
# Lecture 2: Commands and variables

**Last update**: 20200505

## Table of Contents



## 1. Introduction

When it comes to the operating systems nowadays, in high-energy experimental physics we mostly rely on **Linux**. That being said, as an experimental physicist you are sooner or later faced with the following situation: You have turned on your computer and launched the terminal...


... and what now??

You can start clicking with the mouse over the terminal, but quickly you will realize that this leads you nowhere. Next, you can start typing and pressing 'Enter', but especially if you do it for the first time most likely whatever you have typed in the terminal will produce only the error messages. Still, that is something, as it clearly means that there is some secret/magic language which is trying to respond to, or to interpret, your command input, as soon as you have typed something in the terminal and pressed 'Enter'.

What is that secret built-in language available in the terminal? This lecture is all about shedding light on its existence...

Loosely speaking, **shell** is the generic name of any program that user employs to type commands in the terminal (a.k.a. text window). Example **shells**:

- sh
- bash
- ksh
- csh
- fish

To get the list of **shells** available on your computer, type in the terminal:

```
cat /etc/shells
```

The output of that command could look like:

```
/etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
/usr/bin/fish
```

How to select your favorite **shell**? It's simple, just type its name in the terminal and press 'Enter'! E.g. if you want to use **Bash** as your working **shell**, just type in the terminal:

```
bash
```

and press 'Enter' --- now you are in the **Bash** wonderland! Since that is by far the most popular **shell** nowadays, this lecture will focus exclusively on its concepts, syntax and commands. But no worries, at least conceptually, a lot of subjects covered in this lecture apply also to other **shells**! The difference between **shells** is mostly in the syntax, but in essence, they all aim to provide the same functionalities.

In this lecture we will cover only the **Bash** essentials, i.e. we will make you going, but how far you want to go eventually, it depends on your personal determination and time investment.

Traditionally, the first program people write when learning a new programming language is the so-called *"Hello World"* example. Let's keep up with this tradition and type in the terminal:

```
echo "Hello World"
```

This will output in the terminal:

```
Hello World
```

i.e. **Bash** has echoed back the text you have typed in the terminal as an *argument* to **echo** command. Let's move on!

## 2. Shell environment: commands and variables

When you open a terminal, your local environment is defined via some command names and predefined variables, which you can use directly in the current terminal session. Before going more into the details of how to modify the **shell** environment, let's see first how commands and variables are used in general.

### A) Commands

We have already seen how one built-in **Bash** command works, namely **echo**. In the same spirit, we can use in the terminal any other **Linux** command, not necessarily the built-in **Bash** command.

**Example:** What is the current time? Just type in the terminal **date** command and press 'Enter'

```
date
```

This will output in the terminal something like:

```
Mon Apr 20 14:49:10 CEST 2020
```

This is the default formatting of **date** command. Now use the command **date** with the flag (or option) **-u**, in order to modify its default behavior:

```
date -u
```

This will output in the terminal the different text stream:

```
Mon Apr 20 12:49:12 UTC 2020
```

After passing a certain flag, we have instructed command **date** to change its default behavior. In the above example, the flag **-u** caused the command **date** to report time in Coordinated Universal Time format (UTC), instead of Central European Summer Time (CEST) time zone (which is the default for our location).

As another example, we have already seen how we can use **cat** command to view the content of file when inquiring which **shells** are available on the computer. We can pass to command **cat** as arguments more files to read in one go, i.e. we can execute the same command in one go on multiple arguments:

```
cat /etc/shells /etc/hostname
```

The output of that command now could look like:

```
/etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
/usr/bin/fish
transfer.ktas.ph.tum.de
```

Note the new entry in the last line, which is the content of file `/etc/hostname` .

We can, of course, combine options and arguments when invoking a command:

```
cat -n /etc/shells
     1  # /etc/shells: valid login shells
     2  /bin/sh
     3  /bin/dash
     4  /bin/bash
     5  /bin/rbash
     6  /usr/bin/screen
     7  /usr/bin/fish
```

The flag **-n** causes command **cat** to enumerate all lines in the printout.

Based on these simple examples, we now establish the following general statements about commands. In general, all **Bash** and **Linux** commands are conceptually implemented in the same way --- let us now discuss what is conceptually always the same in their implementation and usage.

Generically, for most cases of interest, we are executing commands in the terminal in the following way:

```
<command-name> <option(s)> <argument(s)>
```

This is the right moment to stress the importance and profound meaning of empty character: Empty character is the default input field separator (**IFS**) in the world of **Linux**. If you misuse the empty character, a lot of your input in the terminal will be completely incomprehensible to **Bash**, and to **Linux** commands in general. In the above generic example, empty character separates the three items, which conceptually have a completely different meaning. As the very first step, after you have typed the input in the terminal and pressed 'Enter', the **Bash** splits your input into tokens that are separated (by default, and in a bit simplified picture) with one or more empty characters. Then, it checks whether the very first token is some known **Linux** command, **Bash** keyword, etc.

The command input in **Bash** is terminated either by a new line or by a semi-colon `;` . It is completely equivalent to write:

```
echo "Hello World"
date
```

or

```
echo "Hello World";date
```

or

```
echo "Hello World" ;   date
```

Let us now scrutinize the above generic syntax for command execution term by term:

- `<command-name>` : Whatever you type first in the terminal, i.e. before the next empty character is being encountered on terminal input, **Bash** is trying to interpret as some known **Linux** command, **Bash** keyword, etc. In general, *command-name* can stand for one of the following items:

  1) **Linux** command (i.e. system-wide executable or binary) --- example: **cat**
  2) **Bash** built-in command --- example: **echo**
  3) **Bash** keyword --- example: **for**
  4) alias
  5) function
  6) script

- `<option(s)>` : Options (or flags) are used to modify the default behaviour of command. Options are indicated either with:

  1) **-** (single dash) followed by single character(s), or
  2) **--** (two consecutive dashes) followed by more descriptive explanation about what needs to be modified in the default behaviour of command.

For instance, the frequently used flags **-a** and **--all** are synonyms, in a sense that they modify the default behavior of command in exactly the same way. The first version is easier to type, but the second one is easier to memorize. Example for **date** command:

```
date -u
Mon Apr 20 12:49:12 UTC 2020
date --utc
Mon Apr 20 12:49:12 UTC 2020
```

The output in both cases above is the same, because flags **-u** and **--utc** are synonyms for **date** command.

But how do we know that for command **date** flags **-u** and **--utc** are available, and how do we know in which way they will modify the default behavior of command? All such options for each command are documented in so-called *man pages*. Whenever you develop a new command, it is also essential that you develop its documentation, otherwise nobody will be able to use your command. For built-in **Bash** commands, documentation is retrieved simply with:

```
help <command-name>
```

For **Linux** commands, we can use the command **man** (shortcut for *manual*) to retrieve the documentation. The syntax is fairly simple:

```
man <command-name>
```

**Example:** To see which options are available for built-in **Bash** command **echo**, use:

```
help echo
```

You shall get as an output something like:

```
echo: echo [-neE] [arg ...]
    Write arguments to the standard output.

    Display the ARGs on the standard output followed by a newline.

    Options:
      -n        do not append a newline
      -e        enable interpretation of the following backslash escapes
      -E        explicitly suppress interpretation of backslash escapes

    `echo' interprets the following backslash-escaped characters:
      \a        alert (bell)
      \b        backspace
      \c        suppress further output
      \e        escape character
      \f        form feed
      \n        new line
      \r        carriage return
      \t        horizontal tab
      \v        vertical tab
      \\        backslash
      \0nnn     the character whose ASCII code is NNN (octal).  NNN can be
        0 to 3 octal digits
      \xHH      the eight-bit character whose value is HH (hexadecimal).  HH
        can be one or two hex digits

    Exit Status:
    Returns success unless a write error occurs.
bash-4.2$
```

The command **help** gives a complete description of the built-in **Bash** command (e.g. **echo**, **jobs**, **read**, etc.) or of some **Bash** keyword (e.g. **for**, **if**, etc.).

**Example:** To see which options are available for **Linux** command **date**, use:

```
man date
```

The first page of a rather lengthy output could look like:

```
DATE(1)                          User Commands                          DATE(1)

NAME
       date - print or set the system date and time

SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

DESCRIPTION
       Display the current time in the given FORMAT, or set the system date.

       Mandatory arguments to long options are mandatory for short options too.

       -d, --date=STRING
              display time described by STRING, not 'now'

       -f, --file=DATEFILE
              like --date once for each line of DATEFILE

       -I[TIMESPEC], --iso-8601[=TIMESPEC]
              output  date/time  in  ISO  8601  format.  TIMESPEC='date' for date only (the
              default), 'hours', 'minutes', 'seconds', or 'ns' for date  and  time  to  the
              indicated precision.

       -r, --reference=FILE
              display the last modification time of FILE

 Manual page date(1) line 1 (press h for help or q to quit)
```

In order to exit the *man pages*, press 'q'. You can scroll down the *man page* line-by-line by pressing 'Enter', or page-by-page by pressing 'Spacebar'.

As you can see, even simple commands, like **date**, can have extensive documentation and a lot of options. It is clearly impossible to memorize all options for all commands, therefore usage of **help** and **man** commands is needed almost on a daily basis. After we have covered command names and options, we close this section with the last item: command arguments.

- `<argument(s)>` : This is simple, sometimes you want your command to be executed on the specified argument, or in one go on multiple arguments. For instance, we can make an empty file in the current working directory by using the **touch** command:

```
touch file_1.log
```

But we could create plenty of empty files with **touch** command in one go, not one-by-one, e.g.

```
touch file_1.log file_2.log file_3.log file_4.log
```

Important remark: Since the empty character is an input field separator, never use it as a part of a file or directory name! In such a context, always replace it with underscore "_" or any other character which doesn't have special meaning. For instance:

```
touch file 1.log
```

would literally create two empty files, the first one named `file`, and the second one named `1.log` . If you apply **touch** command on an already existing file, only the time-stamp of that file will be updated to the current time, its content remains exactly the same.

In order to see or list all files and subdirectories in the current working directory, use **ls** command, i.e.

```
ls -al
```

For the meaning of options **-a** and **-l** check the *man pages* of **ls** command. As a side remark, we indicate that options may or may not themselves require specific arguments. Options which do not require specific arguments, can be grouped together, i.e. **ls -a -l** is exactly the same as a shorthand **ls -al**.

In the same spirit, we can create multiple directories in one go, with **mkdir** command, e.g.

```
mkdir subdir_1 subdir_2 subdir_3
```

will make 3 new subdirectories in your current working directory (check again by executing **ls -al**).

We have been using so far only the already existing **Bash** or **Linux** commands. The simplest way to create your own command, with a rather limited functionality and flexibility but nevertheless quite convenient, is to use **Bash** built-in command **alias**. For instance, if you are bored to type something lengthy again and again in the terminal, you can introduce shortcut for it, by using **alias**. For instance, you can abbreviate the lengthy input

```
echo "Hello, welcome to the lecture PH8124"
```

into the simple new command **Welcome**, by creating an alias for it:

```
alias Welcome='echo "Hello, welcome to the lecture PH8124"'
```

Now the following new command in the terminal shall work as well:

```
Welcome
```

Press 'Enter' and the output in the terminal shall be:

```
Hello, welcome to the lecture PH8124
```

Quite frequently, aliases are used in the following context: If you want to connect from your desktop machine to some other computer, e.g. *lxplus* at CERN, you need to type in the terminal something like:

```
ssh -Y abilandz@lxplus.cern.ch
```

But do you really want to type that again and again each time you want to connect to *lxplus* at CERN? You can save a lot of typing, by introducing the alias for it, e.g.:

```
alias lx='ssh -Y abilandz@lxplus.cern.ch'
```

Here basically you have defined the abbreviation (or alias, or shortcut) for lengthy command input, and you have named it simply **lx**. Now it suffices only to type in the terminal

```
lx
```

and you will be connecting to *lxplus* at CERN with much less effort and time investment.

Another typical use case of aliases is to prevent command name typos. For instance, if you realize that too frequently instead of **ls** you have a typo **sl** when writing in the terminal, which does nothing except producing an error message, you can simply define the new alias for it:

```
alias sl=ls
```

With this definition, **sl** is literally a synonym for **ls** command.

If you have forgotten all aliases you have introduced in the current terminal session, just type in the terminal

```
alias
```

and all alias definitions will show up. If you want to see what is the definition of the concrete alias you have introduced, use

```
alias <alias-name>
```

The alias definition can be removed with **Bash** built-in command **unalias**, e.g.:

```
unalias <alias-name>
```

If you want to clean up all aliases in the current terminal, use:

```
unalias -a
```

where also for this command the option **-a** stands for 'all'.

Aliases are definitely a nice feature, but do not overuse them, because:

- By default, aliases are available only in the terminal in which you have defined them. But this can be easily circumvented by modifying the special configuration files `.bashrc` and/or `.bash_aliases` --- to be clarified later in this section;
- When you move to another computer your personal aliases are clearly not available there by default;
- Aliases can overwrite the name of the existing **Linux** or **Bash** command --- aliases will have the higher precedence in execution;
- Aliases cannot accept options or arguments, like regular commands (or **Bash** functions, as we will see later).

Aliases are literally shortcuts for lengthy commands or any other lengthy terminal input. They are defined for convenience only to save typing. Whatever you have defined an alias to stand for, **Bash** with simply inline or replace the alias name in the terminal with its definition, and then execute --- nothing more nor less than that!

## B) Variables

Just as any other programming language, **Bash** also supports a notion of *variable*. How to define variable in **Bash**? Let's say that we want to use the variable named `Var` and initialize it with the value 44? Simply type in the terminal:

```
Var=44
```

While this appears to be the most trivial thing you could do, even at this simple level we can encounter some problems, which can very nicely illustrate some of the general design philosophy used in **Bash** and **Linux**. Most importantly, since an *empty character* is the default input field separator, it would be completely wrong to type any of the following:

```
Var =44 # WRONG!!
Var= 44 # WRONG!!
Var = 44 # WRONG!!
```

In each case, you get an error message, e.g. `Var: command not found`, since **Bash** was trying to interpret the first token in the input, `Var` in this case, as a command name. Since command named `Var` was not found, **Bash** writes the error message in the terminal. Therefore, when introducing and initializing a new variable in **Bash**, make sure there are no empty characters round the *assignment operator* **=** .

As a side remark, from the above three lines, you can also see how to make a comment in **Bash** --- simply use the special character **#** (hash symbol) to start your comment. Once you use it on the particular line, any text after it is being ignored by **Bash**. You can not terminate the comment within a given line in which you have used **#** to start the comment. Therefore, you can terminate the commented text only by starting to write in the new line.

**Example:** Writing a comment in **Bash** .

```
echo "Hi there" # this is some comment which Bash ignores
Hi there
```

Once defined, how to use (or reference) the content stored in a variable? In order to reference the content of a variable, we use the special symbol **$** to achieve that, e.g.

```
Var=44
echo $Var
```

Also this syntax will do the job:

```
Var=44
echo ${Var}
```

In both cases the printout in the terminal is the same, namely:

```
44
```

So what is the difference between the two syntaxes above? The latter is less error-prone (as it clearly delineates with curly braces the variable name from the rest of the code!) and more powerful, as it enables a lot of built-in functionalities for the string manipulations programmatically within **Bash**.

As an example, this will produce the desired result only in the latter case:

```
Var=44
echo "test${Var}test"
```

The output is:

```
test44test
```

If you would have used the first, shorter syntax, then `Vartest` would be interpreted as a variable name, and since such variable was not defined, it would evaluate to the null string.

```
Var=44
echo "test$Vartest"
```

The output is:

```
test
```

Few final additional remarks on variables in **Bash**:

- They are untyped (i.e. you do not need to specify at declaration whether variables are integers, strings, etc.). By default, all **Bash** variables are strings, but if they contain only digits and if you pass them to some operator which takes as argument(s) only integers, then **Bash** will interpret the variable as an integer;
- By convention, for built-in **Bash** variable names we use only capital characters, while for command names we use all low-case characters. For user's variables, use some intermediate case, like `Var` or `someVariable`, to ease the code readability and to avoid potential conflict with the existing built-in **Bash** variable. Variable name starts with a letter or underscore, and may contain any number of following letters, digits and underscores. Variable name can not start with a digit;

- The lifetime of a variable is by default limited to the terminal session in which you have defined it. But you can make its existence persistent in any new terminal you open (i.e. in your *environment*) by adding its definition to the very special `.bashrc` file (more on this at the end of this section!);
- It is possible to store in the variable the output of some command, and then manipulate this output programmatically (more on this later!);
- It is possible to store in the variable the content of an external file (more on this later!);
- There are some built-in variables always set to some values, e.g. **HOME**, **SHELL**, **PATH**, etc. These special variables are the essential part of your **shell** environment, and if they are not set correctly, everything in your current terminal session can start falling apart (more on this later!).

Now that we have covered the very basics of commands and variables, let's see how we can develop the first **Bash** scripts. In order to achieve that, the very first step is to learn how to edit the file in the terminal.

## 3. Editing a file in the terminal

We have already seen how with **touch** command we can make an empty file. Now we will see how we can write a file or edit an already existing file in the terminal. The simplest way to write a new file, solely in the terminal (i.e. without using any graphics-based editor like **gedit**, **emacs**, **vim**, etc.), is to use the command **cat**, in the following construct:

```
cat > someFile.txt
 This text is the content
  of my
first
   file.
CTRL+d
```

In the above construct, command **cat** takes the input directly from the standard input (keyboard by default), and redirects everything via operator `>` into the physical file named `someFile.txt` in your current working directory. You terminate the input, i.e. you mark the end of the file, by first moving to the new line with pressing 'Enter', and then finally by pressing `CTRL+d` on an empty line (if you press `CTRL+d` when the line is not empty, the input to the file is not terminated!). Now you can see the content of your newly created file with:

```
cat someFile.txt
```

Note that **cat** preserves all empty characters, line breaks, etc. In the case you want to append something to the already existing file, we can use a slightly modified construct:

```
cat >> someFile.txt
test 1
test 2
CTRL+d
```

The operator `>>` appends the text at the end of an already existing file. If we would have used `>` to redirect the new content to the already existing file, that file would be overwritten with this new content --- use `>` in such a context with great care! This, however, also implies that the above **cat** construct is rather limited, as it can be used either to write a new file from scratch or to

append new content at the very end of an already existing file. But what if we want to edit the already existing content in the file?

For that sake, we need to use some simple editor which can be run in the terminal (i.e. without graphics). One such, wide-spread, open-source, editor is **nano**, which includes only the bare minimum of functionality needed to edit documents, making it very simple to use. In addition, syntax coloring is available for most of the programming languages. Now as an exercise, let us edit the content of already existing non-empty file `someFile.txt` from previous **cat** example.

```
nano someFile.txt
```

Now you are in the **nano** wonderland, not any longer in the **Bash** shell. This means that the commands you type now and all keyboard strokes are interpreted differently. After you have edited some existing text or wrote something new, simply in **nano** press `CTRL+o` (to write out into the physical file `someFile.txt` what you have edited so far in the editor --- this is the same thing as saving, just jargon is different...). When you are done with editing, press `CTRL+x` to exit **nano** (and type 'y' followed by 'Enter' if you want to save the changes in the same file you started with), and get back to the terminal. Of course, usage of **nano** is not mandatory to edit files, and for large files it is very inconvenient, but there are two nice things about **nano** which shouldn't be underestimated --- it is always available on basically all **Linux** distributions, and it can be run in the terminal (this becomes very relevant when connecting and working remotely on some computer, where access to graphics by default is not enabled!). But for the lengthy file editing, use some graphics-based editor: **gedit** is very easy to use without any prior experience, while **emacs** or **vim** are difficult for beginners, however they offer much more features.

## 4. Your first Bash script

Now that we know a few basic commands and how to write and edit files, we can start writing our first **Bash** scripts. The script is a code snippet for interpreted or scripting language, that is typically executed line-by-line. At the very least, this saves the effort of retyping that particular sequence of commands each time it is needed. Typically, scripts are used to automate the execution of tasks that could alternatively be executed one-by-one by a human operator. A scripting language is a programming language that supports scripts, so clearly **Bash** fits in this category.

Let us now write your first **Bash** script! For instance, you can type in the terminal:

```
nano first.sh
```

Recall that now you are not any longer in the terminal, but in the very simple textual editor called **nano**. Whatever you are typing now, it will be saved in the file `first.sh` --- the file which will hold your first **Bash** script (by convention **Bash** scripts always have an extension **.sh**). Your first **Bash** script could look as follows:

```
#!/bin/bash

echo "Welcome to Bash lecture"
# This is a comment...
echo "Today is:"
date

return 0
```

Now let us have a closer look at the content of your first **Bash** script:

- The first line is mandatory, namely: `#!/bin/bash`
- The first two characters in the first line are mandatory, namely: `#!` (the combination of these two characters is called shebang or hashbang)

What is happening here is the following: `#!` in the first line indicates to the operating system, that whatever follows next on the first line, must be interpreted as a path to the executable (e.g. executable is `/bin/bash` if you want to run **Bash**), which then must be used to interpret the code in all the remaining lines in the script. In this way, you can put up together any script, not necessarily only the one for **Bash** --- you just need to change `/bin/bash` in the first line, and point out to some other executable.

From the above example, you can see that whatever we have previously executed directly in the terminal (e.g. **echo** or **date** commands), we can also write in the script, and then execute all commands in one go, by executing the script. That being said, at the very basic level, scripting saves you the time needed to retype again and again any regular sequence of commands, after you open a new terminal --- for instance, the file `first.sh` you just made, is available in any new terminal you open!

How to execute the **Bash** script? It's simple, just pass the file name as an argument to the command **source** (i.e. in jargon, you need to *source* your script):

```
source first.sh
```

And the output could look like:

```
Welcome to Bash lecture
Today is:
Mon Apr 20 16:07:18 CEST 2020
```

Equivalently, you can use the shortcut notation `.` (dot):

```
. first.sh
```

In this context, **source** and `.` are synonyms (compare the output of `help source` and `help .`).

Finally, `return 0` sets the *exit status* of your script. In general, each command in **Bash** or **Linux** upon execution provides the so-called exit status. This is a fairly general concept, characteristic also for some other programming languages, and it enables us to programmatically check if the command has executed successfully or not. The exit status is classified as:

- 0 : success
- 1, 2, 3, ... , 255 : various error states

The exit status is stored in the special variable **$?**. For instance:

```
date
echo $? # prints 0 , i.e. success
```

and

```
date -q # option -q is NOT supported
echo $? # prints 1 , i.e. one possible exit status for error
```

Typically in your code, after you have executed the command, you check its exit status. Then, depending on the value of its exit status, your subsequent code can branch in multiple directions. Remember that each **Linux** command has an exit status stored in the special variable **$?** upon its execution, so it shall also your **Bash** script. As long as you are executing your script via **source** command, you can set the exit status with the keyword **return** (as in the last line in your above script `first.sh`).

If you forgot to specify the exit status of your script with the keyword **return** , the special variable **$?** is nevertheless automatically set, but now to the exit status of lastly executed command in your script, which can lead to unexpected results.

## 5. Special configuration files in Bash

In this section, we discuss a few important configurations files which have a special meaning to **Bash**. The configuration files, which a user can directly edit, acquire their special meaning only if they are placed in the user's home directory, otherwise **Bash** will not find and execute them. To stress this out, we have prepended `~` (or equivalently with `$HOME`) to the name of the user's configuration files below. The special character `~` (tilde) is the shortcut for the absolute path to your home directory. As an example, execute:

```
echo ~
```

The output might look like:

```
/home/abilandz
```

This is the absolute path to your home directory in the **Linux** file system. The slash `/` delineates directories in the **Linux** directory structure (on **Windows** backslash `\` is used instead in the same context). Each time you login, by default this is your starting working directory. This information is alternatively also stored in environment variable **HOME**, i.e. try:

```
echo $HOME
```

By convention, the name of all configuration files in the home directory begins with '.' (dot), which means that **ls** will not list them by default.

### User's configuration files

These are the personal configuration files in the user's home directory, which can be edited directly:

- `~/.bash_profile` : this configuration file is only executed by **Bash** each time you log in on the computer. There are two synonyms for this file: `~/.bash_login` and `~/.profile` . They

are executed at login only if `~/.bash_profile` is not present in your home directory. The files `~/.bash_profile` and `~/.bash_login` can be read only by **Bash**, while `~/.profile` is read also by some other shells, e.g. **sh** and **ksh**

- `~/.bashrc` : this configuration file is read with the highest priority when you open a new terminal, or when you start a new *subshell* (covered later in the lecture). This file will be read also at login only if you add a line `source ~/.bashrc` in `~/.bash_profile`
- `~/.bash_logout` : Executed whenever you log out from the computer. This is rarely used, but by editing this file, you can for instance automatically delete all temporary files at exit

## System-wide (default) configuration files

If by accident you have deleted your personal configuration files in your home directory, as a backup solution you can always rely on the two system-wide configuration files, which you cannot edit directly without having the administrator privileges:

- `/etc/profile` : the default, system-wide, configuration file which is read at login. It is read before `~/.bash_profile`. This means that you will always have some default settings enabled after you log in on the computer, whether or not `~/.bash_profile` with your personal settings exists or not
- `/etc/bash.bashrc` : the default, system-wide, configuration file which is read each time you open a new terminal or start a new subshell. It is read before `~/.bashrc`. This means that you will always have some default settings enabled after you open a new terminal or start a subshell, whether or not `~/.bashrc` with your personal settings exists or not

We now elaborate on the usage of these configurations files by considering a few concrete examples.

We have already seen how to define your own aliases and variables and we already stressed out one important point: Their lifetime is limited to the duration of the terminal sessions in which you have defined them. In any new terminal you start, their definitions are not known. But there is one important thing which happens behind the scene each time you start a new terminal, and before you can start typing anything: **Bash** reads automatically the configuration files end executes line-by-line whatever is being set in them.

In the most cases of interest, it suffices to know that you need to edit directly your personal file, e.g. `~/.bash_aliases`, and then in the **Bash** configuration files `~/.bash_profile` and `~/.bashrc`, which must be stored directly in your home directory, you insert the line:

```
source ~/.bash_aliases
```

For instance, let us edit in your home directory the file named `~/.bash_aliases` (any other name is perfectly fine as this is your personal file, not a special configuration file!). We start by executing in the terminal:

```
nano ~/.bash_aliases
```

And then in **nano** write the following two lines:

```
Var=44
alias sl=ls
```

Save the file and exit **nano** (press `CTRL+X` and choose 'y' followed by 'Enter'). You can check the content of file `~/.bash_aliases` via

```
cat ~/.bash_aliases
```

or, if the file got too lengthy and you need to scroll page-by-page, via

```
more ~/.bash_aliases
```

Since the content of `~/.bashrc` file is read and executed each time you start a new terminal, and before you can start typing anything in the terminal, your own personal definitions, for instance for aliases and variables, will be re-defined from scratch each time you start a new terminal, and you can re-use them again and again.

Now add the following line at the very end of `~/.bashrc` (if this line is already not inside that file --- by default it is already inside on most **Linux** distributions):

```
source ~/.bash_aliases
```

Each time you run a new terminal, the variable `Var` is set to 44, and you can use **sl** as the synonym for the **ls** command, i.e. you do not need to define them again in the new terminal sessions. In the case you need to add more aliases, simply edit again the file `~/.bash_aliases` .

Finally, we remark that it is much safer to edit directly `~/.bash_aliases` than to edit directly the file `~/.bashrc`, where also some other and more important settings can be defined as well. In the case you move to another computer, you can enable your aliases there simply by porting the file `~/.bash_aliases` , and adding on the new computer in `~/.bashrc` and `~/.bash_profile` the line `source ~/.bash_aliases`. On the other hand, typically it's very difficult to port the whole `~/.bashrc` from one computer to another, especially if they are running different **Linux** distributions.


# Lecture 3: Linux file system. Positional parameters. Your first Linux/Bash command. Command precedence

**Last update**: 20200514

## Table of Contents



## 1. Linux file system

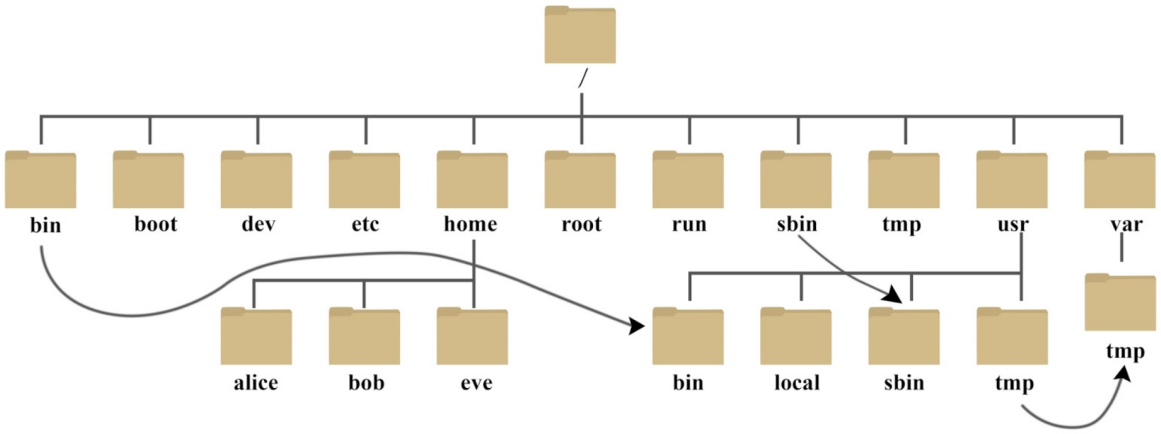
We have already seen how you can make your own files (e.g. with **touch**, **cat** or **nano**), and your own directories (with **mkdir**). The organization of files and directories in **Linux** is not arbitrary, and it follows some common, widely accepted, structure. The top directory is the so-called *root* directory and is denoted by `/` (slash). You can see its content by executing the following code snippet in the terminal:

```
cd /
ls
```

The output could look like:

```
bin  boot  dev  etc  home  lib  media  opt  proc  root  run  sbin  sys  tmp  usr
var
```

All files and directories on your computer are in one of these subdirectories. Schematically, the **Linux** file system structure can be represented with the following diagram:




The **Bash** built-in command **cd** ('change directory') is used to move from the current working directory to some other directory. It accepts only one argument, which is interpreted either as an *absolute path* to the new directory (if the argument starts with `/`), or as a *relative path* to the new directory (relative to your current working directory). If you get confused where you are at the moment in the **Linux** file system (i.e. where is your current working directory in the overall file system hierarchy), you can always get that information either from **Bash** built-in command **pwd** ('print working directory'):

```
pwd
```

or by referencing the content of environment variable **PWD**, which is always set to the absolute path of your current working directory:

```
echo $PWD
```

Both versions return the same answer in all cases of practical interest. However, and as a general rule of thumb, it's always much more efficient to get information directly from environment variable like **PWD**, than to retrieve and store in a variable the same information by executing the command, via the so-called *command substitution operator* (more on this later).

The most important directories in the **Linux** file system structure are:

- `/bin` : essential binaries needed for system functioning at any run level
- `/usr/bin` : binaries used by all locally logged in users
- `/usr/sbin` : binaries used only with superuser (root) privileges
- `/dev` : location of special or device files
- `/etc` : system-wide configuration files
- `/home` : holds user-specific accounts, personal area for each user
- `/proc` : kernel and process information
- `/tmp` : temporary files

We have already used **Linux** commands **date** and **touch**. But to which physical executables (binaries), stored somewhere in the file system, these two commands correspond to? You can figure that out by using the command **which**:

```
which date
/bin/date
```

```
which touch
/usr/bin/touch
```

It is completely equivalent to execute in the terminal the command name, e.g. **date**, or the full absolute path to the corresponding executable:

```
date
Mon Apr 27 16:12:06 CEST 2020
```

is the same as:

```
/bin/date
Mon Apr 27 16:12:06 CEST 2020
```

It would be very tedious and impractical if each time we would like to use some command, we would need to type in the terminal the absolute path to its executable sitting somewhere in the **Linux** file system, both in terms of typing and in terms of memorizing the exact locations. This is precisely where **Bash** (or any other **shell**) is extremely helpful --- **shell** finds the correct executable in the file system for us, after we have typed only the short command name in the terminal, and executes it. Clearly, something is happening here behind the scene: How does **shell** know which physical executable in the file system is linked with the short command name you have typed in the terminal? Hypothetically, we could also have another version of **date** command sitting somewhere else in the file system, e.g. in the directory `/usr/bin/date`. Then there is an ambiguity, since after we have typed in the terminal **date**, it is not clear whether we want `/bin/date` or `/usr/bin/date` to be executed.

This is resolved with a very important environment variable **PATH**. To see its current content, simply type:

```
echo $PATH
```

The output could look like this:

```
/home/abilandz/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

This output looks messy, but in fact it has a well-defined structure and is easy to interpret. In the above output, we can see absolute paths to a few directories, which are separated in this context with the field separator `:` (colon). The directories specified in the environment variable **PATH** are extremely important, because only inside them **Bash** will be searching for a corresponding executable, after you have typed the short command name in the terminal. Literally, the command **date** works because the directory **/bin**, where its corresponding executable `/bin/date` sits, was added to the content of **PATH** variable. The order of directories in **PATH** variable also matters: When **Bash** finds your executable in some directory specified in **PATH**, it will stop searching in the other directories specified in **PATH**. The priority of the search is from left to right. Therefore, if you have two executables in the file system for the same command name, e.g. `/bin/date` and `/usr/bin/date`, and if the content of **PATH** is as in the example above, after you have typed in the terminal **date**, **Bash** would try first to execute `/usr/bin/date` and not `/bin/date`, because `/usr/bin` is specified before `/bin` in the **PATH** variable. However, since there is no **date** executable in `/usr/bin`, **Bash** continues the search for it in `/bin`, finally finds it there, and then executes `/bin/date` .

By manipulating the ordering of directories in **PATH** variable, you can also have your own version of any **Linux** command --- just place the directory with your own executables at the beginning of **PATH** variable, and then those directories will be searched first by **Bash**. For instance, you can have your own executable for **date** in your local directory for binaries (e.g. in

`/home/abilandz/bin`). Then, you need to redefine **PATH** in such a way that it has your personal directory with higher priority, when compared to standard system-wide directories for command executables (like `/bin`, `/usr/bin`, etc.). This is achieved with the following standard code snippet:

```
PATH="/home/abilandz/bin:${PATH}"
```

With this syntax, your personal executables in `/home/abilandz/bin` are prepended to the current content of **PATH**, and will have therefore the highest priority in the **Bash** search.

For the lower priority of your executables, use an alternative standard code snippet:

```
PATH="${PATH}:/home/abilandz/bin"
```

In this example, you have appended your executables to what is already set in **PATH** --- this way you indicate that you want to use your own version of some standard, system-wide, **Linux** command only if its executable is not found by **Bash**. As always, if you want to make such definitions permanent in your terminal, add the above redefinitions of **PATH** into `~/.bashrc` file.

From the above explanation, it is clear that if you unset the **PATH** variable, all commands will stop working when you type them in the terminal, because **Bash** does not know where to search for the corresponding executables.

We finalize the explanation of **PATH** variable with the following concluding remarks:

- The search for the corresponding executable, after you have typed the short command name in the terminal, is optimized in the following ways:
  - Not all the files in the specified directories in **PATH** are considered during the search --- only the files which have *execute permission* (`x`) are taken into account by **Bash** (more on this in a moment!);
  - The recently used commands are *hashed* in the table --- this table is then looked up first by **Bash** after you type the command name in the terminal. To see the current content of the hash table, just type **Bash** built-in command **hash** in the terminal:

    ```
    hash
    ```

    The output could look like:

    ```
    hits    command
       4    /usr/bin/which
       5    /usr/bin/git
       1    /bin/date
       2    /bin/cat
       6    /bin/ls
    ```

    Cleary, the hash mechanism adds it a lot to the efficiency of commands' usage in **Linux**. Each time you login for the first time on computer the hash table is empty, then each terminal keeps its own hash table.
- The **PATH** search can be skipped by the user. In particular, when the command name contains the `/` (slash) character, not necessarily at the beginning of the name, **Bash** will not perform the search for the corresponding executable --- underlying assumption is that you have now yourself specified the path in the file system, either absolute or relative, to the

corresponding executable. In this case, **Bash** tries to execute that command name on the spot. This explains the standard syntax to run the command whose executable is in your current directory:

```
./some-command
```

In this context, the dot `.` is a shortcut syntax for the absolute path to the current working directory (the analogous shorthand notation for the parent directory is `..`). With the above syntax, even if the command **some-command** with a different implementation exists in some directory stored in **PATH**, it will never be searched for and executed, because there is `/` in the above command input.

Some frequently used **Linux** commands to work within the file system are:

- **cp** : copy file(s)

```
cp <file-1> <file-2> # copying and renaming a file
cp <file-1> <file-2> ... <directory> # copying two or more files in the same
directory
                                     # the names of original files are preserved
```

Files and directories in the arguments of **cp** can be specified either with the absolute or the relative paths. This is true in general for all commands which take files and directories as arguments.

- **cp -r** : copy directory and preserve its subdirectory structure

```
cp -r <directory-1> <directory-2> # this will copy the whole first directory
into
                                  # a new subdirectory of the second directory
```

- **rm** : delete file(s)

```
rm <file-1> <file-2> ... # delete the specifed files
```

Use **rm** with great care, because after you deleted the file, there is no way back!

- **rm -rf** : delete one or more directories

```
rm -rf <dir-1> <dir-2> ... # delete the specified directories
```

Flag **-r** ('recursive') is needed to indicate that you want to delete all subdirectories recursively, **-f** ('force') is needed to avoid the prompt message which would ask you for the deleting confirmation of each file separately. Use **rm -rf** with the greatest possible care, because after you have deleted the directory, there is no way to get back any file that was in that directory!

- **mv** : move or rename file(s)

```
mv <file-1> <file-2> # moving, if two files are not in the same directory
                     # renaming, if two files are in the same directory
```

The command **mv** uses the same syntax for directories (no additional flags are needed).

- **du -sh** : ('disk usage') : summary (flag **-s**) for the size of directory in the human-readable (flag **-h**) format

```
du -sh ${HOME} # prints how much disk space your home directory is taking
967M
du -h --max-depth=1 ${HOME} # the size of directory, and differentially of its
subdirectories
du -h --max-depth=2 ${HOME} # the size of directory, differentially of its
subdirectories and all sub-subdirectories
```

- **df -h** : ('disk free') : get the used disk space of all disks

```
df -h # get the status of all disks on your computer
file system      Size  Used Avail Use% Mounted on
/dev/sda1        1.8T  1.6T  132G  93% /
```

- **stat** : display the detailed metadata of file or directory

```
stat Lecture_2.md # just specify the abs. or rel. path to file as an argument
  File: Lecture_2.md
  Size: 97805           Blocks: 384        IO Block: 4096   regular file
Device: 2h/2d   Inode: 12947848928707821  Links: 1
Access: (0666/-rw-rw-rw-)  Uid: ( 1000/abilandz)   Gid: ( 1000/abilandz)
Access: 2020-04-15 21:05:26.002857000 +0200
Modify: 2020-04-28 11:44:53.454187100 +0200
Change: 2020-04-28 11:45:14.515681300 +0200
 Birth: -
```

Later we will learn how to parse through and extract programmatically from any command output (or from any physical file) only the information we need. For the time being, if you want to get only the size of the file in bytes, use:

```
stat -c %s Lecture_2.md
97805
```

For the size of a directory, use instead **du -sh** as explained above. As you can see from the output of **stat**, the example file `Lecture_2.md` is characterized by three timestamps: **Access**, **Modify** and **Change**. These three timestamps are an important part of file metadata, which we cover next.

## A) File metadata

File metadata is any file-related information besides its content. From the user's perspective, the most important file metadata are *timestamps*, *ownership* and *permissions*.

The meaning of three timestamps is as follows:

- **Access (a)** : last time a file was accessed (opened) and read without any modification
- **Modify (m)** : last time a file was modified (i.e. its content has been edited)
- **Change (c)** : last time a file's metadata was changed (e.g. permissions)

These three timestamps are not an overkill, in fact, they enable a lot of very powerful features when searching for specific files or directories in the file system. For instance, by using them, it is possible to list names of all files modified within the last day, to delete all files which were not accessed for more than 1 year, etc. (more on this later).

Next, each file or directory in **Linux** has three distinct levels of ownership:

- **User (u)** : the person who created the file
- **Group (g)** : the wider group to which the person who created the file belongs to
- **Other (o)** : anybody else

File ownership becomes extremely handy in combination with file permissions, when it's very simple to set common access rights for any group of other users.

Finally, each file in **Linux** has three distinct levels of permissions (or access rights):

- **Read (r)** : file can be read
- **Write (w)** : file can be written to (i.e. edited)
- **Execute (x)** : file is executable (i.e. binary, program)

For instance, when you execute

```
ls -al <some-file>
```

you can get the following example output:

```
-rw-rw-rw- 1 abilandz alice 97805 Apr 28 12:23 <some-file>
```

It is very important to understand all entries in this output, and how to modify or set some of them. Reading from left to right:

- **Column #1:**
  - the very first character is the file type : `-` is an ordinary file, `d` is a directory, `l` is soft-link, etc.
  - characters 2, 3 and 4 are fields for `r`, `w` or `x` permissions for the user (i.e. for you)
  - characters 5, 6 and 7 are fields for `r`, `w` or `x` permissions for the group (i.e. wider group of people where your account belongs to)
  - characters 8, 9 and 10 are fields for `r`, `w` or `x` permissions for anybody else
- **Column #2:** Number of files (always 1 for files and 2 or more for directories)
- **Column #3:** The user who owns the file ('abilandz' in this case)
- **Column #4:** The group of users to which the file belongs ('alice' experiment at CERN in this case)
- **Column #5:** The size of the file in bytes (for directories, it has another meaning, it is NOT the size of the directory!)

The meaning of the remaining columns is trivial.

File permissions are changed with the **Linux** command **chmod** ('change mode'). This is best illustrated with a few concrete examples:

```
chmod o+r someFile.txt
```

After the above command was executed, others (`o`) can (`+`) read (`r`) your file `someFile.txt`.

```
chmod go-w someFile.txt
```

In the above example, group members to which your account belongs to (`g`) and all others (`o`) can NOT (`-`) modify or write to (`w`) to your file `someFile.txt`. Therefore, after this simple command execution, only you can edit this file!

```
chmod u+x someFile.txt
```

With the above syntax, the file `someFile.txt` is declared to be an executable and only you as a user (`u`) can (`+`) execute it (`x`). Remember that only the files which are executables are taken into account by **Bash** when searching through the content of directories in **PATH** variable. Therefore, when making your own **Linux** command, two formal aspects must be always met:

1. the directory containing your executable must be included in **PATH**;
2. your executable must have `x` permission.

Next example:

```
chmod ugo+rwx someFile.txt
```

Now everybody (you as a user (`u`), group members (`g`) and others (`o`)), can read (`r`), modify or write to (`w`), or execute your file (`x`). For directories, you can change permissions in one go for all files in all subdirectories, by specifying the flag `-R` ('recursive'), i.e. by using schematically:

```
chmod -R <some-options-to-change-permissions> <some-directory>
```

Finally, we clarify that each permission setting can be represented alternatively by a numerical value. The rule is established with the following simple table:

| **permission** | **r** | **w** | **x** | **-** |
|---|---|---|---|---|
| **value** | 4 | 2 | 1 | 0 |

When these values are added together, the sum is used to set specific permissions.

For example, if you want to set only 'read' and 'write' permissions, you need to use a value 6, because from the above table, it follows immediately: 4 ('read') + 2 ('write') = 6. If you want to remove all 'read', 'write' and 'execute' permissions, you need to specify 0.

For convenience, all possibilities are documented in the table:

| value | permission | standard syntax |
|---|---|---|
| 7 | read, write and execute | rwx |
| 6 | read and write | rw- |
| 5 | read and execute | r-x |
| 4 | read only | r-- |
| 3 | write and execute | -wx |
| 2 | write only | -w- |
| 1 | execute only | --x |
| 0 | none | --- |

**Example:** Make a new file with default permissions, then remove all permissions, and set the permission pattern to `-rwx--xr--` , by using both syntaxes described above. With the first syntax, we would have:

```
touch file.log # make a new file
# the default permission pattern is: -rw-rw-rw-
chmod ugo-rwx file.log # strip off all permissions
# pattern is now: ----------
chmod u+rwx,g+x,o+r file.log # set new permissions
# the final pattern is: -rwx--xr--
```

With the alternative syntax, we proceed as follows:

```
touch file.log # make a new file
# the default permission pattern is: -rw-rw-rw-
chmod 000 file.log # strip off all permissions
# pattern is now: ----------
chmod 714 file.log
# pattern is: -rwx--xr--
```

It practice, it is not needed to remove old permissions and only then to set the new ones --- it was done here that way only for the sake of this exercise, but the old permissions can be directly overwritten.

Before we start developing the new commands from scratch in **Linux**, we need to introduce one very important and fairly generic concept: *positional parameters* (or *script arguments*).

## 2. Positional parameters

In this section we discuss how some arguments to your script can be supplied at execution. This clearly will allow you much more freedom and power in the code development, because nothing needs to be hardcoded in the script body. The very same mechanism can be used also in the implementation of **Bash** functions, as we will see later. We introduce now the so-called *positional parameters* (or *script arguments*).

**Example:** We want to develop a script, let's say `favorite.sh` which takes two arguments: the first one is the name of the collider, the second the name of the experiment. This script then just print something like:

```
My favorite collider is <some-collider>
My favorite experiment at <some-collider> is <some-experiment>
```

The solution goes as follows. In **nano** edit the file named `favorite.sh` with the following content:

```
#!/bin/bash

echo "My favorite collider is ${1}"
echo "My favorite experiment at ${1} is ${2}"

return 0
```

If you now execute this script for instance as:

```
source favorite.sh LHC ALICE
```

the printout looks as follows:

```
My favorite collider is LHC
My favorite experiment at LHC is ALICE
```

So how does this work? It is very simple and straightforward, there is no black magic happening here! Whatever you have typed first after `source favorite.sh` , and before the next empty character is encountered in the command input, is declared as the 1st positional parameter (or 1st script argument). The value of 1st positional parameter is stored in the internal variable `${1}` ('LHC' in the above example). Whatever you have typed next, and before the next empty character is encountered, is declared as the 2nd positional parameter, and its value is stored in the internal variable `${2}` ('ALICE' in the above example). And so on --- in this way you can pass to your script as many arguments as you wish!

Once you fetch programmatically in the body of your script the supplied arguments via variables `${1}`, `${2}`, etc. , you can do all sorts of manipulations on them, which can completely modify the behavior of your script.

Few additional remarks on positional parameters:

- You can programmatically fetch their total number via the variable: `$#`
- You can programmatically fetch them all in one go via the variables: `$*` or `$@` . In most cases of interest, these two variables hold the same result. For the purists: `"$*"` is equal to `"$1 $2 $3 ..."`, while `"$@"` is equal to `"$1" "$2" "$3" ...` . This means that `"$*"` is a single string, while `"$@"` is not, and this will cause a different behavior when you loop over all entries in `"$*"` or `"$@"` . But if you drop the double quotes, there is no difference between the content of special variables `$*` and `$@`
- It is also possible to access directly the very last positional parameter, by using the *indirect reference* ('value of the value') operator `!` — the syntax for the last positional parameter is `${!#}` . As a side remark, indirect reference `!` is a 'sort of pointer' in **Bash**, and its general usage is illustrated with the following code snippet:

```
Alice=44
Bob=Alice
echo ${Bob} # prints Alice
echo ${!Bob} # prints 44
```

In combination with looping, you can programmatically parse over the all supplied arguments to your script (i.e. there is no need to hardwire in the script that you expect exactly a certain number of arguments, etc.).

**Example**: Proof of the principle. Below is the script `arguments.sh` , which uses the **for** loop in **Bash** (to be covered in detail later!), and just counts and prints all arguments supplied to it:

```
#!/bin/bash

echo "Total number of arguments is: $#"
echo "The second argument is: ${2}"
echo "The very last argument is: ${!#}"

for Arg in $*; do
 echo "${Arg}"
done

return 0
```

If you execute this script for instance as:

```
source arguments.sh a bbb cc
```

you will get as a printout:

```
Total number of arguments is: 3
The second argument is: bbb
The very last argument is: cc
a
bbb
cc
```

By using this functionality, you can instruct your own script to behave differently if certain options or arguments are passed to it. Since this is clearly a frequently used feature, the specialized built-in **Bash** command exists to ease the parsing and interpretation of positional parameters (see the documentation of advanced **getopts** ('get options') command).

## 3. Your first Linux/Bash command: Bash functions

As the very first respectable version of your own command in **Linux/Bash**, which can take and interpret arguments, provide exit status, has its own environment, etc., we can consider **Bash** functions.

Functions in **Bash** are very similar to scripts, however, the details of their implementations differ. In addition, functions are safer to use than scripts, since they have a well-defined notion of *local environment*. This means basically that if you have the variable with the same name in your current terminal session, and in the script or in the function you are executing, it's much easier to

prevent the clash of these variables if you use functions. In addition, usage of functions to great extent resembles the usage of **Linux** commands, and in this sense, your first function developed in **Bash** can be also treated as your first **Linux** command!

Example implementation of **Bash** function could look like:

```
#!/bin/bash

function Hello
{
 # This function prints the welcome message
 # Usage: Hello <some-name>

 echo "Hello today!"
 local Name="${1}"
 echo "Your name is: ${Name}"

 return 0

}
```

Save the above code snippet in the file `functions.sh`. Then, in order to call your function **Hello**, just source that file:

```
source functions.sh
```

From this point onward, the definitions of all functions in the file `functions.sh` are loaded in the computer's memory, and can be in the current terminal session used as any other **Linux** or **Bash** built-in command. To check this, try to execute:

```
Hello Alice
```

The output is:

```
Hello today!
Your name is: Alice
```

When compared to the script implementation, there are few differences:

- Usage of keyword **function** (an alternative syntax exists, `<some-name>()`, but it's really a matter of taste which one you prefer)
- Body of the function must be embedded within `{ ... }`
- For any variable needed only within the function, use the keyword **local**, to restrict its scope only within the body of the function. In this way, you will never encounter the clash between variables that were defined with the same name in the function, and in the terminal or in some other code from where you call the function. If a variable is defined in the function without the keyword **local**, call to that function can spoil severely the environment from which the call to the function was executed, which can have dire consequences... As a rule of thumb, each variable you need only in the function, declare as **local**

The rest is the same as for the scripts:

- Functions accept arguments in exactly the same way as scripts, via special `${1}`, `${2}`, ... variables

- You can call a function within another function, but only if it was defined first --- order of implementation matters in scripting languages!
- Do not forget to provide the return value at the end of the function, which sets its exit status. For most of the time functions are executed equivalently as commands, and then their exit status clearly matters
- Typically, you implement all your functions in some file, let's say `functions.sh`, and save it in your home directory (or anywhere else). Then, at the end of `${HOME}/.bash_profile` and `${HOME}/.bashrc` you insert the line:

```
source ${HOME}/functions.sh
```

If you have added the definitions of your personal functions in `${HOME}/.bashrc`, your functions from the file `functions.sh` will be automatically loaded in computer's memory and are ready for usage in each terminal session, just as **Linux** commands --- in this sense the first **Bash** function you have written can be regarded also as your first **Linux** command!

## 4. Command precedence

We have seen that your very first input in the terminal, before the empty character is encountered, will be interpreted by **Bash** as the command name, where the command name can stand for an alias, built-in **Bash** command (e.g. **echo**), **Linux** command (e.g. **date**), **Bash** functions (e.g. **Hello** from the previous example), etc. But what happens if we have for instance alias and **Linux** command named in the same way? For instance:

```
alias date='echo "Hi!"'
```

If after this definition we type in the terminal **date**, we get:

```
date
Hi!
```

What now? Have we just accidentally overwritten and lost permanently the command **date**? Not quite, what happened here is that the alias execution got precedence over the **Linux** command named in the same way. But both the alias **date** and the command **date** now exist simultaneously on your computer.

The command precedence rules in **Bash** are well defined and strictly enforced with the following ordering:

1. aliases
2. **Bash** keywords (**if**, **for**, etc.)
3. **Bash** functions
4. **Bash** built-in commands (**cd**, **type**, etc.)
5. scripts with execute permission and **Linux** commands (at this level, the precedence is determined based on the ordering in **PATH** variable, as we already discussed)

Given the above ordering of command precedence, some care is definitely needed when introducing new aliases or developing new functions in **Bash**, to avoid the name clashes with the existing **Linux** commands.

Additional profiling of command precedence can be achieved with **Bash** built-in commands **builtin**, **command**, and **enable** (check their 'help' pages in **Bash**). For instance, we can force that always the **Bash** built-in command **echo** is executed, even if the alias or function named **echo** exists, with the following syntax:

```
builtin echo <some-text>
```

If you have overwritten accidentally **Linux** command with some alias definition (like in the above example for **date**), use the command **unalias** to revert back:

```
unalias <some-name>
```

In the case you are not sure to which one of the five cases above the command you intend to use in the terminal corresponds to, use the **Bash** built-in command **type**:

```
type date
date is /bin/date
```

The above line tells that **date** is **Linux** command whose executable is `/bin/date`.

Two other examples in this context:

```
type echo
echo is a shell builtin
```

```
type ll
ll is aliased to `ls -alF'
```

For the **Bash** functions, the command **type** also prints the source code of that function. For instance, for the function **Hello** discussed previously you would get:

```
type Hello
Hello is a function
Hello ()
{
    echo "Hello!";
    local Name="${1}";
    echo "Your name is: ${Name}";
    return 0
}
```

This is quite handy, because if you have forgotten the details of the implementation of this particular function, you do not need to dig into the file `functions.sh` where a lot of your additional functions can be implemented in the meanwhile.

Another argument is that this way you can see immediately the implementation of some **Bash** functions which were not developed by you (therefore, you have no idea where in the file system is the file with their source code), but are nevertheless available in your terminal session:

```
type quote
quote is a function
quote ()
{
    local quoted=${1//\'/\'\\\'\'};
    printf "'%s'" "$quoted"
}
```

Finally, it can happen that accidentally you delete the file `functions.sh`. If this file was sourced before you deleted it accidentally, you can still retrieve the implementations of your functions from the computer's memory with **type**, and then just redirect the output to some file.


# Lecture 4: Loops and few other thingies

**Last update**: 20200602

## Table of Contents



## 1. Scripts vs. functions

Now that we have seen how to implement in **Bash** both scripts and functions, we can discuss briefly their similarities, differences and typical use cases.

First, let us start with the execution details of scripts. In general, we run any **Bash** script either by 'sourcing' or by 'executing' that script. The first case corresponds to the following syntax:

```
source someScript.sh # sourcing the script
```

When executed this way, all lines in the script are read and executed by **Bash** one-by-one, just as if they were typed separately in the terminal. The sourced script inherits the environment from the terminal (i.e. from the current shell), and can modify it globally. The exit status of script must be specified with the keyword **return**. Script does not run in a separate process (more on this later).

```
someScript # executing the script
```

This way, you are running your script as any other **Linux** or **Bash** command. As we already saw, this will work only if the directory where the file with the source code of script sits was added to the environment variable **PATH**, and if that file has also the execute (`x`) permission. The executed script does not inherit by default the environment from the terminal, and cannot modify it globally. Therefore, it is much safer to run scripts this way, if you want to keep your current shell environment clean. The exit status of the executed script is specified with the keyword **exit**. When executed this way, the script runs in a separate process (more on this later).

On the other hand, functions behave differently. After you source the file where a function is implemented, **Bash** stores that function in the computer's memory, and from that point onwards you can use that function as any other **Linux** or **Bash** command. For functions, there is no need to bother with using keyword **source**, setting the execute permission, modifying **PATH**, etc. That means that if you have added to your `~/.bashrc` the following line:

```
source ~/functions.sh
```

where in the example file `~/functions.sh` you have the implementation of your **Bash** functions, you can use effortlessly all your functions in any new terminal you open.

Functions are much more suitable for making long scripts modular. In terms of environment protection, functions are much cleaner to use than scripts, due to keyword **local**, which can be used only in the function body, and which limits the scope and lifetime of a variable defined in the function only to the function execution.

If a function **someFunction** and a script **someScript** with execute permission have exactly the same implementation, then executing in the terminal **someFunction** only by its name is more efficient than executing in the terminal a script **someScript** only by its name, because **Bash** function does not start a separate process.

Programmatically, you can fetch the function name in its body implementation via built-in variable **FUNCNAME** (typically by having **echo $FUNCNAME** at the beginning of function body). For scripts, the file name in which the script was implemented can be obtained programmatically from the built-in variable **BASH_SOURCE**. This becomes very important when inspecting only the printout of your code execution (e.g. for debugging purposes), when it's easy to trace back which function or script produced which part of the final result (in this context, the built-in variable **LINENO** can also be handy, because **echo $LINENO** prints literally the line number of the source code where this variable is referenced).

We summarize the above thorough comparison with the following final conclusion: Use **Bash** scripts only for the very simple cases and **Bash** functions for everything else.

## 2. Command chain: && and ||

Since every command in **Linux** and **Bash** has the exit status, it is possible programmatically to branch the code execution, depending on whether a command has executed successfully (exit status 0), or has failed during execution with some error status (exit status 1.. 255). For instance, we would like multiple commands to execute one after another, but only if all of them execute successfully. As soon as one command has failed, we would like immediately to abort the execution of all subsequent commands. In **Bash**, we can achieve that with the *command chain*.

The command chain is a sequence of commands separated either with `&&` or `||` operators. If two commands are chained by `&&`, the second command will be executed only if the first one executed successfully. For instance:

```
mkdir someDirectory && echo "New directory was made."
New directory was made.
```

You will see the printout from **echo** only if the directory was successfully made with the command **mkdir**. On the other hand, if **mkdir** has failed, the command chain has broken, and **echo** is not executed. For instance, we can intentionally mistype **mkdir** just to simulate the failure of the first command in the chain:

```
mkdirrr someDirectory && echo "New directory was made."
mkdirrr: command not found
```

In this case, **echo** is not executed because the failure of **mkdirrr** has broken the command chain `&&`.

On the other hand, if two commands are chained by `||` operator, the second command in the chain will be executed only if the first command has failed:

```
mkdirrr someDirectory || echo "Cannot make directory. Sorry."
mkdirrr: command not found
Cannot make directory. Sorry.
```

The frequent use case of the command chain is to combine both `&&` and `||` operators in the following way:

1. start a command chain by grouping multiple commands with the `&&` operator;
2. append at the end of command chain the very last command with the `||` operator.

Schematically:

```
<command1> && <command2> && <command3> ... || <lastCommand>
```

The main point behind this construct is the following: **lastCommand** is executed if and only if any of the commands **command1**, **command2**, ..., has failed. The command **lastCommand** is not executed only if all of the commands **command1**, **command2**, ..., have executed successfully. Typically, the last command in the above chain would be some error printout accompanied by the code termination, either with **exit** or **return**. Therefore, the **lastCommand** is a sort of safeguard for the execution of all previous commands in the chain.

**Example**: Consider the following command chain

```
echo "Hello" && pwd && date || echo "Failed"
```

Since all commands executed successfully, it creates the following output:

```
Hello
/home/abilandz/Lecture/PH8214/Lecture_4
Mi 15. Mai 07:53:25 CEST 2019
```

The very last command after `||` operator, **echo "Failed"**, is not executed. Now we introduce some error, e.g. we mistype something intentionally:

```
echo "Hello" && pwddd && date || echo "Failed"
```

Now the output is:

```
Hello
pwddd: command not found
Failed
```

The first command in the `&&` chain executed successfully, and the execution continued with the next command in the `&&` chain. However, the second command **pwddd** has failed, and therefore has broken the `&&` chain. From that point onwards, only the command after `||` will be executed, and all the remaining commands in `&&` chain are ignored (the command **date** in this case).

In practice, the most frequent use case of the command chain is illustrated schematically:

```
<someCommand> || return 1
<someOtherCommand> || return 2
...
```

This way, it is possible to add easily an additional layer of protection for the execution of any command in your **Bash** code. Moreover, since the exit status is stored in the special variable **$?**, it is also possible by inspecting the content of that variable upon termination, to fix programmatically the particular reason of the failure, without intervening manually in the code.

## 3. Test construct: [[ ... ]]

For simple testing in **Bash**, we can use either `[[ ... ]]` or `[ ... ]` constructs. The construct `[[ ... ]]` is more powerful than `[ ... ]` since it supports more operators, but it was added to **Bash** later than `[ ... ]`, meaning that it will not work with some older **Bash** versions. Test constructs also return the exit status --- if the test was successful the exit status is set to 0 also in this context. Which operators we can use within these two test constructs depends on the nature of the content of the variable(s) we are putting to the test. Roughly, we can divide the use case of the test construct `[[ ... ]]` in the following 3 categories, and we enlist the meaningful operators for each category:

- General case: `-z, -n, ==, != , =~`
- Integers: `-gt, -ge, -lt, -le, -eq`
- Files and directories: `-f, -d, -e, -s, -nt, -ot`

These 3 distinct categories of the usage of `[[ ... ]]` are best explained with a few concrete examples --- we start with the general case.

### General case

**Example 1**: How to check if some variable **Var** has been initialized?

```
[[ -n ${Var} ]] && echo Yes || echo No
```

Remember the correct syntax and the extreme importance of empty characters within the test construct `[[ ... ]]`, as this is a typical source of errors:

```
[[ -n ${Var} ]] # correct
[[-n ${Var} ]]  # wrong
[[ -n ${Var}]]  # wrong
[[ -n${Var} ]]  # wrong
```

The very frequent use case is to check at the very beginning of the body of a script or a function if the user has supplied some value for the mandatory argument:

```
[[ -n ${1} ]] || return 1
```

If the user didn't provide value for the first argument, the above code snippet will terminate the subsequent execution.

The operator `-n` accepts only one argument and checks whether it is set to the same value, the opposite is achieved with `-z` which exits with 0 if its argument is not set.

If you forgot to specify an operator within the test construct, it is defaulted to `-n`, i.e.

```
[[ ${Var} ]]
```

is equivalent to

```
[[ -n ${Var} ]]
```

**Example 2**: How to check if the content of variable **Var1** is equal to the content of variable **Var2**?

We can illustrate this example with the following code snippet:

```
Var1=a
Var2=ab
[[ ${Var1} == ${Var2} ]] && echo Yes || echo No
```

Note that `==` is the comparison operator, while `=` is the assignment operator. The comparison operator `==` expects two arguments, and it treats both LHS and RHS arguments as strings. Since by default any variable in **Bash** is a string, this operator is applicable to any variable content. In particular, you can also compare integers this way, but it's much safer to do integer comparison with the `-eq` operator, as explained below. The operator `!=` does the opposite to `==`, i.e. it exits with 0 if two strings are not the same.

**Example 3**: How to check if one string contains another string as a substring?

```
Var1=abcd
Var2=bc
[[ ${Var1} =~ ${Var2} ]] && echo "Var1 contains Var2"
```

This frequently used operator is supported only within `[[ ... ]]`, but not within `[ ... ]`.

The executive summary for the first category of operators is provided with the following table:

| Operator | Outcome (exit status) |
|---|---|
| [[ -z ${Var} ]] | true (0) if Var is zero (null) |
| [[ -n ${Var} ]] | true (0) if Var holds some value |
| [[ ${Var1} == ${Var2} ]] | true (0) if Var1 and Var2 are exactly the same |
| [[ ${Var1} != ${Var2} ]] | true (0) if Var1 and Var2 are not exactly the same |
| [[ ${Var1} =~ ${Var2} ]] | true (0) if Var1 contains Var2 as a substring |

## Integers

When it comes to the second group of operators, `-gt, -ge, -lt, -le, -eq`, they are specific in a sense that they can accept only integers as arguments.

**Example 4**: How to check if one integer is greater than some other integer?

```
Var=44
[[ ${Var} -gt 10 ]] && echo Yes || echo No
```

Quite frequently, if your script or function demands that a user must provide exactly the certain number of arguments, you can use the following standard code snippet at the beginning of your code:

```
[[ $# -eq 2 ]] || return 1
```

In the above example, if a user did not provide exactly two arguments, the code execution terminates.

Since the meaning of integer operators is rather obvious, we just provide the executive summary of their usage with the following table:

| Operator | Outcome (exit status) |
|---|---|
| [[ ${Var1} -gt ${Var2} ]] | true (0) if Var1 is greater than Var2 |
| [[ ${Var1} -ge ${Var2} ]] | true (0) if Var1 is greater than or equal to Var2 |
| [[ ${Var1} -lt ${Var2} ]] | true (0) if Var1 is smaller than Var2 |
| [[ ${Var1} -le ${Var2} ]] | true (0) if Var1 is smaller than or equal to Var2 |
| [[ ${Var1} -eq ${Var2} ]] | true (0) if Var1 is equal to Var2 |

We can of course check if the two integers are the same by using the more general string comparison operator `==` (all variables are strings in **Bash**), but whenever you are sure that variables must contain integer content, `-eq` is clearly preferred over `==`.

## Files and directories

The very last group of operators, `-f, -d, -e, -s, -nt, -ot`, expects their argument(s) to be either files or directories. The first four accept one argument, while the last two take two arguments. Their meaning is illustrated in the following examples.

**Example 5**: How to check if the file `${HOME}/test.txt` exists or not?

```
Var=${HOME}/test.txt
[[ -f ${Var} ]] && echo "${Var} exists." || echo "${Var} doesn't exist."
```

Analogously, we can check for the existence of a directory with operator `-d`, for instance:

```
Var=${HOME}/SomeDirectory
[[ -d ${Var} ]] && echo "${Var} exists." || echo "${Var} doesn't exist."
```

Frequently, we want to trigger some code execution only if the file is non-empty, we can check that with the operator `-s`, as in the following example:

```
Var=${HOME}/test.txt
[[ -s ${Var} ]] && echo "${Var} is not empty" || echo "${Var} is empty"
```

For instance, if your script or function is expected to extract some data from the file that user needs to supply as the very first argument, you can implement the following protection at the very beginning against the empty file:

```
[[ -s ${1} ]] || return 1
```

Finally, it is possible to compare directly some file attributes, for instance the modification time.

**Example 6**: How to check if the file `${HOME}/test1.txt` is newer (i.e. modified more recently) than the file `${HOME}/test2.txt`?

This can be answered with operator `-nt` ('newer than') which takes two arguments:

```
File1=${HOME}/test1.txt
File2=${HOME}/test2.txt
[[ ${File1} -nt ${File2} ]] && echo "${File1} is newer" || echo "${File2} is
newer"
```

The executive summary of the most important test operators in this last category is provided in the following table:

| Operator | Outcome (exit status) |
|---|---|
| [[ -f ${Var} ]] | true (0) if Var is the existing file |
| [[ -d ${Var} ]] | true (0) if Var is the existing directory |
| [[ -e ${Var} ]] | true (0) if Var is existing file or directory |
| [[ -s ${Var} ]] | true (0) if Var is a file, and that file is not empty |
| [[ ${Var1} -nt ${Var2} ]] | true (0) if a file Var1 is newer than a file Var2 |
| [[ ${Var1} -ot ${Var2} ]] | true (0) if a file Var1 is older than a file Var2 |

In this section we have summarized the most important options --- for the other available options, check the corresponding documentation of test constructs by executing in the terminal:

```
help test
```

In the end, we indicate that the test construct `[[ ... ]]` can be used to branch the code execution, depending on whether some command executed correctly, or it has failed. If it has failed, we can branch even further the code execution depending on the exit status of a particular error. This is achieved by storing and testing the content of special variable **$?**, schematically:

```
someCommand # variable $? gets updated with the exit status of this command
ExitStatus=$? # store permanently the exit status of previous command
[[ ${ExitStatus} -eq 0 ]] && some-code-if-command-worked
[[ ${ExitStatus} -eq 1 ]] && some-other-code-to-handle-this-particular-error-
state
[[ ${ExitStatus} -eq 2 ]] && some-other-code-to-handle-this-particular-error-
state
...
```

Later we will see that such a code branching can be optimized even further with `if-elif-else-fi` or `case-in-esac` command blocks.

## 4. Catching user input: read

We have seen already how variables can be initialized in a non-interactive way, by initializing them with some concrete values at declaration. We will see now how the user input from the keyboard can be on-the-fly stored directly in some variable. In essence, this feature enables **Bash** scripts and functions to be interactive, in a sense that during the code execution (i.e. at run time), with your input from the keyboard you can steer the code execution in one direction or another. This is achieved with a very powerful **Bash** built-in command **read**.

By default, the command **read** saves input from the keyboard into its built-in variable **REPLY**. Alternatively, you can specify yourself directly the name of the variable(s) which will store the input from the keyboard. This is best illustrated with examples.

**Example 1**: If we use **read** without arguments, the entire line of user input is stored in the built-in variable **REPLY**, as this code snippets demonstrate:

```
read
```

After you have executed **read** in the terminal, this command is waiting for your input from the keyboard. Just type some example input, e.g. `1 22 333`, and press 'Enter'. Now you can programmatically retrieve the input from the keyboard:

```
echo ${REPLY}
1 22 333
```

Instead of relying on built-in variable **REPLY**, another generic usage of **read** is to specify one or more arguments explicitly, in the following schematic way:

```
read Var1 Var2 ...
```

This version takes a line from the keyboard input and breaks it down into words delimited by input field separators. The default input field separator is an empty character, and the input is terminated by pressing the 'Enter'.

**Example 2**: The previous example re-visited, but now using **read** with arguments.

```
read Name Surname
```

After typing that in the terminal, **read** is waiting for your feedback. Type something back, e.g. `James Hetfield`, and press 'Enter'. Now type in the terminal:

```
echo "Your name is ${Name}."
Your name is James.
echo "Your surname is ${Surname}."
Your surname is Hetfield.
```

The user-supplied arguments to the **read** command, **Name** and **Surname**, have become variables **Name** and **Surname**, initialized with the user's input from the keyboard, `James` and `Hetfield`, respectively.

If there are more words in the user's input from the keyboard than the variables supplied as arguments to **read**, all excess words are stored to the last variable.

**Example 3**: The previous example re-re-visited, but now using **read** with fewer arguments that there are words in the user's input.

```
read Var1 Var2
```

Feed to **read** the following input from the keyboard `1 22 a bb`, and press 'Enter'. If you now execute in the terminal

```
echo "Var1 is ${Var1}"
echo "Var2 is ${Var2}"
```

for the output you get:

```
Var1 is 1
Var2 is 22 a bb
```

The branching of the code execution at run-time, depending on the user's input from the keyboard, can be achieved in the following simplified and schematic way:

```
read Answer
[[ ${Answer} == yes ]] && do-something-if-yes
[[ ${Answer} == no ]] && do-something-if-no
```

In combination with `if-elif-else-fi` and `case-in-esac` statements (to be covered later!) the **read** command offers to the user a lot of flexibility on how to handle and modify the code execution at run-time.

The default behavior of **read** can be modified with a bunch of options (execute in the terminal **help read** for the full list). Here we summarize only the most frequently used ones:

```
-p : specify prompt
-s : no printing of input coming from a terminal
-t : timeout
```

For instance:

```
read -p "Waiting for the answer: "
echo ${REPLY}
```

The specified message in the prompt of **read** can hint to the user what to type:

```
read -p "Please choose either 1, 2 or 3: "
echo ${REPLY}
```

For the more complicated menus, **Bash** offers built-in command **select** which is covered later in the lecture.

The flag `-s` ('silent') hides in the terminal the user input:

```
read -s Password
```

Now the user's input is not showed in the terminal as you typed it, but it was stored nevertheless in the variable **Password**. Within your subsequent code you can programmatically do some checks on the content of **Password**. This is a very simple-minded mechanism for how you can handle passwords, etc.

Finally, with the following example:

```
read -t 5
```

the user is given 5 seconds to provide some input from a keyboard. If the user within the specified time interval does not provide any input, the **read** command reaches the timeout and terminates. The code execution proceeds like nothing happened. Therefore, within the specified time interval we are given the chance to type something and to modify the default execution of the code. All the above flags can be combined, which can make the usage of **read** command quite handy, and your scripts both interactive and flexible during execution.

The command **read** can be used in some other contexts as well, e.g. to parse the file content line-by-line in combination with loops---this is covered at the end of today's lecture.

## 5. Arithmetic in Bash

We have already seen that, whatever is typed first in the terminal and before the next empty character is encountered, **Bash** will try to interpreted as command, function, etc. For this reason, we cannot do directly arithmetic in **Bash**. For instance:

```
1+1
1+1: command not found
```

is producing an error, because a command named **1+1** doesn't exist. Other trials produce slightly different error messages, but the reason for the failure is aways the same:

```
1+ 1
1+: command not found
1 + 1
1: command not found
```

Instead, we must use the special operator `(( ... ))` to do integer arithmetic in **Bash**. For instance:

```
echo $((1+1))
```

produces the desired printout

```
2
```

The operator `(( ... ))` can also swallow the variables:

```
Counter=1
((Counter+=10)) # doing some integer manipulation
echo ${Counter} # prints 11
```

Within `(( ... ))` we can use all standard operators to perform integer arithmetic: `+, -, /, *, %, ++, --, **, +=, -=, /=, *=` , with self-explanatory meanings.

**Example:** How to calculate powers of integers in **Bash**? We can raise an integer to some exponent in the following way:

```
Int=5
Exp=2
echo $((Int**Exp)) # prints 25
```

As you can see from the above example, it is not necessary within `(( ... ))` to reference the content of variable explicitly with **$**, the operator itself takes care of that. The following lengthier code is also correct

```
echo $(($Int**$Exp)) # prints 25
echo $((${Int}**${Exp})) # prints 25
```

but clearly it's not as clear and elegant as the very first version.

Operator `(( ... ))` can handle only integers, both in terms of input and output. An attempt to use floating point numbers leads to an error:

```
echo $((1+2.4))
bash: 1+2.4: syntax error: invalid arithmetic operator (error token is ".4")
```

Floating point arithmetic cannot be done directly in **Bash**, but this is not a severe limitation, because we can always invoke some **Linux** command to perform it, like **bc** ('basic calculator'), which is always available---more on this later!

When it comes to division which doesn't yield as the final result an integer, **Bash** doesn't report the error, instead it reports as the result the integer after the fractional part (remainder) is discarded:

```
echo $((7/3)) # prints 2
echo $((8/3)) # prints 2
echo $((9/3)) # prints 3
```

To get the remainder after the division of two integers, we can use the modulo operator (`%`):

```
echo $((7%3)) # prints 1
echo $((8%3)) # prints 2
echo $((9%3)) # prints 0
```

Besides supporting integer arithmetic operators within, `(( ... ))` we can also perform integer comparison by using the familiar `<`, `<=`, `==`, `!=`, `>=` and `>` operators. This is an alternative to integer comparison within the test construct `[[ ... ]]` which has its own operators for integer comparison. For instance, the following code snippet

```
(( ${var1} < ${var2} ))
```

is equivalent to

```
[[ ${var1} -lt ${var2} ]]
```

and so on.

The most frequent use case of `(( ... ))` operator is to increment the content of variable within loops, which we cover next.

## 6. Loops: for, while and until

Just like any other programming language **Bash** also supports loops. The most frequently used loops are **for** and **while** loops, and only they will be discussed in this section in detail. The third possibility, the loop **until**, differs only marginally from the **while** loop, and therefore it won't be addressed separately. In particular, the **while** loop runs the loop *while* the condition is `true`, where the **until** loop runs the loop *until* the condition is `true` (i.e. while the condition is `false`). Besides that, there is no much of a difference between these two versions, and it's a matter of taste which one is preferred to be used. On the other hand, there are a few non-trivial differences between **for** and **while** loops, both in terms of syntax and use cases.

The syntax of **for** and **while** loops is fairly straightforward, and can be grasped easily from a few concrete examples. We start first with the examples for the **for** loop.

**Example 1**: Looping over the specified list of elements.

```
for var in 1 2 3 4; do
 echo "$var"
done
```

The output is:

```
1
2
3
4
```

This version of **for** loop iterates over all elements of a list. These elements are specified between keyword **in** and delimiter `;`. If you omit `;` the list needs to be terminated with the new line. Therefore, a completely equivalent implementation is:

```
for Var in 1 2 3 4
do
 echo "$Var"
done
```

Elements of a list are separated with the empty characters, and elements can be pretty much anything, e.g. consider:

```
Test=abc
for Var in 1 ${Test} 4.44; do
 echo "$Var"
done
```

The output is:

```
1
abc
4.44
```

Later we will see that we can even loop directly over the output of some command (e.g. over all files in a certain directory which match some naming convention, etc.).

**Example 2**: Looping over all arguments supplied to a script or a function.

We have already seen that we can loop over all arguments supplied to a script or a function in the following way:

```
for Arg in $*; do
 echo "Arg is: ${Arg}"
done
```

Since this is a frequently used feature, there exists a shorthand version when you need to loop over the arguments. Consider the following script named `forLoop.sh`, in which we have dropped completely the list of elements in the first line of **for** loop:

```
#!/bin/bash

for Arg; do
 echo "Arg is: $Arg"
done

return 0
```

By executing

```
source forLoop.sh a bb ccc
```

you get the following printout:

```
Arg is: a
Arg is: bb
Arg is: ccc
```

Therefore, if the list of elements is not explicitly specified in the first line of **for** loop, the list has defaulted to all arguments supplied to that script or function.

There exists also the C-style version of **for** loop in **Bash**, which can handle explicitly the increment of a variable. The C-style version looks schematically as:

```
MaxValue=someValue
for ((Counter=0; Counter<$MaxValue; Counter++)); do
 ... some commands ...
done
```

When it comes to the **while** loop, it is used very frequently and conveniently in the combination with the test construct `[[ ... ]]`. The following code snippets illustrate its most typical use cases. For the C-style while loop, we would use the following example syntax:

```
Counter=1
while [[ $Counter -lt 10 ]]; do
 echo "Counter is equal to: $Counter"
 ((Counter++))
done
```

Another frequently used case is illustrated in the following example:

```
while [[ -f someFile ]]; do # check if the file exists
 ...  some work involving the file someFile ...
 sleep 1m # pause code execution for 1 minute
done
```

This loop will keep repeating as long as the file `someFile` is available. When the file is deleted `[[ -f someFile ]]` evaluates to `false`, and the loop terminates.

As a side remark, in the above example we have used the trivial, nevertheless sometimes very handy, **Linux** command **sleep**. This command does nothing, except that it delays the code execution for the time interval specified via the argument. The argument can be interpreted as the time interval either in seconds (s), minutes (m), hours (h) or days (d):

```
sleep 10m # pause the code execution for 10 minutes
sleep 2h  # pause the code execution for 2 hours
```

This command can be used in the simplest cases to avoid a conflict among concurrently running processes. Another use case is to determine the periodicity of infinite loops.

**Example 3:** Infinite loops with the specified periodicity.

The following loop will keep running forever, with the periodicity of once per hour:

```
while true; do
 ... some code that you need again and again ...
 sleep 1h
done
```

In the above code snippet, we have used the **Bash** built-in command **true**, which does nothing except it returns the success exit status 0 each time it is called. There is also **Bash** built-in command **false**, which does nothing except it returns the error exit status 1.

A more sophisticated way to set up the scheduled execution of your code can be achieved with the command **crontab** (check out its man page).

With the keywords **continue** and **break** you can either continue or bail out from **for**, **while** and **until** loops. Outside of these three loops these commands are meaningless, and will produce an error. Their usage is illustrated with the following code snippet:

```
Counter=0
Max=4
while true; do
 ((Counter++))
 [[ ${Counter} -lt ${Max} ]] && echo "Still running" && continue
 echo "Terminating" && break
done
```

Upon execution, it leads to the following printout:

```
Still running
Still running
Still running
Terminating
```

If you have nested the loops, you can from the inner loop continue or break directly the outer loop. The level of the outer loop that you want to continue or break, is specified with the following syntax:

```
break someInteger
```

or

```
continue someInteger
```

In the next section, we discuss how we can combine some of these different functionalities covered by now, and establish another frequently used feature, which is especially handy when we need to parse through the file content line-by-line.

## 7. Parsing the file content: while+read

Very frequently, we need within a script or a function to parse through the content of an external file, and to perform some programmatic action line by line. This can be achieved very conveniently by combining the **while** loop and the **read** command. We remark, however, that this is not the most efficient way to parse the file content, its usage is recommended only for the short files.

As a concrete example, let us have a look at the following script, named `parseFile.sh`. This script takes one argument and that argument must be a file:

```
#!/bin/bash

File=$1 && [[ -f $File ]] || return 1

while read Line; do
 echo "I am reading now: $Line"
 sleep 1s
done < $File

return 0
```

The content of the file is redirected to the loop with `<` operator at the end of the loop.

Then, edit some temporary file, named for instance `data.log`, with the following simple content:

```
10 20 30
100 200
abcd
```

Finally, execute the script with:

```
source parseFile.sh data.log
```

The printout in the terminal is:

```
I am reading now: 10 20 30
I am reading now: 100 200
I am reading now: abcd
```

As we can see, **while+read** construct automatically reads through all the lines in the file, and in each iteration the whole content of the current line is stored in the variable which we have passed as an argument to the **read** command (in the above example it is the variable named **Line** --- if we do not specify any variable, then the built-in variable **REPLY** of command **read** is used automatically). That means that in each iteration within the **while** loop we have at our disposal the content of a line from the external file in the variable, and then we can manipulate its content within the script programmatically.


# Lecture 5: Command substitution. Input/Output (I/O). Conditional statements

**Last update**: 20200528

## Table of Contents



## 1. Command substitution: $( ... )

We have already seen that a value can be stored in a variable by explicit assignment (using the operator `=`), or of the user supplies variables as command line arguments (positional parameters) to a script or a function. In practice, however, one frequently wants to store the output of some command directly into the variable, or even the content of an external file. This can be achieved with the so-called *command substitution operator* `$( ... )`. For instance, we have already seen that the file size in bytes can be printed with the following:

```
stat -c %s someFile.log
```

But how can we fetch the above printout programmatically, and do some manipulation with it later in our code? This is precisely the case when we need to use the command substitution operator:

```
FileSize=$(stat -c %s someFile.log)
```

Now the size of file `someFile.log` is stored directly in the variable **FileSize** and from this point onwards we can reference content of that variable in the same way as the content of any other variable:

```
echo ${FileSize}
```

The operator `$( ... )` can do much more than that. For instance, it can literally in-line the output of any command at the place where this operator was used.

**Example 1**: How to produce the following single-line output, with the current timestamp embedded:

```
Today is Mo 20. Mai 15:33:07 CEST 2019 . What a nice day...
```

This can be achieved with:

```
echo "Today is $(date) . What a nice day..."
```

The command substitution operator literally in-lined the output of **date** command at the place where it was used. This way, we can very elegantly achieve the desired more complex functionality by combining in the very same command input multiple commands, which otherwise we would need to execute one-by-one.

Command substitution operator `$( ... )` is a very neat construct, and it is used frequently. One classical use case is to avoid hardwiring any specific information in your code, since it can change from one computer to another.

**Example 2**: You are working in parallel on two computers, which do not have the same version of the command that you use in your code. You would like to use if possible all the latest functionalities of that command, but if that's not available, you would still like to run your code with the older version of that command. Can you make the code transparent to such a difference? You can do it schematically as follows:

```
Version=$(commandName -v) # flag '-v' typically prints the version
[[ $Version -lt someTreshold ]] && use-older-functionalities
[[ $Version -ge someTreshold ]] && use-newer-functionalities
```

This is just a schematic solution --- most likely the output of **commandName -v** will have some additional information that you need to filter out, but all that can be still done within the command substitution operator.

You can fearlessly nest the command substitution operators, like in the following example.

**Example 3**: How can you get programmatically only the name of the parent directory of the directory in which your script sits (knowing that the environment variable **PWD** holds the full absolute path of script's directory)?

To solve this problem, we need first to introduce two widely used **Linux** commands in this context: **basename** and **dirname**. The command **basename** is typically used in the following way: It takes as an argument the absolute path to some directory or file, and drops the part which corresponds to an absolute path. This is illustrated with the following code snippets:

```
DirectoryPath=/home/abilandz/Lecture/PH8124/Lecture_5
basename ${DirectoryPath}
Lecture_5 # only the directory name is printed
```

On the other hand, the command **dirname** does the opposite: It prints only the absolute path to the specified directory or file. If we reuse the above example:

```
DirectoryPath=/home/abilandz/Lecture/PH8124/Lecture_5
dirname ${DirectoryPath}
/home/abilandz/Lecture/PH8124 # only the abs. path is printed
```

The commands **basename** and **dirname** can be used in exactly the same way for files.

Therefore, the solution to our initial problem can be fairly elegant and concise, if we use these two commands in combination with command substitution operator:

```
DirectoryPath=/home/abilandz/Lecture/PH8124/Lecture_5
ParentDirectoryName=$(basename $(dirname $DirectoryPath))
echo $ParentDirectoryName
PH8124 # only the parent directory name is printed
```

We can use multiple commands within the same command substitution operator, they just need to be separated with delimiter `;`, as in the following example:

```
Var=$(date;pwd)
echo "$Var"
```

The printout is

```
Thu May 14 11:58:28 CEST 2020
/home/abilandz/Lecture
```

It is perfectly fine to inline the output of your function with this operator:

```
echo "Output of my function is: $(someFunction) . Very nice!"
```

or to store the printout of a function in the variable:

```
Var=$(someFunction)
```

Finally, the very neat use case of the command substitution operator is to store the content of some external file in the variable. The relevant syntax is:

```
FileContent=$(< someFile)
```

In the above example, `<` is just a shortcut for the command **cat**, which can be used completely equivalently in this context:

```
FileContent=$(cat someFile)
```

This great functionality circumvents the necessity of dealing with too many temporary files during the code execution, when we are interested to keep the file content only at a particular time. With the above definitions, the following two commands yield exactly the same answer initially:

```
cat someFile # reads the content of a physical file
echo "${FileContent}" # references the content of variable
```

However, if the content of the physical file `someFile` has changed or if it was deleted, that does not affect the value of variable **FileContent**. This is very handy when we need to initialize our script or function with the content of some external file: if we store that information in the variable, we have removed completely the dependency of our code on that external file.

The command substitution operator is frequently used in combination with the **for** loop, when we want to iterate over all elements in the output of some command. Also in this context distinct elements of the list are separated with one or more empty characters. This is best illustrated with the following example:

**Example 4**: How can we loop over all files in the current directory and print the size of each file?

One simple solution (works only if filenames do not contain empty characters!) is provided with the following code snippet:

```
for File in $(ls $PWD); do
 [[ -f $File ]] && Size=$(stat -c %s $File) || continue
 echo "The size of ${File} is: ${Size}"
done
```

Note that if you would have used the lengthy output of **ls** by specifying the flag **-l**, then the loop variable **File** would loop over all entries in the command output separated with one or more empty characters, therefore also over the permission field, user name, etc. This is illustrated in the following example:

```
for Var in $(date); do
 echo $Var
done
```

The output is:

```
Thu
May
14
13:13:50
CEST
2020
```

This is another example to illustrate the importance of empty character as being the default field separator in **Linux/Bash**.

In the end, we would like to remark that the backticks `` `...` `` do the same thing as command substitution operator `$( ... )`:

```
echo "Today is: $(date) . Thanks for the info."
echo "Today is: `date` . Thanks for the info."
```

**Bash** supports backticks in this context only for backward compatibility with some very old shells. There is, however, one important difference: Nesting of backticks `` `...` `` doesn't work properly, only the nesting of command substitution operator `$( ... )` is reliable. That being said, `$( ... )` shall be preferably used in **Bash** instead of backticks `` `...` ``.

## 2. Input/Output (I/O) and redirections

In the previous section we have seen how we can embed the output of one command into the input of another command with the command substitution operator `$( ... )`. Let us now make a further progress in this direction and clarify in more detail the input and output streams of **Linux** commands. By convention, each **Linux** command has three standard input/output (I/O) channels set. More concretely, each **Linux** command has a single way of:

- accepting input : **standard input (*stdin*)** = file descriptor 0
- producing output : **standard output (*stdout*)** = file descriptor 1
- producing error messages : **standard error (*stderr*)** = file descriptor 2

Each command that you execute has these three standard I/O channels set to some default values. By default, standard input is a keyboard (but it can be also a file redirection, touchscreen, etc.). On the other hand, standard output and standard error are by default set to screen. The most important things to remember is:

- *stdout* (file descriptor 1): This is the textual stream you see in the terminal if a command executed successfully;
- *stderr* (file descriptor 2): This is the textual stream you see in the terminal if a command failed (a.k.a. error message).

For instance, when the command **date** executes successfully, it produces the following:

```
date
Sun May 17 11:53:03 CEST 2020
```

The above printout is an example *stdout* stream of command **date**. On the other hand, when the command **date** fails, for instance when it is called with the flag which is not supported:

```
date -q
```

it will print the error message:

```
date: invalid option -- 'q'
```

The above printout is an example *stderr* stream of command **date**. This behavior is true for basically all **Linux** commands.

Since these two streams, *stdout* and *stderr*, are always set for a command, we will now see how to handle them programmatically. In practice, one can programmatically fetch the *stdout* of some command, parse through it, and depending on its content, issue some specific action. In a similar fashion, one can fetch programmatically *stderr* (i.e. error message) of some command, and depending on its content, issue some specific action to fix that particular problem. For that sake, we need to use their respective file descriptors. The following operators are available in **Bash** to handle *stdout* and *stderr* streams:

- `1>` : captures and redirects to file only the successful output of command (*stdout*)
- `2>` : captures and redirects to file only the error message if command failed (*stderr*)
- `&>` : captures and redirects to the same file both the successful output (*stdout*) and the error message (*stderr*)

For instance, if we want to redirect the *stdout* stream of **date** command into a file, we would use:

```
date 1> output.log
```

Whatever the command **date** was printing on the terminal, now is re-directed to the physical file named `output.log`. If that file doesn't exist, it will be automatically created at this point. The file's location in the file system can be specified also in this context both with an absolute and a relative path. If you now execute:

```
cat output.log
```

you get back the output of **date** command:

```
Sun May 17 11:53:03 CEST 2020
```

In this sense, by using `1>` redirection, the printout of some command during execution is stored permanently in the physical file on a local disk.

Analogously, we can also programmatically redirect the error message of command, just need to change the file descriptor:

```
date -q 2> error.log
```

It's perfectly feasible to combine both examples on the same line:

```
someCommand 1> output.log 2> error.log
```

We can also redirect both *stdout* and *stderr* in the same file with `&>` operator:

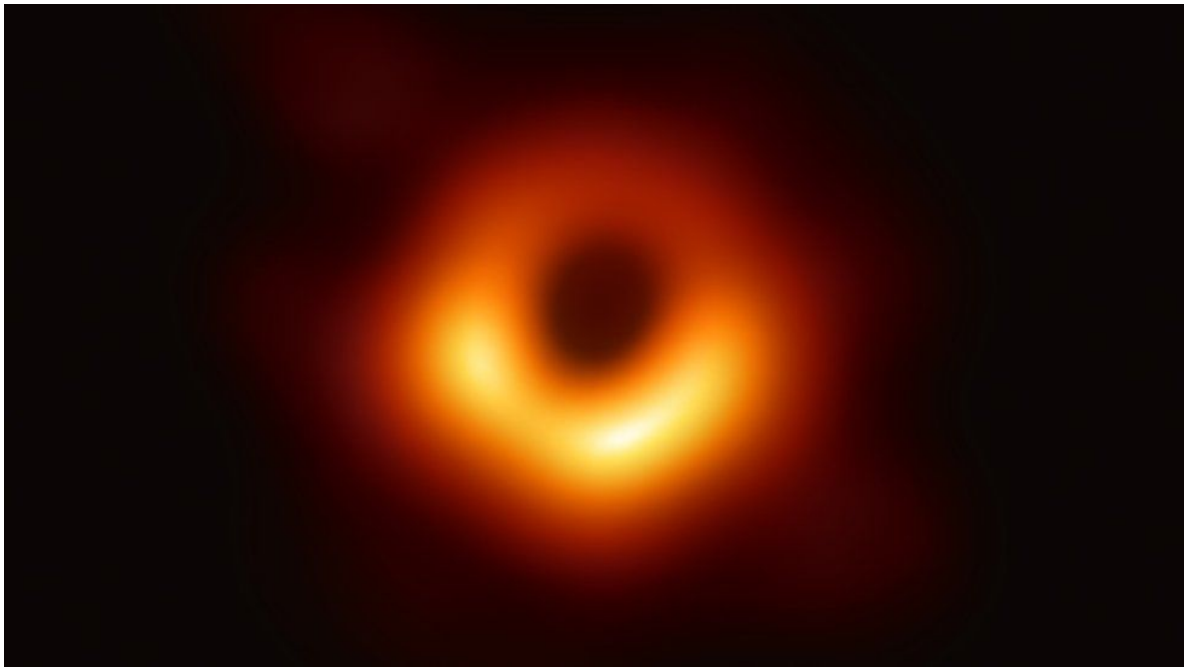
```
someCommand &> outputAndError.log
```

This way we can keep the whole printout which command has produced during execution permanently in some local files, separately for *stdout* and *stderr*, or combined . Then later at any point by inspecting those printouts in the files we can trace back the whole execution, which helps enormously the code development and debugging.

If we re-execute the above examples, the previous content of specified files will be overwritten with the new information. If instead, you want the new information to be appended to the existing content of those files, use instead the operators: `1>>`, `2>>` and `&>>`.

If the file descriptor number is not specified, it is defaulted to 1, i.e. `>` is exactly the same as `1>`, and `>>` is exactly the same as `1>>`.

Especially in the older **Bash** scripts you will see also `2>&1` redirection, but it has exactly the same meaning as `&>`, which was added only in more recent versions of **Bash**. The redirector `2>&1` means literally: Send *stderr* (file descriptor 2) to the same place where *stdout* (file descriptor 1) has been sent. When `2>&1` is used, the order matters --- first we need to indicate where `1>` is redirected, and only then `2>&1` makes sense to use. Because of this limitation, in practice it's much easier to use `&>` in such a context.



There is also a black hole in **Linux**, and it is called `/dev/null`. It happens frequently that you do not want to see the useless printout of some verbose command in the terminal, and you do not want to waste the disk space either to redirect it to some file. Quite frequently, some commands can print some warnings on the screen, after you have acknowledged them and concluded these warnings are harmless, you do not want to see those warnings again and again. This is precisely where the special file `/dev/null` becomes very handy, because whatever you redirect to it, it is lost forever.

**Example 1:** How to redirect only the successful output of a command to a file, and ignore completely the error messages (which are sometimes just the very annoying and harmless warnings)?

This problem is solved with the following code snippet:

```
someCommand 1>someFile 2>/dev/null
```

With the above construct, the file `someFile` will contain only the successful output of `someCommand`. On the other hand, all error messages are permanently lost, because they were redirected to the `/dev/null`.

**Example 2:** How to set programmatically separate *stdout* and *stderr* streams in your own code?

This question is answered with the following concrete example, in which a function expects some arguments from the user. If the user supplied arguments, the functions prints successfull *stdout* stream, and if the user failed to provide arguments, it prints the error message via *stderr* stream:

```
function myFunction
{
 [[ $# -eq 0 ]] && echo "Error: No arguments" >&2 && return 1
 echo "Arguments supplied" >&1 && return 0
}
```

With such an implementation, it is now possible programmatically to handle both *stdout* and *stderr* streams of this function:

```
myFunction a b c 1>output.log 2>error.log
```

In the above use case, the user has supplied some arguments ('a', 'b', 'c'), and therefore only the file `output.log` is filled, with the message defined in the function body for the *stdout* stream, namely 'Arguments supplied'.

```
myFunction 1>output.log 2>error.log
```

In the above example, no arguments were supplied. This is treated as an error within the function and it triggers its *stderr* stream, which is the message defined as 'Error: No arguments' in the function body.

Finally, let us also say a few words about the last file descriptor 0, *stdin* ('standard input'). In general, *stdin* comes from the keyboard, but we can also feed a command with the content of some file. Schematically, we would use:

```
someCommand < someFile
```

The operator `<` redirects the content of `someFile` into the argument of `someCommand`. In fact, `<` is nothing but the shortcut synonym for `0<` redirection. Because a lot of commands, e.g. **cat** or **more**, expect by default input from a file, the below three versions are all equivalent:

```
cat someFile
cat < someFile
cat 0< someFile
```

When you are checking the content of some file with **cat**, you are essentially redirecting its content into *stdin* (file descriptor 0) for the **cat** command.

## 3. Code blocks and brace expansion: { ... }

Clearly, the file descriptors are an extremely nice feature, but they would be even nicer if we would be able to use them to handle the output streams of multiple commands in one go, instead of redirecting the output stream of each command separately. This is possible in **Bash** by using the *code blocks*.

**Bash** code block is basically any sequence of commands within curly braces `{ ... }`.

Before presenting the concrete use cases, we first summarize the general facts about the code block:

1. `{ ... }` inherits the environment and can modify it globally
2. `{ ... }` has its own `1>` and `2>` streaming facilities
3. `{ ... }` does not launch a separate process. Therefore, the rest of a script or a function needs to wait for all commands in the code block to finish

Consider the following code snippet:

```
echo "before code block"
{
 echo "inside code block"
 date
 dateee # intentionally introduced error
} 1>output.log 2>error.log
```

In the very last line, we have redirected the *stdout* and *stderr* streams of all commands within the code block in one go. If we now check the content of files `output.log` and `error.log`, we find the following lines in the file `output.log`:

```
inside code block
Do 23. Mai 08:56:49 CEST 2019
```

and the following line in the file `error.log`:

```
dateee: command not found
```

On the other hand, on the screen the only printout is:

```
before code block
```

because we didn't redirect the first **echo** command anywhere.

Another piece of code in the same script or function can be embedded into another code block, and then redirected to some other files. This way we can easily profile the code with redirectors, and decide what goes on the screen and what is archived in files. Typically, code blocks `{ ... }` are used when it's not beneficial to break down some large monolithic script into functions.

When it comes to redirections, it is possible to treat loops analogously as code blocks. In particular, **for** and **while** loops have their own *stdout* and *stderr* streams, which can be redirected to the output files with `1>` and `2>` operators. In this way, you can disentangle what is happening in a particular loop, from what is happening in the rest of the code. Schematically, we would use for **for** loop:

```
for Var in someList; do
 ... some commands ...
done 1>output.log 2>error.log
```

We can simultaneously feed the **while** loop from an external file, and redirect its *stdout* and *stderr* in some other external files, schematically:

```
while read Line; do
 ... some commands ...
done <someFile.log 1>output.log 2>error.log
```

This way, for instance, we can parse and modify programmatically the example file `someFile.log` line-by-line, save the modified new content immediately in the file `output.log`, and all errors which might occur during the editing in a separate file `error.log`.

To check the influence of code block on the environment in your terminal, you can execute the following code snippet:

```
Var=44
echo "Before : $Var"
{
 echo "Inside : $Var"
 Var=55
}
echo "After  : $Var"
```

Upon execution, this code snippet produces:

```
Before : 44
Inside : 44
After  : 55
```

From this example we can easily see that the code block inherits all settings from the global environment, and that all modifications made inside the code block (e.g. some variables might get a new value) are propagated outside to the global environment, after the code block terminates. The different behavior can be obtained by enclosing the particular code within different type of braces, namely the round braces `( ... )`, to define the *subshell*---this will be covered later.

Very conveniently, the code block `{ ... }` can be combined with the command chain operators, as the next example illustrates.

**Example:** Is it possible to condense the following lines into a single line:

```
someCommand
ExitStatus=$?
[[ $ExitStatus -eq 0 ]] && command1 && command2 && ...
[[ $ExitStatus -ne 0 ]] && commandA && commandB && ...
```

By using the code blocks, this can be rewritten much more elegantly and efficiently as:

```
someCommand && { command1 && command2 && ... ; } || { commandA && commandB &&
... ; }
```

Note the mandatory trailing semicolon `;` within each code block in this context. This is important, because you need to indicate that `}` is not an argument to the last command within the code block---the last command input is terminated with semicolon `;`.

**Brace expansion**

We close this section with a side remark on curly braces. Besides being used to mark the code blocks, curly braces are also used in a completely different context to define programmatically the sequences, via the so-called *brace expansion*.

The syntax to generate sequences by using the brace expansion is demonstrated with the following concrete examples:

```
echo {1..10}
echo {1..10..2}
echo {10..1}
echo {a..f}
echo {-4..4}
```

The corresponding printouts are:

```
1 2 3 4 5 6 7 8 9 10
1 3 5 7 9
10 9 8 7 6 5 4 3 2 1
a b c d e f
-4 -3 -2 -1 0 1 2 3 4
```

Brace expansion is very frequently used in enumerating sequentially either files or directories.

**Example 1:** How to make 100 new directories named `Dir_0, Dir_1, ... Dir_99`?

The solution is very elegant by using the brace expansion mechanism:

```
mkdir Dir_{0..99}
```

**Example 2:** How to make 100 new files named `File_0.data, File_1.data, ... File_99.data`?

We can both prepend and append strings to the brace expansion, so also in this case there is a very elegant solution:

```
touch File_{0..99}.data
```

Brace expansion can be used also in combination with arbitrary string patterns.

**Example 3:** How to make three new files named `someLengthyFileName.log`, `someLengthyFileName.png` and `someLengthyFileName.pdf` in one go?

The solution is:

```
touch someLengthyFileName.{log,png,pdf}
```

which clearly saves a lot of typing.

Multiple brace expansions can be combined in the same command input. For instance, having already named directories or files sequentially, we can easily manipulate only a subset of them, by using the brace expansion.

**Example 4**: Imagine that in some directory you have the following files:

```
File_0.log File_1.log ... File_999.log
File_0.inf File_1.inf ... File_999.inf
File_0.dat File_1.dat ... File_999.dat
```

How to delete each 4th file within the interval 111 to 222, whose extension is `.log` or `.inf`, but not `.dat`? If you use the brace expansion, the solution is very simple and elegant:

```
ls File_{111..222..4}.{log,inf} # always ls before deleting!
rm File_{111..222..4}.{log,inf}
```

Without brace expansion the solution would take much more work. It is also possible to nest brace expansion, but this is rarely used in practice.

## 4. Conditional statements

We have already seen how to branch the code execution in **Bash** by using the command chain `&&` and `||`. For more complicated cases, however, a more elegant and flexible solution can be reached with *conditional statements*, which in **Bash** work very similar to other programming languages. For simpler cases, we can use **if-elif-else-fi** conditional statement, while the syntax of **case-in-esac** is better suitable for more complicated cases.

#### if-elif-else-fi

The typical use case of **if-elif-else-fi** conditional statement is to branch the code execution depending on the outcome of test construct `[[ ... ]]`. Schematically:

```
if [[ someExpression ]]; then
 some code when someExpression succeeds
elif [[ someOtherExpression ]]; then
 some code when someOtherExpression succeeds
 ... even more elif statements ...
else
  some code when all tests above failed
fi
```

You can have as many different **elif**'s branches as you wish, but the very last branch must start with the keyword **else**, and be closed with the keyword **fi**. The keyword **then** doesn't need to be placed on the same line with keywords **if** and **elif**, a completely equivalent syntax is:

```
if [[ someExpression ]]
then
 some code when someExpression succeeds
elif [[ someOtherExpression ]]
then
 some code when someOtherExpression succeeds
 ... even more elif statements ...
else
  some code when all tests above failed
fi
```

However, if the keyword **then** is placed on the same line with keywords **if** and **elif**, it has to be separated with semicolon `;`. The first syntax is more suitable for writing directly in the terminal.

Another typical use case of **if-elif-else-fi** conditional statement is to branch the code execution depending on whether a command or a function execution succeeded (exit status 0) or failed (exist status 1 to 255). Schematically:

```
if someCommand; then
  some code when someCommand succeeded
elif someOtherCommand; then
  some code when someOtherCommand succeeded
  ... even more elif statements ...
else
  some code when all commands above failed
fi
```

In practice, you frequently need to check only the exit status of a command, and do not need to see any output stream when executing that command. That can be achieved with:

```
if someCommand &>/dev/null; then
```

In the same way, you can use all other file descriptors, like `1>` and `2>`, as a part of **if** or **elif** statement.

It is possible to use the command chain within the same **if** or **elif** statement:

```
if someCommand && someFunction; then
```

In this example, the corresponding branch will be executed if the exit status of all chained commands is 0.

Finally, it is also possible to execute sequentially different commands within the same **if** or **elif** statement:

```
if command1; someFunction; command2; then
```

In this example, the corresponding branch will be executed only if the exit status of the very last command **command2** is 0, the exit status of previous commands play no role with this version.

**case-in-esac**

On the other hand, the syntax of **case-in-esac** conditional statement is more elaborate, but also more elegant and powerful. The generic syntax looks like:

```
case someValue in
 firstOption) some code when this option is met ;;
 secondOption) some code when this option is met ;;
 ... even more options ...
 *) some code when all specified options are not met ;;
esac
```

The thing to remember is that in **case-in-esac** conditional statement a specific branch of code execution is embedded within round brace `)` and double semicolon `;;` (yes, double semicolon, no empty character is allowed here!). This peculiar syntax, the unbalanced round brace `)` and the double semicolon `;;` are special to **case-in-esac** conditional statement, and therefore easy to remember.

The usage of **case-in-esac** conditional statement is best illustrated with a few concrete examples.

**Example:** How to implement the support for options in your script or function?

Schematically, for the simplest cases, that can be achieved with the following code snippet:

```
Flag=$1
case $Flag in
 -a)
    echo "The option -a has been specified!"
    echo "For the option -a we do the following..."
  ;;
 -b)
    echo "The option -b has been specified!"
    echo "For the option -b we do the following..."
  ;;
  *)
    echo "The specified flag is not supported"
    return 1 # bail out with error exit status
  ;;
esac
```

For more elaborate cases on how to parse command-line arguments in such context, see **Bash** built-in command **getopts**.

Multiple options can be grouped with `|` (OR) under the same statement, schematically:

```
case someValue in
 firstOption | secondOption | ... )
      ... some code when one option of this group is met ...
    ;;
 someOtherOption | yetAnotherOption | ... )
      ... some code when one option of this group is met ...
    ;;
    ... even more options ...
*) some code when all specified options are not met ;;
esac
```

The **case-in-esac** conditional statement recognizes the so-called POSIX brackets. The most important examples are:

- `[[:alpha:]]` Alphabetic characters [a-zA-Z]
- `[[:digit:]]` Digits [0-9]
- `[[:alnum:]]` Alphanumeric characters [a-zA-Z0-9]

Example use case:

```
Var=someValue
case $Var in
 [[:alpha:]])
   echo "Var is a single alphabetic character"
 ;;
 [[:digit:]])
   echo "Var is a digit"
 ;;
 *)
   echo "Var is something else"
 ;;
esac
```

As the final remark, when developing the code using conditional statements, sometimes we are not sure immediately what to implement in the particular branch. We cannot leave that branch empty or only insert a comment, because both will produce an error:

```
if [[ ${Var1} -gt ${Var2} ]]; then
 # I will implement this part later
fi
```

The error message is:

```
line 4: syntax error near unexpected token `fi'
line 4: `fi '
```

For this sake, we need to use the so-called 'do-nothing' command as a placeholder. The syntax of 'do-nothing' command is simple a colon `:`.

The correct solution to the above problem is:

```
if [[ ${Var1} -gt ${Var2} ]]; then
 : # I will implement this part later
fi
```

'Do nothing' command `:` does literally nothing, except that it always returns the exit status 0, i.e. it always succeeds in what it needs to do, which is not surprising given that fact that it does nothing:

```
:
echo $?
# prints 0
```

Quite remarkably, even such a simple command has some interesting and frequent use cases.

**Example:** How to empty the already existing file, keeping all file permissions intact?

```
: > someFile
```

Literally, we have redirected nothing into the existing file, therefore its content is now nothing. Note that we have kept all file permissions intact in a process. Therefore, this is in general not the same as deleting the existing file, and then creating a new empty file with the same name:

```
rm someFile
touch someFile
```

because now the permissions of a new file are set to default permissions, and we need to invest some additional work to set again our own permissions.

**Example:** Infinite loop in **Bash**.

The simplest implementation is:

```
while :; do
 ... some code ...
done
```

We can also use 'do-nothing' `:` command to write a multi-line comment in **Bash** in combination with the so-called *here-documents*---this will be covered later.


# Lecture 6: String manipulation. Arrays. Piping (`|`). sed, awk and grep

**Last update**: 20200618

## Table of Contents



## 1. String manipulation

**Bash** offers a lot of built-in functionalities to manipulate the content of variables programmatically. Since the content of an external file can be stored in a **Bash** variable, we can to a certain extent solely with built-in **Bash** features manipulate the content of external files as well. However, performance starts to matter typically for large files, when **Linux** core utilities **sed**, **awk** and/or **grep** are more suitable. For very large files, when performance becomes critical, one needs to use the high-level programming languages, like **perl**.

String operators in **Bash** can be used only in combination with curly brace syntax, `${var}`, when the content of a variable is referenced. String operators are used to manipulate the content of variables, typically in one of the following ways:

1. Remove, replace or modify a portion of variable's content that matches some patterns
2. Ensure that variable exists (i.e. that it is defined and has a non-zero value)
3. Set the default value for a variable

The generic syntax for manipulating the content of the variable is:

```
${Var/OldPattern/NewPattern}
```

or

```
${Var//OldPattern/NewPattern}
```

The first version will replace only the first occurrence of the pattern 'old-pattern' with 'new-pattern' within the string which is stored in the variable 'Var', while the second version will replace all occurrences. This is illustrated with the following code snippet:

```
Var=aaBBaa
echo ${Var/aa/CCC}
echo ${Var//aa/CCC}
```

which prints:

```
CCCBBaa
CCCBBCCC
```

It is perfectly fine to re-define the variable on the spot with the new content:

```
Var=${Var/aa/CCC}
```

The new and old patterns do not have to be hardwired, instead, they can be specified via variables:

```
Var=aaBBaa
Old=aa
New=CCC
Var=${Var/$Old/$New}
```

The curly-brace syntax interprets some characters in a special way. This is illustrated with the following examples.

**Example 1:** How to get programmatically the length of the string?

```
Var=1a3b56F8
echo ${#Var} # prints 8
```

**Example 2:** How to lower/upper cases of all characters in the string?

```
Var=aBcDeF
echo ${Var,,} # prints 'abcdef'
echo ${Var^^} # prints 'ABCDF'
```

It is also possible with the curly brace syntax to select substring from variable content, with the following generic syntax is:

```
${Var:offset:length}
```

The above construct returns substring, starting at 'offset', and up to 'length' characters. The first character in the content of variable 'Var' is at the offset 0. If 'length' is omitted, it goes all the way until the end of 'Var'. If 'offset' is less than 0, then it counts from the end of 'Var'. This is illustrated with the following examples:

```
Var=0123456789
echo ${Var:0:4} # prints '0123'
echo ${Var:5:2} # prints '56'
echo ${Var:5} # prints '56789'
echo ${Var:(-2)} # prints '89'
echo ${Var:(-3):2} # prints '78'
```

It is mandatory to embed negative offset within round braces `( ... )` in the above example, since otherwise **Bash** interprets negative integers after colon `:` in such a context in a very special way---this is clarified next.

By using string operators one can set the default value of a variable. Most frequently, one encounters the following two use cases:

1. `${Var:-defaultValue}` : if 'Var' exists and it is not null, return its value. Otherwise, return the hardwired 'defaultValue'. This is basically protection that variable always has some content. For instance:

   ```
   Var=44
   echo ${Var:-100} # prints 44
   ```

   However:

   ```
   unset Var
   echo ${Var:-100} # prints 100
   ```

   This syntax has a very important use case when a script or a function expects the user to supply an argument. Even if the user forgot to do it, we can nevertheless execute the code for some default and meaningful value of that argument. For instance:

   ```
   Var=${1:-defaultValue}
   ```

   This literally means that 'Var' is set to the first argument the user has supplied to a script or a function, but even if the user forgot to do it, the code can still execute by setting 'Var' to 'defaultValue'.

2. `${Var:?someMessage}` : if 'Var' exists and it is not null, return its value. Otherwise, prints 'Var', followed by hardwired text 'someMessage', and abort the current execution of a function (in the case this syntax is used in a script, it only prints the error message). For instance, in the body of your function you can add protection via:

   ```
   function myFunction
   {
    local Var=${1:?first argument is missing}
    ... some code ...
   }
   ```

   In case the user has forgotten to provide the first argument, your function will terminate automatically with the error message:

   ```
   myFunction
   bash: 1: first argument is missing
   ```

   If you do not specify the message, the default message will be produced. For instance:

   ```
   unset someVariable
   Var=${someVariable:?}
   ```

   will produce the following error message:

```
-bash: someVariable: parameter null or not set
```

In both of these examples we have used colon `:` within the curly braces, but this is optional. However, if we omit the colon `:` and use instead the syntax `${var-defaultValue}` and `${var?someMessage}`, the meaning is slightly different: the previous phrase 'exists and it is not null' translates now only into 'exists'. This difference concerns cases like this:

```
var= # Var exists but it is NULL
echo ${var:-44} # prints 44
echo ${var-44} # prints nothing
```

When replacing old patterns with the new ones, **Bash** can handle a few wildcard characters. The most important wildcards are:

1. `*` : zero or more characters
2. `?` : any single character
3. `[ ... ]` : character sets and ranges

Their usage is best illustrated with a few concrete examples:

```
var=1234a5678
echo ${var/a*/TEST} # prints '1234TEST'
```

Here the pattern with the wildcard, 'a*', matches any string starting with 'a' and followed by 0 or more other characters.

```
var=a1234a5678
echo ${var//a?/TEST} # prints 'TEST234TEST678'
```

The pattern with the wildcard 'a?' matches a string starting with the character 'a' and followed by exactly one other character (in the above example, it matched both 'a1' and 'a5', which were both replaced, due to `//` specification within curly braces, into a new pattern 'TEST').

```
var=abcde12345
echo ${var//[b24]/TEST} # prints 'aTESTcde1TEST3TEST5'
```

The pattern '[b24]' matches any single character specified within `[ ... ]` (in the above example, 'b', '2' and '4' were all replaced with 'TEST').

```
var=abcde12345
echo ${var//[b-e]/TEST} # prints 'aTESTTESTTESTTEST12345'
```

The pattern '[b-e]' matches all single characters in the specified range within `[ ... ]` (in the above example, 'b', 'c', 'd' and 'e', i.e. all characters in the range 'b-e' were all replaced with the new pattern 'TEST').

The real power of wildcards is manifested when they are combined:

```
var=a1b2c3d4e5
echo ${var//[b-d]?/TEST} # prints 'a1TESTTESTTESTe5'
```

The pattern '[b-d]?' matches all single characters in the specified range 'b-d' followed up by exactly one other character (in the above example, 'b2', 'c3' and 'd4' were all replaced with 'TEST').

```
Var=acebfd11g
echo ${Var^^[c-f]} # prints 'aCEbFD11g'
```

The pattern '^^[c-f]' will capitalize all single characters, but only in the specified range 'c-f', therefore only 'c', 'd', 'e' and 'f' in the above example get capitalized.

## 2. Arrays

**Bash** also supports arrays, i.e. variables containing multiple values. Since all variables in **Bash** by default are strings, you can store in the very same array integers, text, etc. Array index in **Bash** starts with zero, and there is no limit to the size of an array. There are a few ways in which an array can be initialized with its elements --- the quickest one is to use the round braces `( ... )`. This syntax is illustrated with the following code snippet:

```
SomeArray=( 5 a ccc 44 )
```

Array elements are separated with one or more empty characters. To reference the content of a particular array element, we use again the curly brace notation `${ArrayName[index]}`. For instance, for the above example we have:

```
echo ${SomeArray[0]} # prints '5'
echo ${SomeArray[2]} # prints 'ccc'
```

To get programmatically all array entries, we can use `${ArrayName[*]}` or `${ArrayName[@]}` syntax, for instance:

```
echo ${SomeArray[*]} # prints '5 a ccc 44'
```

The difference between `${ArrayName[*]}` or `${ArrayName[@]}` syntax matters only when used within double quotes, and the explanation is the same as for a difference between `"$*"` and "$@" when referring to the list of positional parameters (see Lecture #2).

This means that we can very conveniently loop over all array entries with:

```
for Entry in ${SomeArray[*]}; do
 echo $Entry
done
```

The printout is:

```
5
a
ccc
44
```

The total number of elements in an array is given by the syntax `${#ArrayName[*]}`:

```
echo ${#SomeArray[*]} # prints '4'
```

We can set the value of a particular array element directly:

```
SomeArray[2]=ddd
```

Now if we print all elements, the initial 3rd element 'ccc' was replaced with the new value 'ddd', and we get:

```
echo ${SomeArray[*]} # prints '5 a ddd 44'
```

In order to remove a particular element of an array, we need to explicitly use the keyword **unset**. This way, the length of an array and all indices are automatically recalculated:

```
unset SomeArray[2]
echo ${SomeArray[*]} # prints '5 a 44'
echo ${#SomeArray[*]} # prints '3', the array was resized
```

On the other hand, unsetting the array element with:

```
SomeArray[2]= # WRONG!!
```

is wrong, since the total size of an array was not reset, i.e. this particular element is still counted as a part of an array, only it has now NULL content.

The whole array can be reset either with

```
unset SomeArray
```

or

```
SomeArray=()
```

The array index works also backward. The last array element is:

```
echo ${SomeArray[-1]}
```

the penultimate array entry is:

```
echo ${SomeArray[-2]}
```

and so on. To append directly to the already existing array a new element, we can use programmatically the following code snippet:

```
SomeArray[${#SomeArray[*]}]=SomeValue
```

The above syntax works, because array indexing starts from 0 and ends with N-1, where N is the total number of array elements. Since `${#SomeArray[*]}` gives the total number of array elements N, the above syntax just appends the new N-th entry.

Quite frequently, we need to prepend or append the same string to all array elements. This can be achieved elegantly with the following syntax:

```
SomeArray=${SomeArray[*]/#/SomePattern} # prepend
SomeArray=${SomeArray[*]/%/SomePattern} # append
```

**Example 1:** We have the following starting array which just contains some file names:

```
Files=( file_0 file_1 file_2 )
```

How to append to all file names the same file extension '.dat'? How to prepend to all file names the same string 'some_'?

The solution to the first question is:

```
Files=( ${Files[*]/%/.dat} )
```

In the above code snippet, we have first appended (by specifying `%`) to all array elements the same extension '.dat', and immediately redefined the array to the new content. The array elements are now:

```
echo ${Files[*]}
file_0.dat file_1.dat file_2.dat
```

The solution to the second question is:

```
Files=( ${Files[*]/#/some_} )
```

In the above code snippet, we have first prepended (by specifying `#`) to all array elements the same string 'some_' , and we have then redefined the array to the new content, so the array elements are now :

```
echo ${Files[*]}
some_file_0.dat some_file_1.dat some_file_2.dat some_file_3.dat
```

By using this functionality, we can prepend programmatically to all file names in a given directory the absolute path to that directory --- we just need to prepend the pattern **${PWD}/**.

The power and flexibility of arrays come from the fact that at array declaration within `( ... )` a lot of other **Bash** functionalities are supported, for instance, the command substitution operator `$( ... )` and brace expansion `{ ... }`. That in particular means that we can effortlessly store the entire output of a command into an array, and then do some manipulation element-by-element.

**Example 2:** Count the number of words in an external file using arrays.

The solution is very simple and elegant:

```
FileContent=$(< SomeFile)
SomeArray=( ${FileContent} )
echo "Number of words: ${#SomeArray[*]}"
```

In the first line we have stored the content of an external file `SomeFile` into variable **FileContent**, and then just defined the array elements by referencing its content. The empty characters which separate the words in the file, now separate the array elements in the definition.

At the expense of becoming a bit cryptic, the above solution can be condensed even further:

```
SomeArray=( $(< SomeFile) )
echo "Number of words: ${#SomeArray[*]}"
```

**Example 3:** How to append entries of one array to another, without using loops?

The solution is:

```
Array_1=( 1 2 3 )
Array_2=( a b c d )
NewArray=( ${Array_1[*]} ${Array_2[*]} )
echo ${NewArray[*]} # prints '1 2 3 a b c d'
```

**Example 4:** Using brace expansion at array declaration.

```
SomeArray=( file_{0..3}.{pdf,eps} )
echo ${SomeArray[*]}
```

The printout is:

```
file_0.pdf file_0.eps file_1.pdf file_1.eps file_2.pdf file_2.eps file_3.pdf
file_3.eps
```

**Example 5:** How to store the output of some command in array?

```
SomeArray=( $(date) )
```

We can now extract from the output of **date** only a particular entry:

```
echo ${SomeArray[*]}
# prints: 'Fri May 29 16:24:25 CEST 2020'
echo "Current month: ${SomeArray[1]}"
# prints: 'Current month: May'
echo "Current time: ${SomeArray[3]}"
# prints: 'Current time: 16:24:25'
```

**Example 6:** How can we catch the user's input directly into an array?

We have already seen that by using **read** command we can catch the user's input, but if we want to store the input in a few different variables, that quickly becomes inconvenient. And quite frequently, we cannot foresee the length of the user's input. For instance, how to handle the user's reply to the question: "Which countries you visited?" That can be solved elegantly with arrays:

```
read -p "Which countries you visited? " -a Countries
```

By using the flag **-a** for command **read**, we have indicated that whatever user types, it will be split according to the empty character, i.e. the default input field separator into words, and then each word is stored as a separate element in an array (in the above example, that arrayed is named 'Countries').

Then we can immediately write for instance:

```
echo "Number of countries is: ${#Countries[*]}"
echo "The first country is: ${Countries[0]}"
echo "The last country is: ${Countries[-1]}"
```

But what if the user visited New Zealand or Northern Ireland? Since these countries contain an empty character in their names, the code above clearly cannot correctly handle these cases. In general, the problems of this type are solved by temporarily changing the default input field separator. The default input field separator is stored in the environment variable **IFS**, and a lot of **Linux** commands rely on its content. We can proceed in the following schematic way:

```
DefaultIFS="$IFS" # save default
IFS=somethingNew
... some code with new IFS ...
IFS="$DefaultIFS" # revert back
```

This is the frequently encountered case in practice, when a certain variable needs to be set only during the command execution. There exists a specialized syntax applicable to cover such uses cases:

```
SomeVariable=someValue SomeCommand
```

Note that there is no semicolon `;` between variable definition and command execution, this way the new definition of variable **SomeVariable** is visible only during the execution of **SomeCommand**. As soon as command terminates, **SomeVariable** gets automatically reset to its default value (if any).

The final solution for our example is therefore:

```
IFS=',' read -p "List (comma separated) countries you visited: " -a Countries
```

This way, the input field separator will be comma `,` but only during the execution of **read**.

Now if the User replied 'New Zealand,Northern Ireland' we have that:

```
echo ${Countries[0]}
# prints: New Zealand
echo ${Countries[1]}
# prints: Northern Ireland
```

As the final remark on arrays, we indicate that multidimensional (associative) arrays are rarely used in **Bash**, but nevertheless, they are supported. They need to be declared explicitly with **Bash** built-in command **declare** and flag **-A**:

```
declare -A SomeArray
```

After such declaration, **Bash** understands how to cope with the following syntax:

```
SomeArray[1,2,3]=a
SomeArray[2,3,1]=bb
```

To reference the content of elements in multidimensional arrays, we use:

```
echo ${SomeArray[1,2,3]} # prints 'a'
echo ${SomeArray[2,3,1]} # prints 'bb'
```

The indices do not have to be hardwired --- index of **Bash** arrays can be any expression that evaluates to 0 or a positive integer.

## 3. Piping: `|`

We have already seen that commands can take their input directly from the user or from files. But in general, one command can take automatically the output of another command as its input. This mechanism is called *piping* and it is a very generic concept in **Linux**.

In order to use the output of one command as the input to another, we use operator `|` ('pipe'), schematically as:

```
firstCommand | secondCommand
```

It is possible to chain with pipe operator `|` multiple commands:

```
firstCommand | secondCommand | thirdCommand | ...
```

In the above example, the successful output, i.e. the *stdout* stream, of `firstCommand` has become the input, i.e. the *stdin*, to `secondCommand`. That command now processes that input, and produces its own output, which is then becoming the input to the `thirdCommand`, and so on.

It is possible to redirect simultaneously both *stdout* and *stderr* stream of one command into *stdin* of another, with the slightly modified pipe operator `|&`, schematically:

```
firstCommand |& secondCommand
```

In the above example, both the successful output stream and the error message of `firstCommand` are simultaneously redirected as an input to `secondCommand`.

Usage of pipe `|` eliminates the need for making temporary files to redirect and store the output of one command, and then supply that temporary file as an input to another command. All commands chained with `|` in the pipeline are running simultaneously and data flow among them is automated without any restriction on the size.

We now provide a few frequently use cases of piping. We have already seen that **Bash** supports directly only integer arithmetic with the construct `(( ... ))`. The floating-point arithmetic in **Bash** can be done by piping the desired expression into the external **Linux** program **bc** ('basic calculator').

**Example 1:** How would you divide 10/7 at the precision of 30 significant digits?

The solution is given by the following code snippet:

```
echo "scale=30; 10/7" | bc
```

The output is:

```
1.428571428571428571428571428571
```

The keyword **scale** sets the precision in **bc** program. Instead of using **bc** interactively and providing via keyboard *stdin* for its execution, we have just piped the *stdout* of **echo** as an input to **bc**.

For more sophisticated use cases, for instance when you want to use special mathematical functions, etc, use **bc -l**. The flag '-l' (ell) loads in the memory in addition the heavy mathematical libraries, which are otherwise not needed for simple calculations. If the scale is not specified, it has defaulted to 1 when **bc** is called, and to 20 when **bc -l** is called.

The math library of **bc** defines the following example functions:

```
s(x) : The sine of x, x is in radians.
c(x) : The cosine of x, x is in radians.
a(x) : The arctangent of x, arctangent returns radians.
l(x) : The natural logarithm of x.
e(x) : The exponential function of raising e to the value x.
j(n,x) : The bessel function of integer order n of x.
```

**Example 2:** How would you calculate `e^2` to the precision of 20 significant digits?

```
echo "e(2)" | bc -l # prints '7.38905609893065022723'
```

Another typical use case of pipe operator `|` is in a combination with **tee** command. Quite frequently, when a certain command is executing, we would like to see its output on the screen, but also simultaneously redirected to some file, so that any time later we can carefully inspect the whole command output by reading through the content of that file.

This can be achieved with the **tee** command, schematically:

```
someCommand | tee someFile.log
```

For instance, the code snippet:

```
date | tee date.log
```

will print the current time on the screen, but it will also simultaneously dump it permanently in the file named `date.log` (check its content with **cat date.log**). In the very same spirit, it is possible to keep the full execution log of any script, function, code block `{ ... }`, loops, etc.

The command **tee** writes simultaneously its input to *stdout* (screen) and redirects it to the files. By default, **tee** overwrites the content of a file, if we want to append instead to the already existing non-empty file, use the following version:

```
someCommand | tee -a someFile.log
```

Flag '-a' in this particular case stands for 'append'.

As the final remark on the pipelines, we consider the following important question: If the pipeline, composed of multiple commands, has failed during execution, how to figure out programmatically which particular command(s) in the pipeline have failed? In order to answer this question, we need to inspect the status of the built-in variable **PIPESTATUS**. This variable is an array holding the exit status of each command in the last executed pipeline:

```
echo "scale=5000; e(2)" | bc -l | more
echo ${PIPESTATUS[*]} # prints '0 0 0'
```

In the above example, we want to determine the result to 5000 significant digits, and then inspect through it screen-by-screen with the **more** command. All three commands in the pipeline, **echo**, **bc** and **more**, executed successfully, therefore the array **PIPESTATUS** holds three zeroes. If only the single command has been executed, that is a trivial pipeline, and **PIPESTATUS** array has only one entry, the very same information which is stored in **$?** variable. The thing to remember is that **PIPESTATUS** gets updated each time we execute command, even the trivial ones like **echo**.

The power of piping is best illustrated in the combination with the three powerful commands **sed**, **awk** and **grep**, which are the core **Linux** utilities for text parsing and manipulation, which we cover in the next section.

## 4. sed, awk and grep

Frequently a text needs to be parsed through and inspected, or updated after the search for some patterns has been performed, in general, modified programmatically for one reason or another. The text in this context can stand for any textual stream coming out of command upon execution, or for any text saved in some physical file. Clearly, it is impractical and in some cases unfeasible to make all such changes in some graphics based editors. In this section, we cover instead how the text can be manipulated programmatically, with the three core **Linux** commands: **grep**, **awk** and **sed**. Combining functionalities of all three of them gives a lot of power when it comes to programmatic text manipulation, and typically covers all cases of practical interest. The usage of these three commands is best learned from concrete examples.

**grep**

The command **grep** ('Globally search a Regular Expression and Print') is used to filter out from the command output or from the physical file the lines containing a certain pattern. Typically, this command is used as follows:

```
grep SomePattern(s) SomeFile(s)
```

The above syntax will select from the specified files only the lines which conform to the specified patterns, and print them on the screen.

Another frequent use case is:

```
SomeCommand(s) | grep SomePattern(s)
```

The above syntax will select on-the-fly from the output of a command only the lines which conform to the specified patterns, and will print them on the screen.

**Example 1:** Copy and paste in the file `grepExample.txt` the following lines:

```
TEST Test test 11test test22
test TEST Test 11test test22
TeST1 TEST1 TESt1 TEST1 TEST1
test TEST Test 11test test
TeST2 TEST2 TEsT2 TEST2 tEST2
```

By using this example file, we now present the most frequently used cases of **grep** command:

```
grep "test" grepExample.txt
```

The output is:

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

By default, the specified pattern ('test' in the above example) is case sensitive and it doesn't have to be an exact match, therefore here the patterns '11test', 'test22' and 'test' were all the matching patterns. Each line which contains one or more of matching patterns is printed by **grep** on the screen by default, but it can be also redirected to a physical file:

```
grep "test" grepExample.txt > filtered.txt
```

In the next example, we instruct **grep** to use a flag '-n', to print all lines containing the pattern 'test', and the numbers of those lines:

```
grep -n "test" grepExample.txt
```

The result is:

```
1:TEST Test test 11test test22
2:test TEST Test 11test test22
4:test TEST Test 11test test
```

We can easily inverse the pattern search, when we need to print all lines in a file which do not contain the pattern 'test', by using the flag '-v':

```
grep -v "test" grepExample.txt
```

Now only the lines which do not contain the pattern 'test' are printed on the screen:

```
TeST1 TEST1 TESt1 TEST1 TEST1
TeST2 TEST2 TEsT2 TEST2 tEST2
```

When we need case insensitive search, we need to use the flag '-i':

```
grep -i "test" grepExample.txt
```

This prints all lines in the file which contain all case insensitive variants of pattern 'test', e.g. 'TEST', 'Test', 'tEsT, etc:

```
TEST Test test 11test test22
test TEST Test 11test test22
TeST1 TEST1 TESt1 TEST1 TEST1
test TEST Test 11test test
TeST2 TEST2 TEsT2 TEST2 tEST2
```

Since each line has at least one case insensitive variant of the specified pattern 'test', the whole file is printed.

Very frequently, we need to filter out all lines in the file which contain the specified pattern only at the very beginning of the line. This is achieved by using the special character '^' (caret):

```
grep "^test" grepExample.txt
```

This results in:

```
test TEST Test 11test test22
test TEST Test 11test test
```

The special character '^' is an anchor for the beginning of a line, and a lot of other commands interpret this character in the same fashion. Opposite to it, if we need to print all lines in the file which contain the specified pattern only at the end of the line, we need to use '$':

```
grep "t22$" grepExample.txt
```

The result is the following two lines:

```
TEST Test test 11test test22
test TEST Test 11test test22
```

In this particular context, the special character '$' is an anchor for the end of a line.

We can perform the pattern search with **grep** even more differentially. If we need to filter out all lines in the file which contain at least one word beginning with the specified pattern, we can proceed in the following way:

```
grep "\<TeST" grepExample.txt
```

Now both 'TeST1' and 'TeST2' will match, since they begin with the specified pattern 'TeST', and the result is:

```
TeST1 TEST1 TESt1 TEST1 TEST1
TeST2 TEST2 TEsT2 TEST2 tEST2
```

Complementary to this option, we can filter out all lines in the file which contain at least one word ending with the specified pattern:

```
grep "ST\>" grepExample.txt
```

Now only 'TEST' will match, since this is the only word in the file which ends up with the specified pattern 'ST', and in the printout we get only the three lines holding 'TEST':

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

Whet it comes to the exact pattern match, we need to use flag '-w':

```
grep -w "Test" grepExample.txt
```

This yields to the following printout:

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

Each of these three lines has at least one exact occurrence of the specified pattern 'Test'.

It is also possible to combine patterns, with the special character '\|':

```
grep "11test\|test22" grepExample.txt
```

This prints all lines containing either the pattern '11test' or 'test22' (this is logical OR operation):

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

We cannot directly use **grep** to obtain the logical AND operation in the pattern search, but this limitation can be circumvented with the usage of pipe:

```
grep "11test" grepExample.txt | grep "test22"
```

This will print all lines which contain both specified patterns:

```
TEST Test test 11test test22
test TEST Test 11test test22
```

In this example, this first **grep** in the pipeline acted on a physical file, while the second **grep** got its input from the output stream of the first **grep**. Whether the input to **grep** is coming from the physical file, or via pipe `|` from the *stdout* or *stderr* stream of some other command, its usage is completely equivalent.

For instance, you can check if the variable has some pattern schematically with:

```
echo $Var | grep SomePattern(s)
```

In cases where only the check for the pattern needs to be performed with **grep**, and there is no not need for the actual printout, we can use the flag '-q' (for quite), like in this example:

```
if grep -q "11test" grepExample.txt; then
 ... some code ...
elif grep -q "test22" grepExample.txt; then
 ... some other code ...
else
 ... yet other code ...
fi
```

**Example 2:** How to select in the current directory only the files whose names begin with the example pattern 'ce' and end up with the pattern '.dat'? The content of the directory is:

```
array.sh   be3.dat  be8.dat  ce1.log  ce4.dat  ce6.log  ce9.dat
array.sh~  be4.dat  be9.dat  ce2.dat  ce4.log  ce7.dat  ce9.log
be0.dat    be5.dat  ce0.dat  ce2.log  ce5.dat  ce7.log  grepExample.txt
be1.dat    be6.dat  ce0.log  ce3.dat  ce5.log  ce8.dat  test.sh
be2.dat    be7.dat  ce1.dat  ce3.log  ce6.dat  ce8.log  test.sh~
```

The solution is:

```
ls | grep "^ce" | grep ".dat$"
```

The **ls** command will print first the list of all files in the current directory, and pipe that list to **grep** for further filtering. Then **grep** filters out the lines in the output of **ls** which begin (the anchor '^') with the pattern 'ce'. That results is then filter further by chaining another pipe. In the 2nd **grep** we have used the anchor '$' since we are interested in the ending. The final output is:

```
ce0.dat
ce1.dat
ce2.dat
ce3.dat
ce4.dat
ce5.dat
ce6.dat
ce7.dat
ce8.dat
ce9.dat
```

**awk**

Now we move to **awk** (named after the initials of its authors: Aho, Weinberg and Kernighan), which is a programming language by itself, designed for text processing. One can easily teach the whole semester only about **awk**, here we will cover only its most important functionalities and which are not available as built-in **Bash** functionalities. The frequently heard comment about **awk** is that its syntax and usage are awkward. Nevertheless, in a lot of cases of practical interest **awk** provides the best and the most elegant solution.

After we supply some input to **awk**, it will break each line of input into fields, separated by default with one or more empty characters. After that, **awk** parses the input and operates on each separate field. Just like with the **grep** command, **awk** can take its input either from a physical file, or from the output stream of another command via a pipe. For instance, if a certain command has produced an output that consists of column-wise entries separated with one or more empty characters, we can get hold of each field programmatically. For instance:

```
date
date | awk '{print $4}'
```

The output is only the current time:

```
Wed Jun  3 15:36:12 CEST 2020
15:36:12
```

In the 2nd line, we have, by using **awk**, isolated directly only the 4th field in the **date** output. In the similar fashion:

```
date | awk '{print $6}'
2020
```

prints only the year, because the 6th field in the output of **date** is reserved for a year.

We can select multiple fields, and immediately on-the-fly do some additional editing:

```
date | awk '{print $4, "some text", $6}'
15:36:12 some text 2020
```

In the same way **awk** operates on the content of files, the file content just needs to be redirected in **awk** with '<' redirector. It is very convenient, for instance, to use **awk** to extract only the values from the specified column(s) in a file. For instance, if the content of the file `someFile.dat` is:

```
a 1
yy 10
c 44
```

we can extract the columns separately with

```
awk '{print $1}' < someFile.dat
a
yy
c
```

and

```
awk '{print $2}' < someFile.dat
1
10
44
```

Typically, one can store such an output in an array, and then process further programmatically all entries, with the following code snippet which combines a few different functionalities covered by now:

```
SomeArray=( $(awk '{print $2}' < someFile.dat) )
echo ${SomeArray[*]}
1 10 44
```

In order to get the total number of fields, we can use the **awk** built-in variable **NF**:

```
date
date | awk '{print NF}'
```

Since the output stream of **date** has 6 entries separated with the empty character, we get:

```
Wed Jun  3 15:36:12 CEST 2020
6
```

On the other hand, the entry from the last field can be achieved directly by referencing the content of **NF** variable:

```
date | awk '{print $NF}'
2020
```

Similarly, the entry from the penultimate field can be obtained directly with:

```
date | awk '{print $(NF-1)}'
CEST
```

and so on.

But what if we want to parse the command output or file content even more differentially? For instance, what if we want to extract programmatically from the output of **date** command only the minutes, and not the full timestamp '15:36:12' by specifying the 4th field? In order to achieve that, we need to change the field separator in **awk** to some non-default value. This is achieved by manipulating the **awk** built-in variable **FS**. To set the field separator **FS** to some non-default value, we use schematically the following syntax:

```
awk 'BEGIN {FS="some-new-single-character-field-separator"} ... '
```

For instance, if we want to use colon `:` as a field separator in **awk**, we would use:

```
awk 'BEGIN {FS=":"} ... '
```

To extract only the minutes from the output of **date** command, we can use the following code snippet:

```
date
date | awk '{print $4}' | awk 'BEGIN {FS=":"}{print $2}'
```

The output is:

```
Wed Jun  3 16:18:44 CEST 2020
18
```

What happened above is literally the following:

1. the command **date** produced its output `Wed Jun 3 16:18:44 CEST 2020`
2. that output was piped as an input for further processing to **awk** command, which extracted the 4th field, taking into account that the default field separator is one or more empty characters. The result after this step was `16:18:44`

3. this intermediate output stream `16:18:44` was then sent via another pipe to **awk** command, which, however, in the 2nd pipe runs with non-default field separator `:` . With respect to `:` as a field separator in the stream `16:18:44`, the 2nd field is minutes, which finally yields the final output `18`

As a rule of thumb, field separators in **awk** shall be always single characters---composite multi-character field separators are possible, but can lead to some inconsistent behavior among different **awk** versions (e.g. **gawk**, **mawk**, etc.).

On the other hand, different single characters can be treated as field separators simultaneously, they just all need to be embedded within `[ ... ]`. For instance, we can treat during the same **awk** execution all three characters colon `:`, semi-colon `;` and comma `,` as equivalent field separators in the following code snippet:

```
echo "1,22;abc:44:1000;123" | awk 'BEGIN{FS="[:;,]"} {print $4}'
```

The output is

```
44
```

The main limitation of **awk**, when used within **Bash** scripts, is that it cannot directly process the values from the **Bash** variables. We need to initialize first with additional syntax some internal **awk** variables with the content of **Bash** variables, before we can use them during **awk** execution, which in practice can be a bit, well, awkward. This limitation is not present in the command **sed**, which we cover next.

**sed**

Finally, there is **sed** ('Stream Editor'), a non-interactive text file editor. It parses the command output or file content line-by-line, and performs specified operations on them. Typically, **sed** covers the following use cases:

1. printing selected lines from a file
2. inserting new lines in a file
3. deleting specified lines in a file
4. searching for and replacing the patterns in a file

We illustrate all four use cases with a few basic examples.

**Example 1:** How to print only the specified lines from the command output? As a concrete example, we consider the output of **stat** command:

```
stat test.sh
```

The output is:

```
  File: test.sh
  Size: 177             Blocks: 0          IO Block: 4096   regular file
Device: 2h/2d   Inode: 21673573207029672  Links: 1
Access: (0666/-rw-rw-rw-)  Uid: ( 1000/abilandz)   Gid: ( 1000/abilandz)
Access: 2020-05-01 12:46:20.551223700 +0200
Modify: 2020-05-29 08:32:38.081673700 +0200
Change: 2020-05-29 08:32:38.081673700 +0200
 Birth: -
```

If we are interested only to print on the screen only a particular line, we need to use **sed** with the flag '-n' and the specifier 'p' ('print'). Flag '-n' is needed to suppress the default printout of the original file. To print only the 2nd line, we can use the following syntax:

```
stat test.sh | sed -n 2p
```

The output is now only the 2nd line:

```
Size: 177             Blocks: 0          IO Block: 4096   regular file
```

With the slightly modified specifier 'p' we can indicate the line ranges. For instance, the syntax

```
stat test.sh | sed -n 2,5p
```

will print lines 2, 3, 4 and 5, and so on.

**Example 2:** How to insert a new 2nd line of text in the already existing file `sedTest.dat`, which has the following content:

```
line 1
line 2
line 3
line 4
```

In general, to insert a new line with **sed**, we need to use the specifier 'i'. The solution is:

```
sed "2i Some text" sedTest.dat
```

This will insert in the second line (the meaning of '2i' specifier) of the file `sedTest.dat` the new text 'Some text'. The original file is not modified, only the **sed** output stream. The **sed** output stream on the screen is:

```
line 1
Some text
line 2
line 3
line 4
```

The above modified output stream can be redirected to a new file with `1> someFile`, but we can also modify in-place the original file. To achieve this, we need to use the flag `-i` ('in-place edit') for **sed** :

```
sed -i "2i Some text" sedTest.dat
```

This will in the 2nd line of the file `sedTest.dat` insert the new text 'Some text' and the original file is modified, without backup. Remember in this context different meaning of 'i':

- '-i' is a flag which instructs **sed** that we want to modify the original file in-place
- 'ni' as an argument indicates that we want to insert something on the nth line

Clearly, it can be potentially dangerous to modify directly the original file in-place, because once the original file is overwritten, there is no way back. To prevent that, we can automatically create the backup of the original file by using the slightly modified flag '-i.backup':

```
sed -i.backup "2i Some text" sedTest.dat
```

This will in the second line of the file `sedTest.dat` insert the text new 'Some text'. The original file is modified, however now also the backup of the original file was created automatically, and is saved in new file named `sedTest.dat.backup`.

**Example 3:** How to delete the 4th from the above file `sedTest.dat`?

To delete lines of the file's of the command's output, we need to use the specifier 'd' ('delete') in **sed**. For instance, if we want to delete the 4th line, we can use the following syntax:

```
sed "4d" sedTest.dat
```

This will delete the 4th ('4d' specifier) line in the file `sedTest.dat`. We can also specify the line ranges for deletion, for instance:

```
sed "2,4d" sedTest.dat
```

This will delete the 2nd, 3rd and 4th lines in the file `sedTest.dat`. The previous comments about in-place modification and backup of the original file apply also in this context.

**Example 4:** Finally, we also illustrate how to replace one pattern in the file with another. This is achieved with the following generic syntax:

```
sed "s/firstPattern/secondPattern/" someFile
```

This will substitute ('s' specifier) in each line of file `sedTest.dat` only the first occurrence of `firstPattern` with `secondPattern`. On the other hand, if we want to replace all occurrences, we need to use the following, slightly modified syntax:

```
sed "s/firstPattern/secondPattern/g" someFile
```

Note the additional specifier 'g' (for 'global') at the end of an expression. For instance, if we consider the file `example.log` with the following content:

```
momentum energy
energy momentum momentum
momentum energy momentum
```

We can replace only the first occurrence of 'momentum' with 'p' on each line with the following syntax:

```
sed "s/momentum/p/" example.log
```

The result is:

```
p energy
energy p momentum
p energy momentum
```

On the other hand, we can replace all occurrence of 'momentum' with 'p' on each line with the slightly modified syntax:

```
sed "s/momentum/p/g" example.log
```

Now the result is:

```
p energy
energy p p
p energy p
```

The very convenient thing about **sed** is that it can interpret **Bash** variables directly. It is perfectly feasible in your script to have something like:

```
Before=OldPatern
After=NewPatern
sed "s/${Before}/${After}/" someFile
```

This gives a lot of flexibility, because old and new patterns can be supplied via arguments, etc. In the same spirit, we can use **sed** to modify on-the-fly the output stream of any command:

```
date
date | sed "s/Wed/Wednesday/"
```

The output is:

```
Wed Jun  3 21:08:49 CEST 2020
Wednesday Jun  3 21:08:49 CEST 2020
```

Another example, when we insert a new line on-the-fly in the output stream of some command:

```
stat test.sh | sed "4i => File permissions, and other thingies:"
```

The output is:

```
  File: test.sh
  Size: 62              Blocks: 0          IO Block: 4096   regular file
Device: 2h/2d   Inode: 26740122787573808  Links: 1
=> File permissions, and other thingies:
Access: (0666/-rw-rw-rw-)  Uid: ( 1000/abilandz)   Gid: ( 1000/abilandz)
Access: 2020-05-11 12:38:23.820690300 +0200
Modify: 2020-05-14 13:13:46.970442600 +0200
Change: 2020-05-14 13:13:46.970442600 +0200
 Birth: -
```

Finally, **sed** provides full support for pattern matching via regular expressions, which increases its power and applicability tremendously.


# Lecture 7: Escaping. Quotes. Handling processes and jobs.

**Last update**: 20200625

## Table of Contents



## 1. Escaping: `\`

Some characters in **Bash** have a special meaning. As an elementary example:

```
echo $var
```

would reference the content of variable named `var` since the variable is preceded with the special character `$`, but `$` itself would not appear in the printout. To see also the special character `$` in the printout, we need to *escape* (or kill) its special meaning with the backslash character `\`.

The escaping mechanism in **Bash** is illustrated in the following example:

```
var=44
echo $var
echo \$var
```

The above code snippet produces the following output:

```
44
$var
```

It is possible in the same way to escape the special meaning of any other special character, and in any other context (not necessarily only in their printout as demonstrated here). If there are multiple special characters in the input expression, they can be escaped one-by-one with backslash `\`.

As another example, we consider the double quotes `"..."`, which also have a special meaning in **Bash** (clarified in the next section!) and are not printed by default:

```
echo "Hi "there""
Hi there # no quotes in the printout
```

However, we can escape the special meaning of two inner-most quotes, and they will show up in the printout:

```
echo "Hi \"there\""
Hi "there"
```

Finally, we can also escape the special meaning of two outer-most quotes:

```
echo \"Hi \"there\"\"
"Hi "there""
```

As another example, we compare:

```
echo "Today is: $(date)"
Sat Jun  6 19:24:20 CEST 2020
```

with

```
echo "Today is: \$(date)"
Today is: $(date)
```

In the second example nothing happened, because the command substitution operator was escaped.

We have already seen that in **Bash** the command input is terminated either with semicolon `;` or with the new line. Frequently, the command input needs to span over a few lines in the terminal, and in order to handle such a case, we need to escape the end of the line, i.e. we need to kill the special meaning of a new line. To achieve that, it suffices to place backslash `\` at the very end of the line:

```
echo "Welcome \
to \
the \
lecture PH8124."
```

This produces the one-line output:

```
Welcome to the lecture PH8124.
```

When used to escape the new line, backslash `\` must be the very last character on that line. The frequent mistake occurs when `\` is followed by an empty character, because then it will merely escape the special meaning of an empty character, and not the special meaning of a new line.

Alternatively, we can escape the meaning of special characters with strong (single) quotes `'...'`, which is the topic of the next section.

## 2. Quotes: `'...'` and `"..."`

**Strong (single) quotes**

In complex expressions, containing a huge number of special characters, it becomes quickly impractical to escape the special meaning of each character separately with `\`. Instead, they can be escaped all in one go by embedding the whole expression within strong (single) quotes `'...'`. This is the primary use case of strong quotes, and their meaning can be literally understood with the following phrase: *what you see is what you get*.

For instance:

```
Var1=44
Var2=440
echo '$Var1   $Var2'
```

The printout is literally:

```
$Var1   $Var2
```

Neither variable was referenced, because the special meaning of both `$`'s was killed with strong quotes, and the exact number of empty characters was also preserved in the printout.

Strong quotes are used frequently to pass the file or directory whose name contains empty characters:

```
ls 'crazy name'
```

Without strong quotes, the command **ls** would interpret 'crazy' and 'name' separately, as two different arguments.

To illustrate the importance of strong quotes, consider the following example:

```
echo 100 > 10 # WRONG!!
```

We wanted to print a literal inequality, `100 > 10`, on the screen but the above code snippet didn't produce any printout on the screen. Instead, something completely different has happened: **echo** printed only `100` but that was redirected immediately into the file named `10`. We can circumvent this problem with:

```
echo '100 > 10'
100 > 10
```

Single quotes may not occur between single quotes, even when preceded by a backslash.

As the last remark, strong quotes appear in a rarely used context, which is here outlined just for completeness sake. Some characters cannot be represented with the literal syntax, instead, we need to use *backslash-escaped characters* for them. The best examples are new line and tab space, which are represented with '\n' and '\t', respectively. However, neither **Bash** nor a lot of **Linux** commands by default interpret such backslash-escaped characters. For instance:

```
echo "Hi\nthere"
```

prints literally:

```
Hi\nthere
```

We need to instruct **echo** to interpret backslash-escaped characters by supplying a flag '-e':

```
echo -e "Hi\nthere"
```

The printout it now:

```
Hi
there
```

Similarly:

```
echo -e "Hi\tthere"
```

prints the tab space:

```
Hi      there
```

and so on.

In general, we can force **Bash** to interpret directly the backslash-escaped characters by using the following generic syntax**:**

```
$'whatever'
```

With this syntax, we do not rely any longer on the details of command implementation, and whether there exists some option, like '-e' for **echo**, which will instruct the command to interpret backslash-escaped characters. For instance:

```
echo $'Hi\nthere'
```

will print:

```
Hi
there
```

Now **Bash** has interpreted the special meaning of '\n' character, not the **echo** command.

**Example:** Prompt the user with the following multi-line question in **read** command:

```
Dear User,
do you want to continue [Y/n]?
```

The problem here is that the command **read**, unlike **echo**, does not have a specialized flag to interpret the backslash-escaped characters. Therefore, the simplest solution is to use `$` in combination with single quotes:

```
read -p $'Dear User,\ndo you want to continue [Y/n]? ' Answer
```

In the next section, we clarify the meaning of weak (double) quotes `"..."`.

**Weak (double) quotes**

Unlike the strong quotes, the weak (double) quotes `"..."` preserve the special meaning of some special characters, while the special meaning of all others is stripped off. Just like within strong quotes, within double quotes the empty character does not retain its special meaning, i.e. it is not any longer the default field separator. The exact number of empty characters is preserved within weak quotes:

```
echo "a b    c"
echo  a b    c
```

The output is:

```
a b    c
a b c
```

We can now compare the effect of two types of quotes in the following example:

```
var1=44
var2=440
```

- no quotes:

  ```
  echo $var1   $var2
  44 440
  ```

- strong quotes:

  ```
  echo '$var1   $var2'
  $var1   $var2
  ```

- weak quotes:

  ```
  echo "$var1   $var2"
  44   440
  ```

In each case, we got a different result. Within double quotes, the content of variables is referenced with the special character `$` , and the exact number of empty characters is preserved.

The special meaning of the following special characters or constructs are preserved within weak quotes `"..."`:

- `$` : referencing content of variable
- `$( ... )` : command substitution operator
- `$(( ... ))` : arithmetic expression evaluation
- `\` : backslash preserves its special meaning within double quotes only in some cases, for instance, when it is followed by `$`, `"`, `\`, or newline.

Nested double quotes are allowed as long as the inner ones are escaped with `\` . For instance,

```
echo "\"test\""
```

prints

```
"test"
```

as expected. On the other hand, single quotes have no special meaning within double quotes:

```
var=44
echo "'$var'"
```

prints

```
'44'
```

To memorize the rules easier, to leading order only the **Bash** constructs which begin with `$` or `\` keep their special meanings within double quotes.

## 3. Handling processes and jobs

In **Linux** world, an executable stored on disk is called a *program*. Loosely speaking, the program loaded into the computer's memory and running is called a *process*. On the other hand, *job* is more specifically a process that is started by a shell. A group of processes launched from a shell can be also considered as a job. Therefore, a job is a shell-only concept, while a process is a more general, system-wide, concept. There are specific **Bash** and **Linux** commands which keep track only of jobs launched from the current shell, but there are also commands which keep track of all processes running on the computer. Therefore, it is important to understand the difference between processes and jobs, and in which context which commands for their handling need to be used.

Jobs launched from the shell can be divided into two important groups: *foreground* and *background* jobs. Foreground jobs are jobs that have control over the terminal, i.e. while they are running nothing else can be done in the current terminal session by the user. The control over the terminal is regained only when the foreground job has finished its execution. Background jobs are jobs that do not have control over the terminal during their execution. They are typically started on multicore machines, when the parallelization of jobs makes perfect sense and reduces a lot the overall execution time. While jobs launched from the current terminal session are running in the background, in that terminal session we have full control over the terminal and can do whatever we want.

By default, any job which is started from the terminal is executed in the foreground. If we want to submit a job execution to the background, we need to end the command line input with the special character `&` . For testing purposes, in this section we use the dummy command **sleep**, which runs a perfectly valid process even though it does nothing besides blocking the execution of subsequent commands for the specified time interval. Whatever is demonstrated in this section for the **sleep** command applies also to any other command, we use **sleep** command merely because of its simplicity. In addition, a word command is used in this section in the broader sense, and it encapsulates also functions, scripts, code blocks, etc.

To illustrate the difference between foreground and background job execution, we first execute a job in the foreground:

```
sleep 10s
```

With this syntax, the command **sleep** runs in the foreground and therefore it takes over the control over the terminal during its execution. What happens now is that for 10s nothing else can be done in the terminal, until the command **sleep** terminates. If we would have started some other command in the foreground, in the very same spirit during the execution of that command, we could not do anything in the terminal, until that command terminates.

With the slightly modified syntax, we can execute a job in the background:

```
sleep 10s &
```

By using the special character `&` at the end of the command input, we have sent the execution of command **sleep** in the background. The main difference to the previous case is that now we can continue immediately to execute another command in the terminal, while the command **sleep** is running in parallel in the background.

When in some **Bash** code a command is started in the background with `&` at the end of command input, that command essentially starts off another process in parallel (that processes *forks off* from the current shell). Note, however, that the *stdout* stream of the forked process is still attached to the shell from which the job was sent to the background, which means that any output of that job will still appear in your terminal, even if the job is running in the background. This sometimes leads to surprising printout in the terminal, if the *stdout* stream of background job was not redirected somewhere else (e.g. to some file or to `/dev/null`).

It is also perfectly feasible to launch in the same command input multiple processes in separate background sessions:

```
sleep 10s & sleep 20s & sleep 30s &
```

With the above syntax, we have three instances of **sleep** command running in parallel in the background. Analogously, we can start off any other three commands to run in parallel in the background.

Next, we will see how a process can be handled programmatically either via its *Process ID (PID)* or its *Job Number*.

**Process IDs and Job Numbers**

**Linux** gives to each process the unique number, called *Process ID (PID)*, when the process was created. On the other hand, **Bash** gives to each job also the number, called *Job Number*, when some process was started in the current shell. Therefore, each process has a unique system-wide PID, while job numbers are unique only within the current terminal. Each terminal keeps track of its own job numbers. The PID of the running process is the same in all terminals. In general, we can handle programmatically in any terminal a running process via its PID, and in the specific terminal both with its PID and with a job number corresponding only to that specific terminal.

The difference between PID and job number is illustrated with the following code snippet:

```
sleep 10m &
[1] 15
```

In the above example, the number `15` is a system-wide PID, given by **Linux** to the command **sleep 10m** executed in the background. This information is accessible in any terminal on the computer, i.e. `15` is the unique identifier for the command **sleep 10m** across the whole computer. If multiple users are using the same computer, PID is unique for all processes of all

users, which is an essential feature for multitasking.

On the other hand, `[1]` is the job number given by **Bash** (not by the operating system!), to the command **sleep 10m** sent to the background. This information is visible only to the terminal in which this command was executed. In particular, `[1]` in this example indicates that this is the first job sent to the background in the current terminal session, and which is still running. If we have another open terminal, in that terminal `[1]` indicates a completely different job. When job execution of all jobs running in the background terminates, the job counter is reset, and `[1]` can be given later to some other job sent to the background, in the same terminal.

The two most frequently used commands to handle running jobs and processes programmatically are **jobs** and **top**. In order to get the list of running jobs which were submitted only from the current terminal, we can use:

```
jobs -l
```

The output of this command might look for instance:

```
[1]+    15 Running                 sleep 10m &
```

The above output literally means that in the current terminal session there is one job, which:

- was started with the command input **sleep 10m &**
- at the moment is in the state 'Running'
- its job number is `[1]`
- its PID is `15`

Other possible job states include 'Done', 'Terminated', 'Hangup', 'Stopped', 'Aborted', 'Quit', 'Interrupt', etc., and some of them are discussed in detail later.

If we execute another command, for instance:

```
sleep 20m &
jobs -l
```

we now see that both commands are running in parallel in the background (remember, we use **sleep** for its simplicity of usage, but what is explained here applies to any other command):

```
[1]-    15 Running                 sleep 10m &
[2]+    17 Running                 sleep 20m &
```

In the above output, symbol `+` next to the job number indicates the most recent job sent to the background in the current terminal, while symbol `-` indicates the one before the most recent job sent to the background. Only these two jobs get the special treatment and notation in the output of **jobs** command.

We now demonstrate how the running job or process can be terminated programmatically. To terminate the particular job, we need to use the **Bash** built-in command **kill**, either by specifying job number or PID as an argument. The syntax is a bit different, to kill a job by job number we use:

```
kill %2
```

and to kill a job via PID we use:

```
kill 17
```

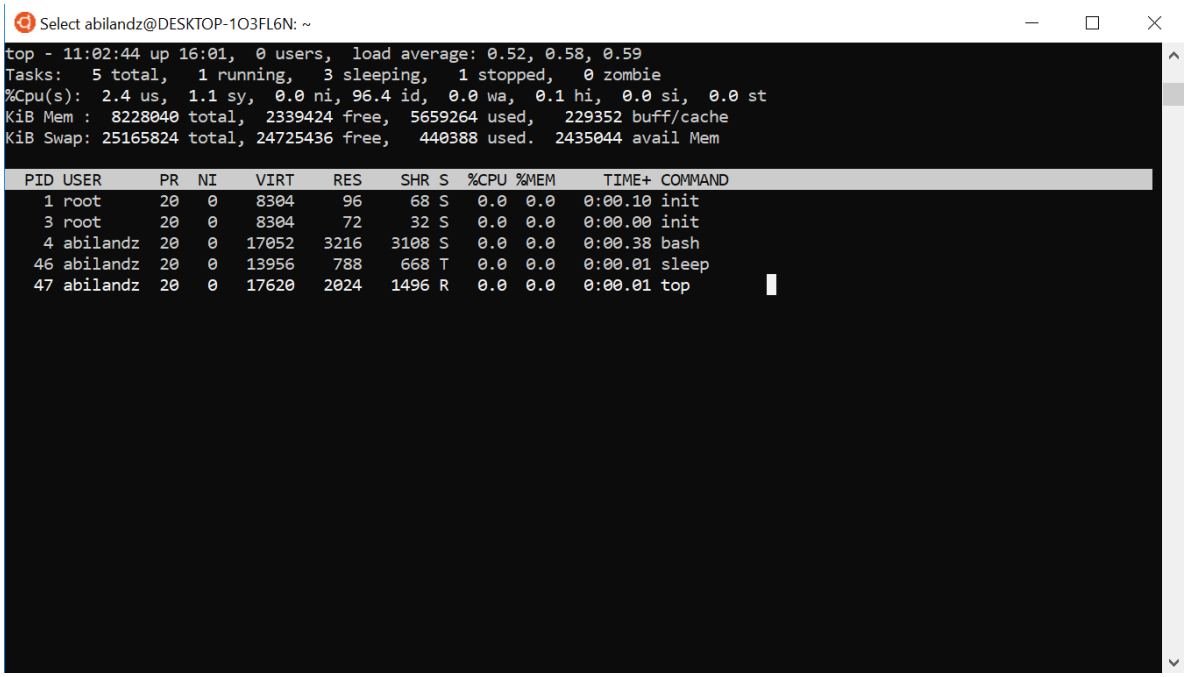
Note the usage of percentage symbol `%` in the first case---without it, **Bash** would attempt to kill the process with system-wide PID 2. Only after the percentage symbol `%` is used, **Bash** will interpret the following number as the job number, which is specific and known only to the current terminal. Note also that only the second version can be used from any terminal, as the PID of any job or process is the same in all terminals. Later we will see that the command **kill**, despite its terse name, can do much more than mere termination of running jobs.

We have seen already how we can get the list of all background jobs started from the current terminal with **jobs -l** command. With the more general command named **top** we can get the list of all running processes on the computer, from all users, running both in foreground and background.

After executing in the terminal:

```
top
```

the output could look like this:




The command **top** continuously updates the terminal display with the summary of the current status of system resources followed by the list of most CPU-intensive processes (default ordering). The first column contains the PID of each running process, followed by the user name, priority of the process, 'nice' value of the process, memory and CPU consumption, total running time, etc. In order to parse the output of **top** programmatically, or to redirect it to some file, we need to run command **top** in the batch (text) mode via:

```
top -b
```

We can dump the output of **top** command to some external file with the following syntax:

```
top -b > topOutput.log
```

We can also pipe that output so some other command, for instance:

```
top -b | grep ${USER}
```

The above code snippet filters out only the information relevant for your own processes.

The command **top** can be run from any terminal on the computer, and its printout to a large extent will be the same in each terminal. On the other hand, the output of the command **jobs** will be completely different from one terminal to another.

Closely related to **top** command is **ps** command (see the corresponding 'man' pages), which gives only the current snapshot of currently active processes, while **top** is being continuously updated and can be used interactively.

In the case you are interested only in the PID of the running process, there is also a command **pidof**, which takes as an argument only the process name:

```
sleep 10s &
[1] 433
pidof sleep
433
```

This command becomes very handy if there are multiple instances of the same command running in parallel, and we need to get PIDs of all of them. For instance:

```
sleep 10s & sleep 20s & sleep 30s &
[1] 587
[2] 588
[3] 589
```

We now have 3 instances of the same command **sleep** running in parallel. We can get the list of all PIDs corresponding to different instances of the same command with:

```
pidof sleep
587 588 589
```

There is also a related command **pkill**, which can terminate on the spot all running instances of the same command, just by its name. For the above example, we can terminate all 3 instances of the command **sleep** running in parallel as follows:

```
pkill sleep
[1]   Done                    sleep 10s
[2]   Done                    sleep 20s
[3]-  Done                    sleep 30s
```

To conclude this section, we remark that one very important process is always listed in the output of **top** command and is called **init**. The process **init** is the grandfather of all processes on the system because all other processes run under it. Every process can be traced back to **init**, and it always has a PID of 1.

**Moving job execution from background to foreground, and vice versa**

We have already seen how the job execution can be sent to the background by appending the special character `&` to the command input. The similar functionality can be achieved with the **Bash** built-in command **bg**, only the syntax and typical use cases are slightly different. Typically, the command **bg** is used after the job was started in the foreground, but then for one reason or another, we need to regain control over the terminal in order to do something else. The trivial solution is to terminate the running job, and then restart it later from scratch. But there is a more elegant and efficient solution, which amounts to the following two generic steps:

1. Suspend the foreground job with `Ctrl+Z`
2. Resume (not restart!) the suspended job in the background with **bg** command

This is best illustrated with the concrete example. Imagine that we have started in the foreground the following command (we stress it out again that the following discussion applies to any other command, **sleep** is used only because of its simplicity!):

```
sleep 10m
```

Now the terminal is blocked for 10 minutes because the command **sleep** is running in the foreground. We can, however, suspend the execution of the command **sleep** by pressing `Ctrl+Z` and regain control of the terminal. After we have regained the control over the terminal, we can start executing other commands. In the meanwhile, the suspended command doesn't do anything:

```
jobs -l
[2]+    15 Stopped                 sleep 10m
```

After pressing `Ctrl+Z` the job was not killed or terminated, it was suspended. The job remains in exactly the same state as it was at the time of the suspension. The suspended job does literally nothing, it is on hold until its execution is resumed. From the user's perspective, the execution of this job appears to be paused. In the output of command **jobs -l** the state description 'Stopped' is a bit misleading, and 'Paused' or even 'Frozen' would be a much better word to describe the state of the job after we suspended it with `Ctrl+Z`. The suspended job will no longer use any CPU, but it will, however, still claim the same amount of RAM. This last fact implies that we can re-start it anytime later and it will continue where it stopped.

To restart the suspended job in the background, we can use the following generic syntax:

```
bg %jobNumber
```

To re-start in the background the above **sleep** command, whose job number is `2`, we need to use:

```
bg %2
```

There is an alternative syntax, which is more limited in scope but sometimes can be nevertheless handier---to restart the suspended job in the background we can also use:

```
bg %commandName
```

If we have only one instance of a given command running and suspended, it suffices to specify only the name of that command to restart it in the background, i.e. it is not needed to specify all options and arguments. However, if we have multiple instances of the same command running with different options and arguments, the whole composite command input needs to be enclosed within strong quotes, for instance:

```
bg %'sleep 10m'
```

If we have multiple instances of the same command running with exactly the same options and arguments, clearly the 2nd version becomes ambiguous. However, we can in that case still use the first syntax and restart the suspended job in the background via its job number, which is always unique.

After restarting the suspended job in the background, we see the following:

```
jobs -l
[2]+    15 Running                 sleep 10m &
```

This is precisely what we wanted to achieve: We have suspended with `Ctrl+Z` the job running in the foreground which was blocking the terminal input, and then restarted its execution in the background with the command **bg %jobNumber**. While that job is now running in parallel in the background, we can do our thing in the terminal again. As the last remark, we stress out that the command **bg** can accept as an argument the job number, but not its PID.

A closely related command is the **Bash** built-in command **fg**. This command moves the jobs running in the background to the foreground. Before discussing its syntax, we first stress out the following important fact: It is impossible solely by using **Bash** built-in features to bring to the foreground a process running in the background in the current shell instance if it was not started in the background from the current shell instance. Basically, this means that you cannot in the current terminal take over a process that was started in a different terminal. To achieve that level of flexibility, there are specialized programs available that allow us to move other programs around from one shell instance to another, for instance **screen**.

After using command **fg**, the background job is continuing to run in the foreground and is, therefore, taking over the control over the terminal. Generically, the syntax of **fg** command is:

```
fg %jobNumber
```

or the alternative version

```
fg %commandName
```

The explanation for both versions of the syntax is the same as for the **bg** command outlined previously, and we do not repeat it here.

We illustrate the usage of **fg** command with the following simple example. First, we launch the command **sleep** (or any other command) in the background:

```
sleep 30m &
```

The relevant line in the output of **jobs -l** command might look like:

```
[5]   5194 Running                 sleep 30m &
```

If we want to bring to the foreground the execution of this background job, we need to use either:

```
fg %5
```

or

```
fg %'sleep 30m'
```

If used without arguments, both **bg** and **fg** commands act on the most recent job.

We finalize this section by mentioning that **Bash** provides the two special variables relevant in this context:

- `$$` : this variable holds the PID of the currently running process
- `$!` : this is the PID of the last job sent to the background

For instance, we can programmatically close the current terminal session by using:

```
kill -9 $$
```

The above line can be placed at the end of the script, if after the script execution we do not need that terminal session any longer. The meaning of option `-9` to command **kill** will be clarified a bit later.

The second special variable, `$!`, has a very neat use case in combination with the **Bash** built-in command **wait**. Quite frequently, we can release the execution burden on the current script by sending part of the execution to the separate processes to run in parallel in the background. We can hold the execution of the main script, and continue only when the last job sent to the background has terminated, with the following syntax:

```
wait $!
```

However, it can happen that the last job sent to the background has terminated before than some other jobs sent to the background earlier. This problem is fixed with the even simpler syntax:

```
wait
```

With the above syntax, the execution of the main script will wait until all jobs running in the background are terminated. This feature is extremely neat on multicore machines: Whenever your script is facing some CPU intensive task, that task can be split across multiple processes, and then each process can be sent independently to the background:

```
commandInput1 &
commandInput2 &
...
wait
```

With the above syntax, while processes **commandInput1**, **commandInput2**, ..., are all running in parallel in the background, the main script waits with further execution. Only when all background processes have terminated, the main script will proceed with further execution.

The classical example when the above functionality can be used is the case when we need to process large datasets. The starting large dataset can be split into subsamples, and then each subsample can be analyzed in parallel, schematically:

```
commandName subsample1 &
commandName subsample2 &
...
wait
```

The main script waits all jobs running in parallel in the background to terminate, and then merely collects the output result of each job, and combines them to obtain the final, large statistic result. Clearly, this feature can considerably speed up the script execution on multicore machines, and all this can be achieved programmatically.

**Sending signals to the running processes**

We have already seen that we can suspend the running job by hitting `Ctrl+Z`, and that we can terminate the running job by executing the command input **kill -9 processPID** in the terminal. Conceptually, there is no much of a difference in what is happening in these two cases, and these two examples are just a small subset of *signals* that we can send to the process. In this section we cover in detail from the user's perspective how the signals can be sent programmatically to the running processes, and modify their running conditions on the fly. In the next section, we will cover this topic from the developer's side, i.e. we will discuss the code implementation which is needed to enable the process to receive and handle the signals while running.

Loosely speaking, a signal is a message that a user sends programmatically to the running process. One running process can also send a signal to another running process. A signal is typically sent when some abnormal event takes place or when we want another process to do something per explicit request. As we already saw, two processes can communicate with pipes `|`. Signals are another way for running processes to communicate with each other.

All available signals have:

- numbers (starting from 1)
- names

To get the list of all signals on your system by name and number, we can execute:

```
kill -l
```

The output could look like:

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

When we are executing in the terminal:

```
kill -9 somePID
```

we are essentially sending the signal number `9`, i.e. the signal with the name `SIGKILL`, to the running process whose PID is `somePID` . For instance, if we start a process in the background:

```
sleep 10m &
[1] 9485
```

we can terminate it either with

```
kill -9 9485
```

or with

```
kill -SIGKILL 9485
```

In both cases, we get the same result:

```
jobs -l
[1]+  9485 Killed                  sleep 10m
```

For the most frequently used signals, there are also the case-insensitive shortcut versions, e.g.:

```
kill -KILL 9485
kill -kill 9485
```

From the table above, we see there are 64 different signals we can send to the running process. Note, however, that some signals are typically used only by the operating system, to tell the process that something went wrong (e.g. division by zero was encountered). As another remark, we indicate that it is somewhat more portable to use signal by its name instead of by its number across different platforms: It is unlikely that the name of the signal like `KILL` will be interpreted in any other way, however number `9` can be.

After we have illustrated the simple use case of **Bash** built-in command **kill**, let us now elaborate on it more in detail. The command **kill** is used to send signals to the already running job, or to any new job. If used without arguments, it will send the default signal to the running process. That default signal is `TERM` ('terminate', number 15), which usually has the same effect as the signal `INT` ('interrupt', number 2). Whenever we execute the following command in a shell:

```
kill somePID
```

we are essentially sending the signal `TERM` ('terminate') to the running process:

```
kill -TERM somePID
```

On the other hand, when we hit `Ctrl+Z` to suspend a running process, we are essentially using a shortcut for the following command input

```
kill -TSTP somePID
```

The signal `TSTP` ('suspend') has a signal number `20`, so pressing `Ctrl+Z` is also equivalent to the following:

```
kill -20 somePID
```

When we hit `Ctrl+C` to interrupt the running process, we are using a shortcut for sending the `INT` signal. Pressing `Ctrl+C` is therefore completely equivalent to:

```
kill -INT somePID
```

or a shorter version (see the above table):

```
kill -2 somePID
```

Yet another way to terminate a running process is to send the `QUIT` signal (number 3):

```
kill -QUIT somePID
```

This case typically produces the message 'core dumped', for instance:

```
sleep 20m &
[2] 11126
kill -QUIT 11126
jobs -l
[2]- 11126 Quit                  (core dumped) sleep 20m
```

The message `Quit (core dumped)` indicates that there is a file called 'core' which contains the image of the process to which you sent a signal. The name 'core' is a very old-fashioned name for computer's memory, and 'core dumps' are generated when the process receives certain signals (such as `QUIT`, `SEGV`, etc.), which the **Linux** kernel sends to the process when it accesses memory outside its address space.

Although it sounds trivial, it makes actually a big difference with which signal we kill the job. Recommended ordering of signals used to terminate the job is the following:

1. **kill** : the default signal is `TERM` (similar to `INT`). If we kill the process this way, we still give a chance to the process to clean up (for instance, to delete all temporary files it was using while running) before terminating. May or may not terminate the running job.
2. **kill -QUIT** : this dumps the process' memory image in the file named 'core', which can be used for debugging. May or may not terminate the running job.
3. **kill -KILL** : 'last-ditch', we use this as the very last resort. If we send this signal to the process, the process is killed by the operating system now and unconditionally. The process cannot clean up. The signal `KILL` always succeeds and terminates the running process, whatever are the consequences. If even this signal has failed, that means that the operating system has failed.

We remark that sending signals to the running job is not only about terminating its execution. For instance, we can resume programmatically the suspended job by sending it the signal `CONT` (number 18). That is illustrated with the following sequence:

```
sleep 44m &
[1] 12254
jobs -l
[1]+ 12254 Running                 sleep 44m &
kill -TSTP 12254
jobs -l
[1]+  Stopped                 sleep 44m
kill -CONT 12254
[1]+ 12254 Running                 sleep 44m &
```

In the next section we will see how we can further customize the signal catching.

At the end of this section, we stress out that, since the command **kill** can accept PID as an argument which is system-wide available, we can send signals to the jobs running in one terminal, by executing **kill** command with the specified signal in another terminal. In practice, if the running process has crashed and frozen the current terminal, we can still try to recover it by using its PID and sending to it signals with **kill** command from another terminal.

**Catching signals in your own code**

We have already seen how we can send the signal to the process, taking for granted that the implementation of that process has the relevant lines in the source code which can handle that particular signal. In this section, we clarify what is happening behind the scene when a process receives a signal.

We introduce and discuss first the commands which are used to handle programmatically the signal input. This can be achieved by using the **Bash** built-in command **trap**. In general, programs can be set up to trap specific signals, and interpret them in their own way. The command **trap** is used mostly for bullet-proofing, i.e. ensuring that your program behaves well under abnormal circumstances. The generic syntax of **trap** command is:

```
trap someCommand signal1 signal2 ...
```

The above generic syntax is interpreted as follows: When any of the signals `signal1`, `signal2`, `...`, is received, the following sequence follows:

1. pause the program execution and execute command **someCommand**
2. resume the program execution

After the execution of **someCommand** has terminated, the program execution resumes just after the command that was interrupted. In this context, **someCommand** can be also a script or a function. The signals `signal1`, `signal2`, `...`, can be specified either by signal name or by signal number.

The usage of **trap** is best illustrated with examples. We use the script named `trapExample.sh` with the following content:

```
#!/bin/bash

trap "echo Hi there!; echo How is life?" USR1
trap "pwd; ls" USR2

while :; do
 date
 sleep 10s
done

return 0
```

This script does nothing except that every 10 seconds prints the time stamp via **date** command. It is not possible to catch via **trap** the arbitrary user-defined signal, we have to use the standard 64 signals enlisted with **kill -l** (or **trap -l**). The closest we can get it to use `USR1` and `USR2` signals (numbers 10 and 12) as the standard supported signals reserved for the user's custom input.

We send this script to execute in the background via:

```
source trapExample.sh &
[1] 87
```

Every 10 seconds on the screen we get the timestamp printed, and in this simple example, that's the proof that our script is running. Now, while the script is running in the background, we start to communicate with our script by sending the signals to it:

```
kill -USR1 %1
```

The script pauses its execution, and responds to the signal `USR1` to produce the following output:

```
Hi there!
How is life?
```

Our signal was literally trapped by **trap** command, and whatever we have defined to correspond to the signal `USR1`, it will be executed. After that signal is processed, the script resumes normal execution, and we see every 10 seconds again on the screen the timestamp printed.

After sending the signal `ÙSR2`:

```
kill -USR2 %1
```

the script execution is paused again, this time two commands **pwd** and **ls** are executed, and the script resumes execution.

After sending these two signals, the script is still running in the background:

```
jobs -l
[1]+  87 Running                 source trapExample.sh &
```

Therefore, by using the **trap** command we can programmatically and on-the-fly modify the behaviour of the running program, without terminating its execution, changing something in the code, and restarting from scratch. Just like we have implemented traps for signals `USR1` and `USR2`, we can implement our own version of traps for the more standard signals like `ÌNT`, `TERM`, etc.

We conclude this section with a few additional remarks. The traps can be reset, by using the following generic syntax:

```
trap - someSignal
```

Signals sent to your script can be ignored by using the following syntax:

```
trap "" someSignal
```

For instance, if we want to prevent `Ctrl+C` (which is a shortcut for **kill -INT**) to terminate the script execution, at the beginning of the script we need to add:

```
trap "" INT
```

With the above implementation, whenever signal `INT` is received, the script will literally do nothing about it.

The only signal which cannot be trapped, and therefore in particular which cannot be ignored, is `KILL`. That explains why **kill -KILL** or **kill -9** will always and unconditionally terminate your running program.


# Lecture 8: Bash fancy features

**Last update:** 20200624

## Table of Contents



## 1. Subshells: `( ... )`

We have already seen how the code block `{ ... }` can be used in **Bash** to embed the specific code snippet inside, and allocate only to the execution of that code snippet separate I/O facilities from the rest of the code. Another way of achieving this functionality is to use *subshell* `( ... )`. Any code block, when embedded within the round braces `( ... )`, forms the subshell. Generically, a subshell is defined as:

```
(
 command-input-1
 command-input-2
 ...
 command-input-n
)
```

or completely equivalently, with the one-liner:

```
( command-input-1; command-input-2; ... ; command-input-n; )
```

Just like it was done for the code block `{ ... }` in the previous sections, we now provide an executive summary of the most important features of subshell `( ... )`:

1. it inherits the current environment and cannot modify it globally
2. has its own `1>` and `2>` streaming facilities
3. starts a separate process, its PID can be obtained programmatically from the built-in variable **BASHPID**

From the above list, it is clear that there are a lot of similarities between `{ ... }` and `( ... )`. Basically, there are only two important differences:

1. when compared to the parent shell, `( ... )` starts a new process, while `{ ... }` does not

2. both `( ... )` and `{ ... }` inherit the environment from the parent shell, but `{ ... }` can modify it globally while `( ... )` cannot

When it comes to *stdout* and *stderr* streams, there is no difference between `( ... )` and `{ ... }`. The subshell `( ... )` is usually less efficient than the code block `{ ... }`, because it runs a separate process. However, since it cannot modify the environment in which it is run globally, `( ... )` is safer. As a rule of thumb, `( ... )` shall be preferred over `{ ... }` unless the efficiency is concern.

Typically, the subshells are executed in the background, by using the following generic syntax:

```
( command-input-1; command-input-2; ... ; command-input-n; ) &
```

The advantage of running subshells in the background is that now by using the **wait** command we can decide whether the rest of the code in the script will or will not wait the subshell execution to terminate (just like for any other command running in the background). This is very handy because we can use in that subshell automatically the already initialized environment in the script, execute the subshell, get back the result and keep environment unmodified.

The use case of subshell is illustrated with the following simple example:

```
( echo "Subshell PID: ${BASHPID}"; date; date -q; sleep 10m; ) 1>output.log
2>error.log &
```

Completely equivalently, the above code snippet could have been implemented across multiple lines:

```
(
 echo "Subshell PID: ${BASHPID}"
 date
 date -q
 sleep 10m
) 1>output.log 2>error.log &
```

It's a matter of personal taste which of the two versions is used in practice, but from the **Bash** perspective, they are the same. After executing, we see now with **jobs -l** the following printout:

```
[2]+  7941 Running                 ( echo "Subshell PID: ${BASHPID}"; date; date
-q; sleep 10m ) > output.log 2> error.log &
```

This means that the whole composite code inside the subshell now behaves like any other command running in the background. Any *stdout* printout in the body of subshell from any command is redirected with `1>` in the file 'output.log', whose content is:

```
Subshell PID: 7941
Do 4. Jul 07:37:55 CEST 2019
```

On the other hand, any *stderr* stream within subshell body was redirected with `2>` in the separate file 'error.log':

```
date: invalid option -- 'q'
Try 'date --help' for more information.
```

We can suspend the subshell execution just as we did it for individual commands:

```
kill -TSTP 7941
jobs -l
# [2]+  Stopped                 ( echo "Subshell PID: ${BASHPID}"; date; date -
q; sleep 10m ) > output.log 2> error.log
```

and resume it:

```
kill -CONT 7941
jobs -l
# [2]+  7941 Running                 ( echo "Subshell PID: ${BASHPID}"; date;
date -q; sleep 10m ) > output.log 2> error.log &
```

In the same fashion, we can terminate only the subshell execution, without affecting the script execution from which the subshell was launched. This is true whether or not subshell is executed in the background. For instance, if we have the following schematic situation:

```
some code
( some massive computation in subshell )
the remaining code waiting subshell to terminate
```

If the code execution got stuck because of some massive computation in the subshell, instead of terminating the whole script and restarting , we can terminate differentially only the subshell from a separate terminal, by sending some of the signals `TSTP`, `INT`, `QUIT` or `KILL` directly to the PID corresponding to the subshell. This is not possible for code block `{ ... }` because its PID is the same as the PID of the parent shell.

## 2. Process substitution operator: `<( ... )`

Process substitution operator `<( ... )` is a feature which is not provided by all shells, therefore its usage typically leads to some portability problems across different shells. Nevertheless, since it is supported by **Bash** and since it comes very handy in some frequently encountered cases in practice, we introduce it next.

Loosely speaking, process substitution operator translates on the fly the standard output stream of any command into a 'virtual' file, which then can be fed to the commands which accept only files as arguments. The true mechanism behind the scene is much more complicated than that (it relies on so-called 'named pipes'), but this loose explanation captures its behaviour in practice quite well.

Within `<( ... )` operator we can execute as many commands as we wish, separated with `;` delimiter. Because of this, the process substitution operator offers more flexibility than pipe `|`, which can forward directly the *stdout* of only one command as an input to another command.

In we execute multiple commands within `<( ... )` operator, the output of each command is summed up in one large common 'virtual' file, which then we can feed for further processing to commands like **grep**, **awk**, etc. This is best illustrated with the examples.

**Example 1:** How to extract only the text in the 4th column from the output of 3 consecutive commands? By using the process substitution operator and **awk**, the solution is very simple and elegant:

```
awk '{print $4}' <(command1; command2; command3)
```

In order to solve this problem without using the `<( ... )` operator, we would need to use something like:

```
command1 > someFile
command2 >> someFile
command3 >> someFile
awk '{print $4}' someFile
rm someFile
```

Clearly, by using the process substitution operator `<( ... )` we have saved quite some unnecessary coding steps.

**Example 2:** How to parse line-by-line through the output of some command and manipulate each line programmatically?

We have already seen that we can with the **while+read** construct parse line-by-line through the content of some external file. By using the process substitution operator `<( ... )`, we can trivially extend that functionality to the output of some command. For instance, let us do some programmatic manipulation on the output of **ls -al** command:

```
while read; do
 echo "In the line $REPLY there are ${#REPLY} characters"
done < <(ls -al)
```

The output of this command could look like:

```
In the line -rw-r--r--  1 abilandz abilandz  9508 Jul  2 09:42 bash_logo.png
there are 64 characters
In the line -rw-rw-r--  1 abilandz abilandz  7652 Jul  2 10:51 colours.png there
are 62 characters
```

This works because from the perspective of **while+read** construct, redirection from external file via `< someFile` or via process substitution operator `< <( ... )` is completely equivalent, because the output of `<( ... )` is essentially a 'virtual file'.

**Example 3:** How to check programmatically if the printouts of two commands are exactly the same?

In order to compare the content of two files, we can use the **diff** command, with the following generic syntax:

```
diff file1 file2 && echo same || echo different
```

With the process substitution operator, we can extend the functionality of **diff** command to the comparison between the printout of different commands. The generic use case could look like:

```
diff <(command1) <(command2) && echo same || echo different
```

Without the process substitution operator, we would need to dump the printout of each command in some temporary file, and then compare the content of those temporary files with **diff** command. Clearly, process substitution operator saves also in this frequently encountered example a lot of additional and completely trivial coding.

## 3. Here strings: `<<<`

'Here strings' is a **Bash** feature which can save a bit of typesetting in some frequently encountered use cases. For instance, with 'here strings' we can feed the already made string directly into the command, without typing that string manually. Some commands (e.g. **bc**), we first need to start, and only then supply the string as an input interactively from the keyboard. Such cases can also be bypassed with 'here strings' and instead of typing directly from the keyboard, the commands can process the input string programmatically.

The generic syntax for the usage of 'here strings' can be represented in the following schematic way:

```
command <<< "some hardwired string"
```

In the above example, the input string is hardwired, but it can be also supplied as the content of variable:

```
var="some hardwired string"
command <<< ${var}
```

The command substitution operator `$( ... )` and arithmetic expansion `$(( ... ))` can be used as well on the right hand side of `<<<` operator. What **Bash** is doing when it encountered `<<<` can be summarized as follows: Whatever is on the right hand side of `<<<` undergoes expansion (e.g. `${var}` is replaced with variable content, `$(( ... ))` is replaced with the result of arithmetic evaluation, etc.), and then the resulting expression is simply fed to the 'stdin' of **command**. The result of expansion is supplied as a single string to **command**, with a newline always appended.

It is also possible to think about 'here strings' as being the shortcut notation for the following common construct:

```
echo "someString" | command
```

because the output of above line is the same as of:

```
command <<< "someString"
```

Therefore, instead of using the command **echo** in combination with pipe `|`, we can simply condense the syntax and use only `<<<`, the final printout is exactly the same.
**Example 1:** Let's define `var=20180524`. How to check programmatically with **grep** if this string begins with the pattern '2018'?

The two solutions below yield the same result:

```
echo ${var} | grep "^2018"
grep "^2018" <<< ${var}
```

'Here strings' are frequently combined with **sed**, that is illustrated in the next example.

**Example 2:** Let's define `Var=20180524`. How to change pattern '2018' into '2020' programmatically and immediately redefine the variable `Var` to the new content?

```
sed "s/2018/2020/" <<< $Var
echo ${Var}
# prints 20180524, content of 'Var' is unchanged
```

Note that the content of the starting variable `Var` is still '20180524', we only saw the pattern replacement in the printout on the screen. If we want to update immediately the content of the starting variable, this can be achieved with:

```
Var=$(sed "s/2018/2020/" <<< $Var)
echo ${Var}
# prints 20200524, content of 'Var' is changed
```

Finally, we illustrate the typical usage of 'here strings' in combination with **bc** command, to perform floating point arithmetic's. We start by recalling the example from Lecture 6.

**Example 3 (revised from Lecture 6):** How can we divide 10/7 at the precision of 30 significant digits? Solution was given previously by the following code snippet:

```
echo "scale=30; 10/7" | bc
```

However, this problem can be also solved a bit simpler with the usage of 'here strings':

```
bc <<< "scale=30; 10/7"
```

The output in both cases is the same and it reads:

```
1.428571428571428571428571428571
```

There are the cases in which **echo** + `|` is more efficient, while there are also the cases in which the 'here strings' `<<<` run faster, so both versions are used frequently in practice.

## 4. Here documents: `<<`

'Here documents' are typically used within shell scripts to write programmatically entire files, including the new scripts. Programmatically written files with 'here documents' can be used as a further input to the script which made them or as an input to some other commands. Programmatically written scripts with 'here documents' can be immediately executed in the script which created them.

There are three important use cases of 'here documents'. The first one uses the following generic syntax:

```
cat > someFile <<HERE-DOC
... some code here ...
HERE-DOC
```

The 'here document' begins with `<<` and its body is marked with the matching pair of delimiters. The name of the delimiters is arbitrary, as long as the closing one does not coincide with some string in the code in the body. The above construct evaluates all code inside delimiters named 'HERE-DOC', and dumps it in the external file named 'someFile'. There is nothing special about the choice of 'HERE-DOC' to name the delimiters, we could have uses as well something like

```
cat > someFile <<EOF
... some code here ...
EOF
```

The empty characters at the end of delimiters matter, and this is a typical source of errors. If the name of the opening delimiter is 'HERE-DOC' (no empty characters!) and the name of the closing delimiter is 'HERE_DOC ' (one empty character at the end!), we will get an error, as the two delimiters do not match exactly each other. Therefore, make sure there are no trailing empty characters after the delimiters.

With the above version of 'here documents', **Bash** supports variable and command substitution in the body of 'here documents', which enables to write files programmatically. This is best illustrated with the following simple script named 'testHD.sh', which takes as two arguments two floats, sums them up, and dumps everything (alongside with some other info) in the external file named 'output.log':

```
#!/bin/bash

[[ 2 -eq $# ]] || return 1

cat > output.log <<HERE-DOC
Today is: $(date)
File produced by script: ${BASH_SOURCE}
Sum is: $(bc <<< "scale=2; $1 + $2")
HERE-DOC

return 0
```

If we execute this script for instance with:

```
source testHD.sh 2.44 10.23
```

this script has by using the 'here document' created an external file named 'output.log', whose content is:

```
Today is: Tue Jun 16 14:04:52 CEST 2020
File produced by script: testHD.sh
Sum is: 12.67
```

Note that we did not need to use **echo** in the body of 'here documents' to dump any text into the file 'output.log'. On the other hand, if we would have used code block `{ ... }` to achieve the same results, the code inside the code block would be clogged with **echo** statements:

```
#!/bin/bash

[[ 2 -eq $# ]] || return 1

{
 echo Today is: $(date)
 echo File produced by script: ${BASH_SOURCE}
 echo Sum is: $(bc <<< "scale=2; $1 + $2")
} > output.log

return 0
```

Because of this difference, 'here documents' are simpler and more elegant when writing programmatically the files than the code blocks `{ ... }`.

The second important use of 'here documents' uses the following generic syntax:

```
cat > someFile.log <<'HERE-DOC'
... some code here ...
HERE-DOC
```

Note the use of quotes round the opening delimiter. In this version, the code in the body of 'here documents' is literally evaluated with the following phrase: What you typed is what you get. The meaning of all special symbols in the code in the body of 'here document' is killed if the starting delimiter is enclosed within quotes. For instance, if we modify the previous example:

```
#!/bin/bash

[[ 2 -eq $# ]] || return 1

cat > newScript.sh <<'HERE-DOC'
Today is: $(date)
File produced by script: ${BASH_SOURCE}
Sum is: $(bc <<< "scale=2; $1 + $2")
HERE-DOC

return 0
```

and execute it as before, the content of the file 'newScript.sh', to which we have redirected now the content of 'here document', is dramatically different:

```
Today is: $(date)
File produced by script: ${BASH_SOURCE}
Sum is: $(bc <<< "scale=2; $1 + $2")
```

This is perfectly valid **Bash** source code. Literally, what we type in the body of this version of 'here document' is what we get in the external file.

The above version of 'here documents' has an obvious and important use case: We can within one **Bash** script programmatically write (and execute immediately if necessary) another **Bash** script, with all special characters and **Bash** syntax in place. This feature is more general, because with this version of 'here documents' we can preserve the special syntax of any other programming language, dump the code in external file, compile it and use executable immediately in the very same **Bash** script in which you have written the initial source code.

Finally, 'here documents' can be used to comment out multiple lines of **Bash** source code in one go, instead of using `#` at the beginning of each line. This is illustrated with the following example:

```
... code line 1 ...
... code line 2 ...
... code line 3 ...
... code line 4 ...
... code line 5 ...
... code line 6 ...
... code line 7 ...
... code line 8 ...
... code line 9 ...
```

How for instance we would comment out the code in the lines 2 to 8, and execute only the code in the lines 1 and 9? This can be achieved in the following way:

```
... code line 1 ...
: <<'HERE-DOC'
... code line 2 ...
... code line 3 ...
... code line 4 ...
... code line 5 ...
... code line 6 ...
... code line 7 ...
... code line 8 ...
HERE-DOC
... code line 9 ...
```

We have essentially declared lines from 2 to 8 to be the body of 'here document', and then its content redirected to the infamous 'do-nothing' command `:`. This is yet another very neat use case of 'do-nothing' command! When using 'here documents' to comment out piece of the code, the version in which the opening delimiter is enclosed in quotes is safer, as it will prevent any potential variable or command substitution in the body, before passing the code over to 'do-nothing' command.

As this is the frequent source of painful debugging, we close this section by remarking again that the closing delimiter in 'here documents' shall not be followed by any trailing empty character.

## 5. Menus

Menus are built in **Bash** programmatically with the built-in command **select**, with the following generic syntax:

```
select someVariable in someList; do
 ... some commands ...
 break
done
```

The menu for selection is specified via the list, schematically indicated as 'someList', which is placed between the key word **in** and semicolon `;`. The list can be specified by hardwiring some entries separated with empty characters, by using the command substitution operator `$( ... )`, brace expansion mechanism, etc. If the the key word **break** is omitted, menu is offered again

and again for selection.

We illustrate now with a few concrete examples how to build menus in **Bash**. We start by saving the following code in the file 'selectCountry.sh'

```
#!/bin/bash

select Country in Germany Italy France Spain England; do
 echo "You have selected: $Country"
 break
done

return 0
```

After executing the script via:

```
source selectCountry.sh
```

the following menu appears on the screen:

```
1) Germany
2) Italy
3) France
4) Spain
5) England
#?
```

**Bash** has offered us with the predefined menu, and is now waiting for reply. If we for instance type `1` and press `Enter` we get as a result the following printout:

```
You have selected: Germany
```

Since we have used the key word **break**, the **select** command bails out immediately after the first selection in the menu was done.

As another example, we can build menu from the output of some command:

```
#!/bin/bash

select File in $(ls *.sh); do
 echo "You have selected script: $File"
 source $File
 break
done

return 0
```

The above code offers on the menu all scripts in the current working directory, and then we can directly execute the script from menu, literally with one key stroke, instead of typing **source scriptName**.

The default prompt symbol in **select** is `#?`, while the default prompt symbol in **Bash** is `$`. How the prompt will look like is determined from the content of special environment variables **PS1** (for **Bash**), and **PS3** (for **select**). For instance, we can re-execute the previous example with:

```
PS3="What is your choice? " source selectCountry.sh
```

The menu is now:

```
1) Germany
2) Italy
3) France
4) Spain
5) England
What is your choice?
```

As a concluding remark, we indicate that if the key word **in** is omitted and the list is not specified, the list is defaulted to the positional parameters supplied to the script or function, and menu is built out of all supplied positional parameters.


# Lecture 9: Real-life examples

**Last update**: 20200624

## Table of Contents



## 1. Command history search

When working directly in the terminal we want to speed up the command input as much as possible. Besides, frequently we want to be able to re-use the certain command input again, without re-typing it from scratch. A lot of typing is saved by using `TAB`, which autocompletes the command input to existing commands names (here commands are meant in the broader sense and include also **Bash** functions, aliases, etc.). In the case command is expecting as argument a file or directory, `TAB` will also autocomplete file or directory name, after we have typed in the first few characters and hit `TAB`. Last but not least, `TAB` also autocompletes the **Bash** variable names when we are referencing their content. This is illustrated with the following three simple examples:

```
dirn + TAB
# autocompletes to command 'dirname'

cat fileW + TAB
# autocompletes to file named 'fileWithLengthyName.log'

LongNameVariable=44
echo $Lo + TAB
# autocompletes to 'echo $LongNameVariable'
```

In general, when there are multiple matches for the initial pattern, after pressing once the `TAB` nothing happens, however pressing two times `TAB TAB` lists all matches:

```
ls Lecture_9 + TAB + TAB
# prints Lecture_9.md Lecture_9.html Lecture_9.pdf
```

The precedence of text completion via `TAB` can be summarized as follows: commands and functions first, then files and directories. If we want to alter these default precedence rules in some special cases of interest, we can use the following three special characters:

- if the text is preceded with `$` : variable completion with `TAB` takes precedence
- if the text is preceded with `~` : username completion with `TAB` takes precedence

The autocompletion via `TAB` is a very neat feature and speeds up a lot the typing, but it cannot help us to re-use what we have already typed.

To achieve that, we need to use **Bash** built-in command **history**. After we type in the terminal

```
history
```

all the command input which we have typed in the terminal recently (not necessarily only in the current terminal!) will be printed and enumerated by **Bash**. For instance, the output could be:

```
  525  ls
  526  nano readMe.txt
  527  ls
  528  tar -czf Lecture_8.tar.gz Lecture_8/
  529  ls
  530  cd ..
  531  ls
  532  typora Homework_7.md &
  533  cd ../Lecture_9
```

From where **Bash** has retrieved this detailed information of what we typed of late in the terminal? All previously typed commands are stored by default in the file to which the environment variable **HISTFILE** is pointing to:

```
echo $HISTFILE
```

The printout could look like:

```
/home/abilandz/.bash_history
```

That means that, by default, the history of all our command input is saved in the file `.bash_history` placed in the home directory. By default, at maximum 1000 lines of command input are kept in this file, but that can be changed by modifying the **Bash** environment variable **HISTSIZE**. If we now have a look at the content of the **Bash** command history file:

```
cat /home/abilandz/.bash_history
```

we see that we get a similar printout like the one from the command **history** showed above. The printout is similar, but not the same, and we will now clarify this difference, which sometimes leads to big confusion.

Each time we start a new terminal, the file `~/.bash_history` is read. From that point onward, each terminal maintains its own history (i.e. its own list of all commands we have typed in the terminal). When we exit the terminal, **Bash** updates the `~/.bash_history` file with the history which corresponds to that terminal. Therefore, and very importantly, the current content of `~/.bash_history` will correspond to the last terminal we have closed.

Some frequently used flags for the command **history** and their meanings are summarized here:

- `-c` : clears the history list (but it doesn't clean the content of `~/.bash_history` instantly, remember that this file gets updated automatically only after we exit the terminal!)
- `-d someNumber` : clears the **history** entry only at the line 'someNumber'
- `-a` : forces appending history lines from the current terminal to the history file `~/.bash_history`. With this option, we save permanently all commands we have typed in the history file, even without exiting the terminal

After understanding the **history** mechanism, we now demonstrate how we can directly extract only the entry we need with a few convenient shortcuts:

- Use up and down arrow (or equivalently `Ctrl+N` and `Ctrl+P`) in the terminal to browse through (in the specified order!):
  - terminal's own history
  - the content of `~/.bash_history`
- `Ctrl+R` : inverse history search. After we press `Ctrl+R`, the following prompt appears:

  ```
  (reverse-i-search)`':
  ```

  Now we can type the pattern which will be used to search for some previously used command input that contained that pattern. We can keep pressing `Ctrl+R`, until the command input we are looking for appears. By pressing the right arrow, that command input is copied in the terminal, and we can now reuse it again.

**Example:** The inverse history search is an extremely handy feature, and we now illustrate it with the concrete example. Imagine a scenario in which we have typed in the terminal for** loop, followed by a lot of other commands:

```
for i in {1..10}; do echo $i; done
... one zillion other commands ...
```

Do we need to retype the whole **for** loop from scratch, in case we need that command input again? It suffices only to do the following:

```
Ctrl+R
(reverse-i-search)`': # type the pattern'fo'
(reverse-i-search)`fo': for i in {1..10}; do echo $i; done
```

Press the right arrow, and the offered result from the inverse history search is copied in the terminal, and can be reused. If the offered result from the inverse history search is not what we need, we can keep pressing `Ctrl+R` to browse through all results which match the specified pattern.

We indicate now how we can directly re-execute any command input from the history list. For instance, if the command

```
history
```

has produced the following output

```
  188  ls
  189  grep Ctrl Lecture_?/Lect*
  190  for i in {1..10}; do echo $i; done
  191  ls
  192  pwd
```

we can execute any command from above just by typing `!commandNumberInTheList`. Given the above output, the following input in the terminal:

```
!190
```

gives immediately

```
for i in {1..10}; do echo $i; done
1
2
3
4
5
6
7
8
9
10
```

For more elaborate cases of retrieving and even editing the command input from **history**, please see the documentation of the command **fc** ('fix command').

Finally, we remark that programmatically we can retrieve the last argument of the previously executed command. This functionality is achieved via the special Bash variable `$_`. If the previously executed command has only one argument, then the content of `$_` is that argument. For instance:

```
mkdir Dir1 Dir2 Dir3
echo $_
# prints Dir3
```

A frequent use case is the following example:

```
ls file_{1..99}.log
rm $_
```

If the first line has expanded in the list of files we want to delete, we can reuse the same brace expansion n the second line as the argument for **rm** command.

**Example:** How to make a few directories, and automatically move into the last one created?

```
mkdir Dir1 Dir2 Dir3 && cd $_
```

## 2. Searching for files and directories: find

We have already seen how we can list the content of the specified directory with known location in the filesystem with **ls** command. However, in case we need to search for specific files or directories at unknown location in the filesystem hierarchy, **ls** command cannot be used. Instead, we can use the **Linux** command **find** is which was designed precisely for that sake. This powerful command can perform search by name, by creation, accession and modification date, by owner and permissions etc. In addition, **find** can immediately perform some action on the result of its search (for instance, it can immediately delete all files it has found, rename all directories, etc.).

The generic usage of command **find** can be described as follows:

```
find Where What Action
```

When interpreting its arguments, **find** defaults the first arguments without `-` or `--` as a list of directories in which the search will be performed. Therefore, in the above generic syntax 'Where' stands for one or more directories. After that, **find** expects one or more options starting with `-` or `--` , which will typically nail down what it needs to search for ('What' in the above syntax). Finally, there exists a special option **-exec** after which we can optionally set the commands which **find** will execute immediately on the results it has found ('Action').

The usage of **find** is best illustrated on concrete examples. Let us start with a directory named 'Examples' in which we have the following situation:

```
$ ls Examples/
Directory_0  Directory_1  Directory_2  Directory_3  file_0.dat  file_0.pdf
file_0.png  file_1.dat  file_1.pdf  file_1.png
```

**Example 1:** Find and print on the screen only the files in the specified directory.

```
find Examples/ -type f
```

The result is:

```
Examples/file_0.png
Examples/file_1.dat
Examples/file_1.pdf
Examples/file_1.png
Examples/file_0.dat
Examples/file_0.pdf
```

By default, the result of **find** is not sorted, but you can trivially, and in general, sort the output of some command by piping to the command named **sort**:

```
find Examples/ -type f | sort
```

The output is now sorted:

```
Examples/file_0.dat
Examples/file_0.pdf
Examples/file_0.png
Examples/file_1.dat
Examples/file_1.pdf
Examples/file_1.png
```

**Example 2:** Find all subdirectories in the specified directory.

```
find Examples/ -type d
```

The result is:

```
Examples/Directory_3
Examples/Directory_2
Examples/Directory_0
Examples/Directory_1
```

**Example 3:** Find all files with an extension '.pdf' in the specified directory

```
find Examples/ -type f -name "*.pdf"
```

The result is:

```
Examples/file_1.pdf
Examples/file_0.pdf
```

**Example 4:** Find all files with an extension '.pdf' and pattern 'file_0' in their name, in the specified directory.

```
find Examples/ -type f -name "*.pdf" -a -name "*file_0*"
```

The result is:

```
Examples/file_0.pdf
```

From the above example, we see that **find** interprets the flag `-a` as the logical `AND`. Similarly, the flag `-o` can be used within **find** as the logical `OR`.

Since the flag `-name` is very frequently used, it deserves some additional clarification. The usage of quotes in the pattern, as in `"*.pdf"`, was essential, because now the special characters will be supplied as the special characters to the **find** command, and will prevent **Bash** to expand them. Dropping quotes round the pattern is a typical mistake when **find** is used:

```
find Examples/ -type f -name *.pdf # WRONG!!
```

The above syntax is wrong, because **Bash** now will first expand the pattern `*.pdf` to match all files in the current working directory (not in the directory `Examples` !) that end with `.pdf`, and only then those fully expanded file names will be supplied to the command **find**. Clearly, this will work only by accident if in the current working directory where we have executed the command **find** there was no a single file which ends in `.pdf`, and therefore `*.pdf` remained unexpanded.

Alternatively, the special symbols can be supplied to **find** with the escaping mechanism `\`. Summarizing everything:

```
find Examples/ -type f -name "*.pdf" # CORRECT
find Examples/ -type f -name '*.pdf' # CORRECT
find Examples/ -type f -name \*.pdf # CORRECT
find Examples/ -type f -name *.pdf # WRONG!!
```

**Example 5:** Find all files with an extension '.pdf' larger than 10 KB in the specified directory.

```
find pathToDirectory(-ies) -type f -name "*.pdf" -size +10k
```

Here prefix `+` is not trivial, if we would omit it, the flag `-size 10k` would filter out instead the files whose size is exactly 10 KB. Unfortunately, syntax for KB in **find** is a small 'k', and not capital 'K' (like in **ls -lh**), which frequently leads to confusion. Analogously, files which are smaller than 10 KB in size, we would filter out by using prefix `-`, i.e. `-size -10k`.

**Example 6:** Find all files with an extension '.tex' modified within last 10 days.

```
find pathToDirectory(-ies) -type f -name "*.tex" -mtime -10
```

Similarly as with the option `-size`, the option `-mtime +10` means more than 10 days ago, `-mtime 10` exactly 10 days ago, and `-mtime -10` less than 10 days ago. Closely related flags are `-atime` and `-ctime`. The flag `-atime` traces when the files were last accessed (i.e. read without being modified, for instance using **cat** command), while the flag `-ctime` traces when the file's metadata (permissions, name, location, etc.) were last time changed.

**Example 7:** Find all obsolete files in the specified directory(-ies) which were not accessed for more than 1 year.

```
find pathToDirectory(-ies) -type f -atime +365
```

The three frequently used flags `-mtime`, `-ctime` and `-atime` have the lowest resolution of 1 day. To perform the search with even finer time resolution in minutes, we need to use the flags `-mmin`, `-cmin` and `-amin`.

By default, the command **find** searches through all subdirectories of specified directory. If we start the search in some top level directory in the file hierarchy, the search can take forever. If we are sure that the targeted files are not deeper in the directory structure than a certain level, we can use flags `-maxdepth` and `-mindepth` to greatly optimize the search.

**Example 8:** Find all files with an extension '.pdf' in the specified directory, not going deeper than 2 levels in the subdirectory structure.

```
find pathToDirectory(-ies) -maxdepth 2 -type f -name "*.pdf"
```

**Example 9:** Find all files with an extension '.pdf' in the specified directory, by looking only in the subdirectories (i.e. not in the current directory, and not in subsubdirectories, subsubsubdirectories, etc.). The solution is:

```
find pathToDirectory(-ies) -mindepth 2 -maxdepth 2 -type f -name "*.pdf"
```

Finally, and very importantly, we describe the flag `-exec` which is used to specify the 'Action' part in the previously mentioned generic syntax, i.e. the command input which **find** needs to execute on the spot on the search outcome. The syntax for flag `-exec` is a bit peculiar, but there are essentially two important things to remember:

- `\;` : the command input after the flag `-exec` is determined this way
- `{}` : when used in combination with `-exec`, this is a placeholder for the found file or directory

**Example 10:** Find all empty files in the specified directories and for each of them prints its size.

```
find pathToDirectory(-ies) -type f -exec stat -c %s {} \;
```

From this example it is self-evident when and how we use the placeholder `{}` for the found file or directory in combination with `-exec` flag.

**Example 11:** Find all empty files in the specified directory, and delete them immediately:

```
find pathToDirectory(-ies) -type f -size 0 -exec rm {} \;
```

It is also possible to execute multiple commands on the files or directories which **find** has found, we just need to use separate `-exec` flag for each command input:

**Example 12:** Find all files in the specified directory, and for each of them: a) print the full metadata with **ls -al**; and b) print the size with **stat -c %s**.

```
find pathToDirectory(-ies) -type f -exec ls -al {} \; -exec stat -c %s {} \;
```

Equivalently we can use **while+read** construct in combination with the process substitution operator `<( ... )` to achieve the same result:

```
while read File; do
 ls -al $File
 stat -c %s $File
done < <(find pathToDirectory(-ies) -type f)
```

The second solution is more readable, less error prone and easier to generalize by adding more actions to the found files.

## 3. Online monitoring: tail -f

In general, we can view the whole content of the file with the **cat** command, or if we want paging to appear one screen at a time we can use commands like **more** or **less**. On the other hand, we can select and view only the select part of the file with commands like **sed**. For instance, if the starting file named 'example.txt' has the following content:

```
line 1
line 2
line 3
line 4
line 5
line 6
line 7
```

we have already seen in previous sections that we can select with **sed** for viewing only the lines in the specified range (both ends included), like in the following example:

```
sed -n 2,4p example.txt
```

Flag `-n` ensures that the starting file is not superimposed with the desired selected printout, while `p` stands for 'print'. The result is:

```
line 2
line 3
line 4
```

Alternatively, if we are interested to print only the first 'n' lines of the file, we can use **head -n** command. For instance:

```
head -3 example.txt
```

will print only the first 3 lines:

```
line 1
line 2
line 3
```

On the other hand, if we are interested to print only last 'n' lines of the file, we can use **tail -n** command. For instance, to get programmatically only the last line in the file, we can use:

```
tail -1 example.txt
```

which gives for the above example:

```
line 7
```

Without arguments, **head** and **tail** print by default the first and the last 10 lines, respectively.

Besides these simple use cases, the important non-trivial use case is provided with the flag **-f** of **tail** command. Namely, with **tail -f** we can monitor online the output of file as the file content gets updated. That means that if we have redirected the `stdout` stream of some command to a file, and if we execute **tail -f** on that file, we will monitor what that command is doing just as we are looking at its printout on the screen. However, what is non-trivial here is that we can execute **tail -f** on that file from any terminal and monitor online what that command is doing, not necessarily from the same terminal where the command is started. This is best illustrated with the following example:

```
( echo ${BASHPID}; while :; do echo "1: $(date)"; sleep 10s; done; ) 1>first.log
&
( echo ${BASHPID}; while :; do echo "2: $(date)"; sleep 20s; done; )
1>second.log &
( echo ${BASHPID}; while :; do echo "3: $(date)"; sleep 30s; done; ) 1>third.log
&
```

With this example, we have started three subshells in the background, where each of them after 10s, 20s and 30s, respectively, prints the time stamp, which is redirected via `1>` in its own log file. Since all 3 subshells are running in the background, we have the control over the terminal, and we can for instance checkout the status of submitted jobs:

```
jobs -l
[1]  20860 Running                 ( echo ${BASHPID}; while :; do echo "1:
$(date)"; sleep 10s; done ) > first.log &
[2]- 20867 Running                 ( echo ${BASHPID}; while :; do echo "2:
$(date)"; sleep 20s; done ) > second.log &
[3]+ 20873 Running                 ( echo ${BASHPID}; while :; do echo "3:
$(date)"; sleep 30s; done ) > third.log &
```

The great thing now is that we can see directly in the terminal what each of these jobs is doing in the background. For instance:

```
tail -f first.log
```

This gives:

```
20860
1: Tue Jun 23 12:39:57 CEST 2020
1: Tue Jun 23 12:40:07 CEST 2020
1: Tue Jun 23 12:40:17 CEST 2020
1: Tue Jun 23 12:40:27 CEST 2020
1: Tue Jun 23 12:40:37 CEST 2020
1: Tue Jun 23 12:40:47 CEST 2020
```

As the subshell execution proceeds, the output of **tail -f** gets updated on the screen automatically, just like the subshell is directly running in the terminal, and not in the background. If we now hit `Ctrl-C`, we terminate only the **tail -f** command, without any interference with the running subshell in the background. After terminating with `Ctrl-C`, we can inspect the status of second subshell running in the background by executing:

```
tail -f second.log
```

This gives:

```
20867
2: Tue Jun 23 12:40:02 CEST 2020
2: Tue Jun 23 12:40:22 CEST 2020
2: Tue Jun 23 12:40:42 CEST 2020
2: Tue Jun 23 12:41:02 CEST 2020
```

We now monitor online what the second subshell running in the background is doing. This functionality works from any terminal, since viewing the content of physical files with **tail -f** is not limited to any particular terminal.

## 4. Timing: timeout and time

Frequently in practice we are faced with the situation when the command execution gets stalled, without clear indication when its execution might resume. For instance, if we are copying files over the network, and if the network connection experiences a problem, the copying itself will hang until the network connection recovers. But for instance if are copying over network 1000 files containing our data, and if we managed to copy 90% of them, clearly we can reach the decent statistics and reliable results in our analysis, even if we did not analyse the whole dataset.

In general, we can prevent command to hang forever with the **timeout** command. This command in essence ensures that some command is run within a specified time limit. Its generic syntax is:

```
timeout someInterval someCommand
```

This syntax translates into the following use case: Start a command named 'someCommand', and terminate it if it still runs after the specified time interval 'someInterval'.

Again, we use command **sleep** to illustrate the use cases of **timeout** in concrete examples, but the examples below apply to any other command:

```
timeout 5s sleep 10s
```

This will terminate **sleep** command already after 5s, with non-zero exit status 124 (check the 'man' pages of **timeout**). The exit status is non-zero, since command failed to complete its execution within the specified time interval. By default, **timeout** terminates the command execution with `TERM` signal, but we can send any other supported signal (check out the list with **kill -l**) by using **-s** flag, for instance:

```
timeout -s KILL 5s sleep 10s
```

In the case command fails by itself within the specified time interval, then the exit status of **timeout** is the exit status of command.

Finally, in the case of successful command completion within the specified time interval, e.g.

```
timeout 20s sleep 10s
```

the exit status of **timeout** is 0.

As the last remark, we indicate that unfortunately command **timeout** can deal only with **Linux** commands, and not for instance with **Bash** functions.

In a completely different context, we use expression 'timing' when we want to summarize the usage of system resources by a given command. This is simply achieved with the command **time**. Its generic syntax for most cases of interest is very simple:

```
time someCommand
```

Here 'someCommand' is meant in a broader sense, and can be any **Linux** command, **Bash** built-in command, **Bash** function, etc. For instance:

```
time sleep 4s
```

gives the following expected printout:

```
real    0m4.002s
user    0m0.000s
sys 0m0.002s
```

However, we can use also **time** for **Bash** code snippets directly:

```
time for i in {1..1000}; do date; done > /dev/null
```

produces:

```
real    0m1.499s
user    0m0.165s
sys 0m1.397s
```

This is clearly a great utility when the efficiency of code execution starts to matter.

## 5. Counting: wc

The number of different elements (lines, words, characters) in the file content, or in the `stdout` of some command, can be conveniently obtained with the command **wc** ('word count').

For instance:

```
echo "a bb ccc" | wc
```

provides the following output:

```
      1       3       9
```

The first entry is the number of lines (1), then the number of words (3, namely "a", "bb" and "ccc"), and finally the number of characters (9, since both the empty characters and hidden new lines also count!). Using **wc** to count the number of characters frequently leads to surprises and is not recommended because the hidden new line character '\n' at the end of each line is also counted, for instance:

```
echo | wc
```

prints

```
       1       0       1
```

In the above printout, 1 character corresponds to the default new line character '\n' in **echo**.

For the counting of the number of lines and words this command behaves as expected. Typically, we just want number of lines or number of words, when flags **-l** or **-w** can be used.

The command **wc** can also count the elements of the physical file. For instance, if the starting file 'wcExample.txt' has the following content

```
line 1 a
line 2 bb b
line 3 c ccc cc
```

then we can use the following syntax:

```
wc -l < wcExample.txt
# prints 3, total number of lines
wc -l < wcExample.txt
# prints 12, total number of words
```

## 6. Building programmatically command input: eval

We have already seen how we can programmatically provide the arguments to the commands by referencing the content of some variables with the general syntax `${var}`. Now we generalize this idea and illustrate how we can build the whole command input programmatically, including even the pipes. This can be achieved by using the **Bash** built-in command **eval**. Some experts warn against its usage due to potential security holes and argue that this command shall be instead renamed into 'evil'. Typically, the command **eval** can be used to force additional re-evaluation of command input, if its interpretation ended up in some intermediate state.

In essence, the command **eval** enforces the command-line processing once again. It's a very powerful feature, which enables to write scripts that create command string on-the-fly and then pass it to **Bash** for execution. By using this mechanism, the **Bash** scripts can for instance modify their behaviour when they are already running.

We start with the following example, where from the **date** command we extract only the hour, minutes and seconds:

```
date | awk '{print $4}'
# prints 12:22:02
```

But now let us attempt in the **Bash** variable **DateSimple** to store that command input:

```
DateSimple="date | awk '{print $4}'"
```

If we now attempt to reference the content of the variable **DateSimple**, and use it directly in the same way as command input, we get an error:

```
$DateSimple
# date: extra operand ‘awk’
# Try 'date --help' for more information.
```

However, this saves the day:

```
eval $DateSimple
```

What happened above is the following: Upon expanding the content of variable **DateSimple**, **Bash** interpreted pipe `|` and `awk` as arguments to **date** command, and since the command **date** can handle only one argument (besides flags which are indicated with prepended `-` or `--`), it bailed out when it hit at the second argument, which is string `awk`. When interpreting the command input, one of the very first thing **Bash** is looking for are pipes `|`, however, since in the literal command input **$DateSimple** there are no pipes (before expansion!), **Bash** stopped searching for them. Then after expanding **$DateSimple**, **Bash** continued with the other steps in the command input interpretation, none of which includes pipes. Therefore, the pipe `|` ended up being interpreted as a mere argument to **date** command. This sort of problems can be fixed with **eval**, because that commands literally forces the command input re-interpretation from scratch. After using **eval $DateSimple**, and after expanding **$DateSimple**, **Bash** goes from scratch through the command input interpretation, and interprets the pipe `|` correctly.

To a certain degree, echoing the command input into the file, and then sourcing that file, achieves the same functionality as **eval**, but it is much less efficient.

Finally, we mention the following frequently encountered example. We can generate sequences with brace expansion, for instance:

```
echo {0..9}
```

prints

```
0 1 2 3 4 5 6 7 8 9
```

But what if we want to pass low and upper ends via variables? We could try:

```
Min=0
Max=9
echo {$Min..$Max}
```

The result is however only the following printout:

```
{0..9}
```

This is a nice example where **eval** saves the day again, because

```
eval echo {$Min..$Max}
```

forces the re-interpretation of intermediate result `{0..9}`, and produces the desired output:

```
0 1 2 3 4 5 6 7 8 9
```