arithmetics we can use `bc`, which is an arbitrary precision calculator.

```
bc <<< '2 + 2'
```

```
echo '2 + 2' | bc
```

This example script calculates monthly loan payments:

```
vim loan-calc.sh
```

`loan-calc.sh`

```
#!/bin/bash

# loan-calc: script to calculate monthly loan payments

PROGNAME="${0##*/}" # Use parameter expansion to get basename
```

```
usage () {
    cat <<- EOF
        Usage: $PROGNAME PRINCIPAL INTEREST MONTHS
        Where:
        PRINCIPAL is the amount of the loan.
        INTEREST is the APR as a number (7% = 0.07).
        MONTHS is the length of the loan's term.
        EOF
}

if (($# != 3)); then
    usage
    exit 1
fi

principal=$1
interest=$2
months=$3

bc <<- EOF
        scale = 10
        i = $interest / 12
        p = $principal
        n = $months
        a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
        print a, "\n"
EOF
```

```
./loan-calc.sh 135000 0.0775 180
```

This example calculates the monthly payment for a $135,000 loan at 7.75 percent APR for 180 months (15 years). Notice the precision of the answer. This is determined by the value given to the special `scale` variable in the `bc` script.

For more details about `bc` see:

```
man bc
```

```
info bc
```

DOWNLOAD LESSON13/PART4.CAST

📄 Intro

In this lesson we will see:

📄 1. Arrays

Arrays are variables that hold more than one value at a time.

📄 2. Group commands. Subshells. Process substitutio…

1. Group commands and subshells.

📄 3. Traps. Asynchronous execution. Named pipes.

1. We know that programs can respond to signals. We can add this

Intro

In this lesson we will see:

- arrays
- group commands
- subshell
- process substitution
- traps
- asynchronous execution
- named pipes

> **NOTE**
>
> Let's get first some examples:

```
mkdir -p examples  &&  cd examples/

wget https://linux-cli.fs.al/examples/lesson14.tgz

tar xfz lesson14.tgz

cd lesson14/  &&  ls
```

> **DOWNLOAD LESSON14/INTRO.CAST**

1. Arrays

Arrays are variables that hold more than one value at a time.

1. Getting started:

```
a[0]=foo
```

```
echo ${a[0]}
```

```
days=(Sun Mon Tue Wed Thu Fri Sat)
```

```
echo ${days[0]}
```

```
echo ${days[*]}
```

```
echo $days[*]
```

```
echo $days
```

If no index is given, the first item is returned.

```
days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

```
echo ${days[*]}
```

2. Let's see an example that counts files by modification time and shows them in a table. Such a script could be used to determine when a system is most active.

```
./hours.sh
```

```
./hours.sh /usr/bin/
```

```
vim hours.sh
```

hours.sh

```
#!/bin/bash

# hours: script to count files by modification time

usage () {
    echo "usage: ${0##*/} directory" >&2
}

# Check that argument is a directory
if [[ ! -d "$1" ]]; then
    usage
    exit 1
fi

# Initialize array
for i in {0..23}; do hours[i]=0; done

# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
    j="${i#0}"
    ((++hours[j]))
    ((++count))
done
```

```
# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
    j=$((i + 12))
    printf "%02d\t%d\t%02d\t%d\n" \
        "$i" \
        "${hours[i]}" \
        "$j" \
        "${hours[j]}"
done
printf "\nTotal files = %d\n" $count
```

To get the last modification time of the files we use the command `stat`:

```
stat --help | less
```

```
stat -c %y *
```

```
stat -c %y * | cut -c 12-13
```

We use the hour of modification as an index for the array.

3. Outputting the entire contents of an array:

```
animals=("a dog" "a cat" "a fish")
```

```
for i in ${animals[*]}; do echo $i; done
```

```
for i in ${animals[@]}; do echo $i; done
```

```
for i in "${animals[*]}"; do echo $i; done
```

```
for i in "${animals[@]}"; do echo $i; done
```

Notice that this is similar to the behavior of the array of the positional parameters: `$*`, `$@`, `"$*"`, `"$@"`.

4. The number of array elements:

```
a[100]=foo
```

```
echo ${#a[@]}
```

There is only one element in the array.

```
echo ${#a[100]}
```

This is the length of the element 100.

Remember that `$#` is the number of positional parameters.

5. Finding the subscripts used by an array:

```
foo=([2]=a [4]=b [6]=c)
```

```
for i in "${foo[@]}"; do echo $i; done
```

```
for i in "${!foo[@]}"; do echo $i; done
```

6. Adding elements to the end of an array:

```
foo=(a b c)
```

```
echo ${foo[@]}
```

```
foo+=(d e f)
```

```
echo ${foo[@]}
```

7. It is not so hard to sort an array with a little bit of coding:

```
vim array-sort.sh
```

array-sort.sh

```
#!/bin/bash

# array-sort: Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
#a_sorted=($(echo "${a[@]}" | tr ' ' "\n" | sort))
echo "Sorted array:   ${a_sorted[@]}"
```

```
./array-sort.sh
```

8. To delete an array, use the `unset` command:

```
foo=(a b c d e f)
```

```
echo ${foo[@]}
```

```
unset foo
```

```
echo ${foo[@]}
```

It may also be used to delete single array elements.

```
foo=(a b c d e f)
```

```
echo ${foo[@]}
```

```
unset 'foo[2]'
```

```
echo ${foo[@]}
```

Notice that the array element must be quoted to prevent the shell from performing pathname expansion.

9. Notice also that the assignment of an empty value to an array does not empty its contents:

```
foo=(a b c d e f)
```

```
foo=
```

```
echo ${foo[@]}
```

This is because any reference to an array variable without a subscript refers to element zero of the array. For example:

```
foo=(a b c d e f)
```

```
echo ${foo[@]}
```

```
foo=A
```

```
echo ${foo[@]}
```

10. Associative arrays use strings rather than integers as array indexes:

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

Associative arrays must be created with `declare -A`. Its elements are accessed in the same way as the integer indexed arrays:

```
echo ${colors["green"]}
```

> **DOWNLOAD LESSON14/PART1.CAST**

2. Group commands. Subshells. Process substitutions.

1. Group commands and subshells.

   Group command: `{ command1; command2; [command3; ...] }`

   Subshell: `(command1; command2; [command3;...])`

   Because of the way bash implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace.

   Group commands and subshells are both used to manage redirection:

   ```
   date > foo.txt
   ```

   ```
   ls -l > output.txt
   echo "Listing of foo.txt" >> output.txt
   cat foo.txt >> output.txt
   ```

   ```
   { ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
   ```

   ```
   (ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
   ```

   ```
   { ls -l; echo "Listing of foo.txt"; cat foo.txt; } | less
   ```

2. Let's see an example that prints a listing of the files in a directory, along with the names of the file's owner and group owner. At the end of the listing, the

script prints a tally of the number of files belonging to each owner and group.

```
./array-2.sh /usr/bin
```

```
vim array-2.sh
```

array-2.sh

```
#!/bin/bash

# array-2: Use arrays to tally file owners

declare -A files file_group file_owner groups owners

if [[ ! -d "$1" ]]; then
    echo "Usage: ${0##/} dir" >&2
    exit 1
fi

for i in "$1"/*; do
    owner="$(stat -c %U "$i")"
    group="$(stat -c %G "$i")"
    files["$i"]="$i"
    file_owner["$i"]="$owner"
    file_group["$i"]="$group"
    ((++owners[$owner]))
    ((++groups[$group]))
done

# List the collected files
{
  for i in "${files[@]}"; do
      printf "%-40s %-10s %-10s\n" \
          "$i" "${file_owner["$i"]}" "${file_group["$i"]}"
  done
} | sort
echo
```

```
# List owners
echo "File owners:"
{
  for i in "${!owners[@]}"; do
      printf "%-10s: %5d file(s)\n" "$i" "${owners["$i"]}"
  done
} | sort
echo

# List groups
echo "File group owners:"
{
  for i in "${!groups[@]}"; do
      printf "%-10s: %5d file(s)\n" "$i" "${groups["$i"]}"
  done
} | sort
```

3. A group command (`{ . . . }`) executes all of its commands in the current shell.

   A subshell (`( . . . )`), as the name suggests, executes its commands in a child copy of the current shell. This means the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell's environment (including variable assignment) are lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to subshells. Group commands are both faster and require less memory.

   We have seen before that using `read` with pipe does not work as we might expect:

```
echo "foo" | read
```

```
echo $REPLY
```

This is because the shell executes the command after the pipe (`read` in this case) in a subshell. The command `read` assigns a value to the variable `REPLAY` in the environment of the subshell, but once the command is done executing, the subshell and its environment are destroyed. So, the variable `REPLAY` of the current shell still is unassigned (does not have a value).

4. To work around this problem, shell provides a special form of expansion, called *process substitution*.

   For processes that produce standard output it looks like this: `<(list-of-commands)`

   For processes that intake standard input it looks like this: `>(list-of-commands)`

   To solve our problem with read, we can employ process substitution like this:

   ```
   read < <(echo "foo")
   ```

   ```
   echo $REPLY
   ```

   What is happening is that *process substitution* allows us to treat the output of a subshell as an ordinary file for purposes of redirection.

   ```
   echo <(echo "foo")
   ```

   By using `echo` we see that the output of the subshell is being provided by a file named `/dev/fd/63`.

5. Let's see an example of a `read` loop that processes the contents of a directory listing created by a subshell:

   ```
   vim pro-sub.sh
   ```

---

pro-sub.sh

```
#!/bin/bash

# pro-sub: demo of process substitution

while read attr links owner group size d1 d2 d3 filename; do
    cat <<- EOF
        Filename:   $filename
        Size:       $size
        Owner:      $owner
        Group:      $group
        Modified:   $d1 $d2 $d3
        Links:      $links
        Attributes: $attr

        EOF
done < <(ls -lh $1 | tail -n +2)
```

Because we are using `read`, we cannot use a pipe to send data to it.

```
./pro-sub.sh
```

```
./pro-sub.sh /usr/bin | less
```

DOWNLOAD LESSON14/PART2.CAST

3. Traps. Asynchronous execution. Named pipes.

1. We know that programs can respond to signals. We can add this capability to our scripts too. Bash provides a mechanism for this purpose known as a *trap*. Let's see a simple example:

```
vim trap-demo.sh
```

trap-demo.sh

```
#!/bin/bash

# trap-demo: simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

When we press Ctrl-c while the script is running, the script will intercept the signal and will respond to it by running the `echo` command. Let's try it:

```
./trap-demo.sh
```

Press Ctrl-c a few times and see what happens.

2. It is more convenient to tell `trap` to call a function in response to a signal, instead of a complex command. Let's see another example:

```
vim trap-demo2.sh
```

`trap-demo2.sh`

```
#!/bin/bash

# trap-demo2: simple signal handling demo

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0
}

exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

Note the inclusion of an exit command in each of the signal-handling functions. Without an exit, the script would continue after completing the function.

```
./trap-demo2.sh
```

Press Ctrl-c.

3. Bash has a builtin command to help manage *asynchronous execution*. The `wait` command causes a parent script to pause until a specified process (i.e., the child script) finishes.

   This can be best explained by an example. We will need two scripts, a parent script, and a child script:

```
vim async-child.sh
```

<details>
<summary>async-child.sh</summary>

```
#!/bin/bash

# async-child: Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."
```

</details>

   This is a simple script that runs for 5 seconds.

```
vim async-parent.sh
```

<details>
<summary>async-parent.sh</summary>

```
#!/bin/bash

# async-parent: Asynchronous execution demo (parent)

echo "Parent: starting..."
```

```
echo "Parent: launching child script..."
./async-child.sh &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2

echo "Parent: pausing to wait for child to finish..."
wait "$pid"

echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

From this script we launch the child script. Since we are appending `&` after it, the parent script will not wait for the child to finish executing but will continue running. Both of the scripts are now running in parallel. Immediately after launching the child, the parent uses the special variable `$!` to get the process ID (PID) of the child. This variable always contains the PID of the last job put into the background. Then, later in the parent script, we use the command `wait` to stop the parent from running any further, until the child script is finished. Let's try it:

```
./async-parent.sh
```

All the messages output from the parent are prefixed with `Parent:` and all the messages output from the child are prefixed with `Child:`. This helps us understand the flow of execution.

4. *Named pipes* behave like files but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed) pipes, data goes in one end and emerges out the other.

   With named pipes, it is possible to set up something like this: `process1 > named_pipe`, an this: `process2 < named_pipe` and it will behave like this:

`process1 | process2`. The only difference is that `process1` and `process2` run in the current shell, not in a subshell, which makes named pipes more useful, even if they are a bit less convenient than using a pipe operator (`|`).

A named pipe can be created with the command `mkfifo`:

```
mkfifo pipe1
```

```
ls -l pipe1
```

Notice that the first letter in the attributes field is "p", indicating that it is a named pipe.

```
ls -l > pipe1 &
```

```
cat < pipe1
```

This is similar to:

```
ls -l | cat
```

However the named pipe is more flexible, because the two commands connected by the pipe can be executed even in different terminals.

However these two examples are not the same thing:

```
echo "abc" | read
```

```
echo $REPLY
```

```
echo "abc" > pipe1 &
```

```
read < pipe1
```

```
echo $REPLY
```

Let's remove `pipe1`:

```
rm pipe1
```

5. Another example with a named pipe:

```
mkfifo pipe1
```

```
while true; do read line < pipe1; echo "You said: '$line'"; done &
```

```
echo Hi > pipe1
```

```
echo Hello > pipe1
```

```
echo "The quick brown fox jumped over the lazy dog." > pipe1
```

```
fg
```

Press Ctrl-c to stop the `while` loop.

```
```

```
rm pipe1
```

DOWNLOAD LESSON14/PART3.CAST

📄 Intro

In this lesson we will study the scripts that solve some simple

📄 Examples 1

1. This is a simple script that gets as argument the URL of a web page

📄 Examples 2

1. This is a script that calculates the Fibonacci numbers, using an

📄 Examples 3

Write a bash script that collects web pages from the internet,

Intro

In this lesson we will study the scripts that solve some simple problems:

- A script that gets as argument the URL of a web page and returns all the URLs inside that page.
- A script that gets as argument the URL of a web page and returns a list of the 100 most frequently used words inside it.
- A script that calculates the Fibonacci numbers, using an iterative algorithm.
- A script that solves the problem of Towers of Hanoi, using a recursive algorithm.
- A script that collects web pages from the internet, starting from a root page, extracts the words in each page, and creates a list of words.

NOTE

Let's get first the examples:

```
mkdir -p examples  &&  cd examples/

wget https://linux-cli.fs.al/examples/lesson15.tgz

tar xfz lesson15.tgz

cd lesson15/  &&  ls
```

DOWNLOAD LESSON15/INTRO.CAST

Examples 1

1. This is a simple script that gets as argument the URL of a web page and returns all the URLs inside that page:

get_urls.sh

```
#!/bin/bash
# Get all the URLs inside a given HTML page.

PAGE=$1

if [[ -z $PAGE ]]; then
    echo "Usage: $0 <html-page>" >&2
    exit 1
fi

wget -qO- "$PAGE" \
    | grep -Eoi '<a [^>]+>' \
    | grep -Eo 'href="?([^\"]+)"?' \
    | grep -v 'mailto:' \
    | sed -e 's/"//g' -e 's/href=//'
```

```
vim get_urls.sh
```

```
./get_urls.sh
```

```
url=http://linuxcommand.org/
```

```
./get_urls.sh $url
```

Let's see how it works:

```
wget -qO- $url
```

```
wget -qO- $url | grep -Eoi '<a [^>]+>'
```

The option `-E` is for extended regexp syntax, `-o` is for displaying only the matching part, and `-i` is for case insensitive. We are extracting all the *anchor* tags.

```
wget -qO- $url | grep -Eoi '<a [^>]+>' | grep -Eo 'href="?
([^\"]+)"?'
```

Extracting the attribute `href`.

```
wget -qO- $url \
    | grep -Eoi '<a [^>]+>' \
    | grep -Eoi 'href="?([^\"]+)"?' \
    | grep -v 'mailto:' \
    | sed -e 's/"//g' -e 's/href=//'
```

2. This is a simple script that gets as argument the URL of a web page and returns a list of the 100 most frequently used words inside it:

<details>
<summary>get_words.sh</summary>

```
#!/bin/bash
# Return a list of the 100 most frequently used words inside a
given page.
```
</details>

```
PAGE=$1

if [[ -z $PAGE ]]; then
    echo "Usage: $0 <html-page>" >&2
    exit 1
fi

wget -q -O- "$PAGE" \
    | tr "\n" ' ' \
    | sed -e 's/<[^>]*>/ /g' \
    | sed -e 's/&[^;]*;/ /g' \
    | tr -cs A-Za-z\' '\n' \
    | tr A-Z a-z \
    | sort \
    | uniq -c \
    | sort -k1,1nr -k2 \
    | sed 100q \
    | sed -E 's/^ +//' \
    | cut -d' ' -f2
```

```
vim get_words.sh
```

```
./get_words.sh
```

```
url=https://en.wikipedia.org/wiki/Linux
```

```
./get_words.sh $url
```

```
./get_words.sh $url | less
```

```
./get_words.sh $url | wc -l
```

```
wget -qO- $url \
   | tr "\n" ' ' \
   | sed -e 's/<[^>]*>/ /g' \
   | sed -e 's/&[^;]*;/ /g' \
   | tr -cs A-Za-z\' '\n' \
   | tr A-Z a-z \
   | less
```

```
wget -qO- $url \
   | tr "\n" ' ' \
   | sed -e 's/<[^>]*>/ /g' \
   | sed -e 's/&[^;]*;/ /g' \
   | tr -cs A-Za-z\' '\n' \
   | tr A-Z a-z \
   | sort \
   | uniq -c \
   | less
```

```
wget -qO- $url \
   | tr "\n" ' ' \
   | sed -e 's/<[^>]*>/ /g' \
   | sed -e 's/&[^;]*;/ /g' \
   | tr -cs A-Za-z\' '\n' \
   | tr A-Z a-z \
   | sort \
   | uniq -c \
   | sort -k1,1nr -k2 \
   | sed 100q \
   | less
```

DOWNLOAD LESSON15/PART1.CAST

Examples 2

1. This is a script that calculates the Fibonacci numbers, using an iterative algorithm:

fibo.sh

```
#!/bin/bash

N=$1
if [[ -z $N ]]; then
    echo "Usage: $0 <n>" >&2
    exit 1
fi

[[ $N == 0 ]] && echo 0 && exit
[[ $N == 1 ]] && echo 1 && exit

f0=0
f1=1
for ((i=1; i<N; i++)); do
    fib=$((f0 + f1))
    f0=$f1
    f1=$fib
done
echo $f1
```

```
vim fibo.sh
```

```
./fibo.sh
```

```
for i in {0..10}; do ./fibo.sh $i; done
```

```
./fibo.sh 100
```

2. This is a script that solves the problem of Towers of Hanoi, using a recursive algorithm:

hanoi.sh

```
#!/bin/bash

solve_hanoi() {
    local disks=$1 src=$2 dst=$3 aux=$4
    if ((disks > 0)); then
        solve_hanoi $((disks - 1)) $src $aux $dst
        echo "move $src --> $dst"
        ((nr_moves++))
        solve_hanoi $((disks - 1)) $aux $dst $src
    fi
}

read -p "Towers of Hanoi. How many disks? " disks

nr_moves=0      # start with no moves
solve_hanoi $disks 'src' 'dst' 'aux'

echo "It took $nr_moves moves to solve Towers for $disks disks."
```

```
vim hanoi.sh
```

```
./hanoi.sh <<< 0
```

```
./hanoi.sh <<< 1
```

```
./hanoi.sh <<< 2
```

```
./hanoi.sh <<< 3
```

```
./hanoi.sh <<< 4
```

```
./hanoi.sh <<< 5
```

DOWNLOAD LESSON15/PART2.CAST

Examples 3

Write a bash script that collects web pages from the internet, starting from a root page, extracts the words in each page, and creates a list of words.

```
vim crawler.sh
```

crawler.sh

```
#!/bin/bash

# stop the program after running for this many seconds
RUNTIME=10

check() {
    # check dependencies
    hash lynx || { echo "Please install lynx" >&2; exit 1; }
    hash w3m || { echo "Please install w3m" >&2; exit 1; }

    # check that there is an argument
    [[ -z $1 ]] && { echo "Usage: $0 <url>" >&2; exit 1; }
}

start_timer() {
    {
        sleep $RUNTIME
        echo "...Timeout..."
        local nr_words=$(cat tmp/words.txt | wc -l)
        echo "The number of collected words: $nr_words"
        kill -9 $$ >/dev/null 2>&1
    } &
}

get_urls() {
    lynx "$1" -listonly -nonumbers -dump 2>/dev/null \
```

```
        | grep -E '^https?://' \
        | sed -e 's/#.*$//'
}

get_words() {
    w3m -dump "$1" \
        | sed -e "s/[^[:alnum:]' -]\+//g" \
        | tr -cs A-Za-z\' '\n' \
        | tr A-Z a-z \
        | sort -u
}

get_words_1() {
    wget -q -O- "$1" \
        | tr "\n" ' ' \
        | sed -e 's/<[^>]*>/ /g' \
        | sed -e 's/&[^;]*;/ /g' \
        | tr -cs A-Za-z\' '\n' \
        | tr A-Z a-z \
        | sort -u
}

main() {
    check "$@"
    start_timer

    mkdir -p tmp/
    rm -f tmp/{todo,done}.txt
    touch tmp/{todo,done,words}.txt

    echo "$1" > tmp/todo.txt
    local url
    while true; do
        # pop the top url from todo.txt
        url=$(head -1 tmp/todo.txt)
        sed -i tmp/todo.txt -e 1d

        # check whether we have processed already this url
        grep -qF "$url" tmp/done.txt && continue
        echo "$url"

        # extract the links from it and append them to todo.txt
```

```
            get_urls "$url" >> tmp/todo.txt

            # extract all the words from this url
            get_words "$url" >> tmp/words.txt
            sort -u tmp/words.txt > tmp/words1.txt
            mv tmp/words1.txt tmp/words.txt

            # mark this url as processed
            echo "$url" >> tmp/done.txt
        done
}

# call the main function
main "$@"
```

The file `tmp/todo.txt` contains a list of URLs, one per line, that are to be visited. Initially we add to it the root URL that is given as an argument.

The file `tmp/done.txt` contains a list of URLs, one per line, that are already visited. The file `tmp/words.txt` contains a list of words that have been collected so far, one per line and sorted alphabetically.

There is an infinite loop in which we do these steps:

1. Get the top URL from `tmp/todo.txt` (and delete it from the file).
2. If not valid (does not start with `http://` or `https://`), continue with the next URL.
3. If valid, extract all the URLs on this page and append them to `tmp/todo.txt`, in order to visit them later.
4. Get all the words from this page and merge them to `tmp/words.txt`.
5. Append this URL to `tmp/done.txt`, so that we don't process it again.

This infinite loop is never stopped, but there is a timer which stops the program after running for a certain amount of seconds.

```
./crawler.sh
```

```
sudo apt install -y w3m
```

```
./crawler.sh
```

```
url=https://en.wikipedia.org/wiki/Linux
```

```
./crawler.sh $url
```

```
less tmp/words.txt
```

Let's increase the `RUNTIME`:

```
sed -i crawler.sh -e '/^RUNTIME=/ c RUNTIME=100'
```

```
rm -rf tmp/
```

```
./crawler.sh $url
```

```
cat tmp/words.txt | wc -l
```

```
less tmp/words.txt
```

> **DOWNLOAD LESSON15/PART3.CAST**