Bash Lecture 1 - Basics

Why this lecture series

★To know UNIX/Linux command line

★To gain ability in command line usage

★To introduce scripting (in bash)

★ To have some basic programming

★To introduce python programming

★To introduce useful python libraries

Bibliography and learning materials

★ Bibliography:

https://www.rigacci.org/docs/biblio/online/sysadmin/toc.htm

https://www.tldp.org/LDP/abs/html/

★ Learning Materials:

http://www.ee.surrey.ac.uk/Teaching/Unix/

https://github.com/gtaffoni/Learn-Python/blob/master/Lectures/ShellLecture01.pdf

https://github.com/gtaffoni/Learn-Python/blob/master/Lectures/ShellLecture02.pdf

https://github.com/bertocco/alfabetizzazione_bash

Arguments of this lesson

- ★ How the shell works with you and linux
- ★ Features of a shell
- ★ Manipulating the shell environment

What is a shell?

★ SHELL is the human interface point for UNIX

SHELL is a program layer that provides an environment to enter commands and parameters to produce a given result.

To meet varying needs, UNIX has provided different shells. They differ in the various options they give the user to manipulate the commands and in the complexity and capabilities of the scripting language.

- Bourne (sh)
- Bourne Again (bash)
- Korn (ksh)
- C shells (csh)
- TC-Shel (tcsh)
- Z shell (zsh)

Why bash?

- ★Flexible
- ★ More friendly than others
- ★The default in the most part of linux distributions

UNIX commands (1)

★ General form of a command:

command [flags] [argument1] [argument2] …

Example:

`ls -a -l` *or* `ls -al`

★ Arguments can be optional or mandatory

★ All commands have a return code (0 if OK)

Read return code: `echo $?`

The return codes can be used as part of control logic in shell scripts

★ All UNIX commands have an help:

`man command` or `man <number> command`

UNIX commands (2)

★ All commands:

- accept inputs from the standard input,

  is where UNIX gets the input for a command

- display output on standard output

  is where UNIX displays output from a command

- display error message on standard error

  is where UNIX displays any errors as a result of the execution of a command

★ UNIX has redirection capabilities: to redirect one or more of these (see advanced & scripting lesson)

Exercises

★ Verify that you are using bash:

```
echo $SHELL
```

★Explore help command

type `help`

★ Explore help for `ls` command

type `ls -h`

★List files `ls` or `ls -l` and check differences

★ List all files `ls -al`

★List files by date (direct and reverse order) `ls -trl`

Command `echo` and strings

★When one or more strings are provided as arguments, echo by default repeats those stings on the screen.

Example (try)

echo This is a pen.

It is not necessary to surround the strings with quotes, as it does not affect what is written on the screen. If quotes (either single or double) are used, they are not repeated on the screen (try `echo “This is a pen.”`).

★`echo` can also show the value of a particular variable if the name of the variable is preceded directly (i.e., with no intervening spaces) by the dollar character ($), which tells the shell to substitute the value of the variable for its name. Example (try):

x=5; echo The number is $x.

Command `echo` examples

★echo This is a pen.

★x=5
echo The number is $x.
echo “The number is $x.”

★Simple backup script
OF=/home/me/my-backup-$(date +%Y%m%d).tgz
tar -cZf $OF /home/me/
ls -l (to check the result)

`export`

`export` exports environment variables (also to children of the current process). Example:

ubuntu~$ export a=test_env
ubuntu:~$ echo $a
test_env
ubuntu:~$ /bin/bash
ubuntu:~$ echo $a
test_env
ubuntu:~$ exit
exit
ubuntu:~$ echo $a
test_env

`export` called with no arguments prints all of the variables in the shell's environment.

`unset` frees variables

User related commands: passwd

★`passwd` changes user's password

Example: type `passwd`

```
$ passwd
Changing password for bertocco.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
Sorry, passwords do not match
passwd: Authentication token manipulation error
passwd: password unchanged
$ passwd
Changing password for bertocco.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

User related commands: who and whoami

★`who` show who is logged on

Print information about users who are currently logged in.

★`whoami`

Print the user name associated with the current effective user ID.

Exercise:

try the commands and then type `man who`

and try some option

File manipulation commands (1)

It exists a set of file manipulation commands to manage files and directories.

To use these commands, the user needs to have right on the file to manage.

drwxrwxr-x 2 bertocco bertocco 20480 Dec 14 09:00 BACKUP

owner group size last_access_date file-name

drwxrwxr-x permissions representation:

d means it is a directory (- for a file)

rwx means readable, writable, executable by owner

rwx readable, writable, executable by group

r-x readable, NOT writable, executable by

File Permissions

Understand the meaning of:

```
drwxrwxr-x   2 bertocco bertocco      4096 Apr 26  2018 config
-rw-rw-r--   1 bertocco bertocco     10240 Mar 13  2017 config.tar
-rw-------   1 bertocco bertocco 960065536 Dec  3 22:02 core.3040
-rw-rw-r--   1 bertocco bertocco   7290880 May  8  2017 demo_EGIconf.tar
drwxr-xr-x.  4 bertocco bertocco      4096 Dec  7 15:57 Desktop
drwx------. 12 bertocco bertocco      4096 Aug 13 19:01 dev
drwxr-xr-x. 14 bertocco bertocco      4096 Nov 28 17:18 Documents
drwxr-----. 13 bertocco bertocco      8192 Dec 10 12:35 Downloads
drwxrwxr-x   2 bertocco bertocco       147 Apr 24  2018 exchange
-rw-r--r--   1 bertocco bertocco       181 Apr 13  2017 filmatini_utili.txt
```

Main file manipulation commands

- `touch` creates a file
- `mkdir  mydir` creates a directory (where you are)
- `mkdir -p /*onedir*/twodir/threedir`
- `rmdir mydir` delete an empty directory
- `rmdir -f` force to delete a non empty directory (BE CAREFUL!)
- `rmdir -rf` force to recursively delete a non empty directory BE VERY CAREFUL!!!
- `cp file1 file2` copy file1 on file2 (overwriting it if already exists, creating file2 if it does not exist)
- `cp -i file1 file2` before copy asks “are you sure?”
- `\cp file1 file2` remove the -i flag if set
- `rm file1` removes file1
- `mv file1 file2` moves file1 on file2 (it is the same of
- `cp file1 file2`; `rm file1`

File manipulation commands: Exercises

- Create a file
- Create a directory
- Create a directory tree
- Create files in the directory tree
- Remove a file
- remove a directory (emprty and not empty)
- remove a directory tree (emprty and not empty)
- Rename a file

`ls` (1)

`ls` can be used to inquire about the various attributes of one or more files or directories.

You must have read permission to a directory to be able to use the ls command on that directory and the files under that directory.

The `ls` command generates the output to standard output, which can be redirected, using the UNIX redirection operator >, to a file.

`ls` (2)

You can provide the names of one or more filenames or directories to the ls command. The file and directory names are optional. If you do not provide them, UNIX processes the current directory.

Be default, the list of files within a directory is sorted by filename. You can modify the sort order by using some of the flags.

You should also be aware that the files starting with . (period) will not be processed unless you use the -a flag with the ls command. This means that the entries . (single period) and .. (two consecutive periods) will not be processed by default.

`ls`: Exercices

- Try and understand differences:

ls; ls -l; ls -al

- Try and understand differences:

ls -trl; ls -tl

- Try an output redirection:

ls -l > myfileout.txt

`cat`

`cat` is used to display a text file or to concatenate multiple files into a single file.

By default, the cat command generates outputs into the standard output and accepts input from standard input.

The cat command takes in one or more filenames as its arguments. The files are concatenated in the order they appear in the argument list.

`cat`: Exercices

- Display file on a terminal

  cat testfile

- concatenate multiple files for display on the terminal

  cat testfile1 testfile2 testfile3

- concatenate these files into a file called testfile, use the redirection operator > as follows:

  cat testfile1 testfile2 testfile2 > testfile

- If the file testfile already exists, it is overwritten with the concatenated files testfile1, testfile2 and testfile3. If testfile already exists and you want to concatenate at the end of the existing file, instead of using the redirection operator >, you must use the >> (two consecutive greater than sign) operator as follows:

  cat testfile1 testfile2 testfile2 >> testfile

`ln`

`ln` provides alternate names for the same file.

It links a file name to another one and it is possible to link a file to another name in the same directory or the same name in another directory.

When linking a filename to another filename, you can specify only two arguments: the source filename and the target filename. When linking a filename to a directory, you can specify multiple filenames to be linked to the same directory.

The flags that can be used with the ln command are as follows:

-s to create a soft link to another file or directory. In a soft link, the linked file contains the name of the original file. When an operation on the linked filename is done, the name of the original file in the link is used to reference the original file.

-f to ensure that the destination filename is replaced by the linked filename if the file already exists.

`ln`: Exercices

★If you want to link testfile1 to testfile2 in the current directory, execute the following command:

ln testfile1 testfile2

This creates a hard linked testfile2 linking it to tesftfile1. In this case, if one of the files is removed, the other will remain unaltered.

★If testfile is in the current directory and is to be linked to testfile in the directory /u/testuser/testdir, execute the following command:

ln testfile /u/testuser/testdir

★To create a symbolic link of testfile1 in the current directory, execute the following command:

ln -s testfile1 testfile2

This creates a linked testfile2, which will contain the name of testfile1. If you remove testfile1, you will be left with an orphan testfile2, which points to nowhere.

Locating commands

- ★To execute a command, UNIX has to locate the command before it can execute it
- ★UNIX uses the concept of search path to locate the commands.
- ★Search path is a list of directories in the order to be searched for locating commands. Usually it contains standard paths (/bin, /*usr*/bin, ...)
- ★Modify the search path for your environment modifying the PATH environment variable

`which`

★`which` can be used to find whether a particular command exists in you search path. If it does exist, which tells you which directory contains that command.

Examples (try with existing and not existing commands):

```
which  pippo
which gedit
which vim
```

File information commands

★Each file and directory in UNIX has several attributes associated with it. UNIX provides several commands to inquire about and process these attributes

`find`

★`find` search for the particular file giving the flexibility to search for a file by various attributes: name, size, permission, and so on. Additionally, the find command allows to execute commands on the files that are found as a result of the search.

Command:

find directory-name search-expression

`find` Examples (try)

```
find . -name pippo
find /etc -name networking
find /etc -name netw    # nothing found
find /etc -name netw\*

find -size 18        # 18 blocks files
find -size 1024c   # 1024 bytes

find . -print
```

Try other options

`file`

★`file` can be used to determine the type of the specified file.

Examples (try):

$ file /etc/networking/interfaces

/etc/networking/interfaces: cannot open `/etc/networking/interfaces' (No such file or directory)

$ file /etc/network/interfaces

/etc/network/interfaces: ASCII text

UNIX Processes

Usually, a command or a script that you can execute consists of one or more processes.

The processes can be categorized into the following broad groups:

- ★ Interactive processes, which are those executed at the terminal. Can execute either in foreground or in background. In a foreground process, the input is accepted from standard input, output is displayed to standard output, and error messages to standard error. In background, the terminal is detached from the process so that it can be used for executing other commands. It is possible to move a process from foreground to background and vice versa (<ctrl+bg>; <ctrl+fg>.
- ★ Batch processes are not submitted from terminals. They are submitted to job queues to be executed sequentially.
- ★ Deamons are never-ending processes that wait to service requests from other processes.

Process related commands (2)

★In UNIX, each process has a number of attributes associated with it. The following is a list of some of these attributes:

Process ID is a unique identifier assigned to each process by UNIX. You can identify a process during its life cycle by using process ID.
Real User ID is the user ID of the user who initiated the process.
Effective User ID is the user ID associated with each process. It determines the process's access to system resources. Under normal circumstances, the Real User ID and Effective User ID are one and the same. But it is

Process attributes

★In UNIX, each process has a number of attributes associated with it. The following is a list of some of these attributes:

- Process ID is a unique identifier assigned to each process
- Real User ID is the user ID of the user who initiated the process.
- Effective User ID is the user ID associated with each process. It determines the process's access to system resources. Under normal circumstances, the Real User ID and Effective User ID are one and the same. They can differ by setting the Set User ID flag on the executable program, if you want a program to be executed with special privilege without actually granting the user special privilege.
- Real Group ID is the group ID of the user who initiated the process.
- Effective Group ID is the group ID that determines the access rights. The effective group ID is similar to the effective user ID.
- Priority (Nice Number) is the priority associated with a process relative to the other processes executing in the system.

Process Related Commands

★a command or a script that you can execute consists of one or more processes.

The main are:

- `kill`
- `ps`
- `wait`
- `nohup`
- `sleep`

`ps`

★`ps` command is used to find out which processes are currently running.

Exercises:

- Try the following commands, check the differences in the output. Read the flag meaning using `man ps`:

  ps

  ps -ef

  ps -aux

`kill`

★`kill` is used to send signals to an executing process. The process must be a nonforeground process for you to be able to send a signal to it using this command.

★The default action of the command is to terminate the process by sending it a signal. If the process has been programmed for receiving such a signal. In such a case, the process will process the signal as programmed.

★ You can kill only the processes initiated by you. However, the root user can kill any process in the system.

★The flags associated with the kill commands are as follows:

-l to obtain a list of all the signal numbers and their names that are supported by the system.

-'signal number' is the signal number to be sent to the process. You can also use a signal name in place of the number. The strongest signal you can send to a process is 9 or kill.

`kill` Exercises

- ★Look for a process PID of a process belonging of you (using ps) and kill it using two different signals: -9 and -15.
- ★List all available signals and red the differences between the two signal previously used

`wait` with exercises

★`wait` is tp wait for completion of jobs. It takes one or more process IDs as arguments. This is useful while doing shell programming when you want a process to be finished before the next process is invoked.
If you do not specify a process ID, UNIX will find out all the processes running for the current environment and wait for termination of all of them.

★Examples:

- `wait` If you want to find out whether all the processes you have started have completed
- `wait 15060` If you want to find out whether the process ID 15060 has completed

★The return code from the wait command is zero if you invoked the wait command without any arguments. If you invoked the wait command with multiple process IDs, the return code depends on the return code from the last process ID specified.

`wait`: exercise

- ★From a shell launch an infinite process using:
  `while true; do echo looping; sleep 2; done`
- ★From another shell find the pid of this process using `ps` command
- ★From a third shell launch a process waiting for the end of the initial infinite loop
  pid=<your_process_pid>; wait $pid
- ★From a fourth shell kill the first process (pid)
- ★ Check in the third shell that your waiting process ended

NOT WORKING using shells. Needs scripting: next time.

`nohup`

★When you are executing processes under UNIX, they can be running in foreground or background. In a foreground process, you are waiting at the terminal for the process to finish. Under such circumstances, you cannot use the terminal until the process is finished. You can put the foreground process into background as follows:

Ctrl-z

bg

The processes in UNIX will be terminated when you logout of the system or exit the current shell whether they are running in foreground or background. The only way to ensure that the process currently running is not terminated when you exit is to use the nohup command.

`nohup`

The nohup command has default redirection for the standard output. It redirects the messages to a file called nohup.out under the directory from which the command was executed. That is, if you want to execute a script called sample_script in background from the current directory, use the following command:

nohup sample_script &

The & (ampersand) tells UNIX to execute the command in background. If you omit the &, the command is executed in foreground. In this case, all the messages will be redirected to nohup.out under the current directory. If the nohup.out file already exists, the output will be appended to it.

`nohup`: Examples

nohup grep sample_string * &

nohup grep sample_string * > mygrep.out &

nohup my_script > my_script.out &

`sleep`

`sleep` wait for a certain period of time between execution of commands. This can be used in cases where you want to check for, say, the presence of a file, every 15 minutes. The argument is specified in seconds.

Examples: If you want to wait for 5 minutes between commands, use:

```
sleep 300
```

Small shell script that reminds you twice to go home, with a 5-minute wait between reminders:

```
echo "Time to go home"
sleep 300
echo "Final call to go home ....."
```

File Content Related Commands

★Commands that can be used to look at the contents of the file or parts of it. You can use these commands to look at the top or bottom of a file, search for strings in the file, and so on.

`more`

★`more` can be used to display the contents of a file one screen at a time. By default, the more command displays one screen worth of data at a time. The more command pauses at the end of display of each page. To continue, press a space bar so that the next page is displayed or press the Return or Enter key to display the next line. Mostly the more command is used where output from other commands are piped into the more command for display.

★Try

`less`

★`less` is to view the contents of a file. This may not be available by default on all UNIX systems. It behaves similarly to the more command. The less command allows you to go backward as well as forward in the file by default.

★Try

★Cat <a big file> | less

`tail`

★`tail` to display, on standard output, a file starting from a specified point from the start or bottom of the file. Whether it starts from the top of the file or end of the file depends on the parameter and flags used. One of the flags, -f, can be used to look at the bottom of a file continuously as it grows in size. By default, tail displays the last 10 lines of the file.

`tail` exercises

tail -f500 /*var*/log/syslog

list of flags that can be used with the tail command:
-c number to start from the specified character position number.
-b number to start from the specified 512-byte block position number.
-k number to start from the specified 1024-byte block position number.
-n number to start display of the file in the specified line number.
-r number to display lines from the file in reverse order.
-f to display the end of the file continuously as it grows in size.

`wc`

`wc` counts the number of bytes, words, and lines in specified files. A word is a number of characters stringed together delimited either by a space or a newline character.

Following is a list of flags that can be used with the wc command:

-l to count only the number of lines in the file.
-w to count only the number of words in the file.
-c to count only the number of bytes in the file.

You can use multiple filenames as argument to the wc command.

`wc` exercices

wc file

wc -w file

cat <file> | wc -l

wc -w <file1> <file2>

`read`

`read` is used in shell scripts to read each field from a file and assign them to shell variables. A field is a string of bytes that are separated by a space or newline character. If the number of fields read is less than the number of variables specified, the rest of the fields are unassigned. Flag -r to treat a \(backslash) as part of the input record and not as a control character.

`read` Examples

Examples Following is a piece of shell script code that reads first name and last name from namefile and prints them:

```
while read -r lname fname
do
        echo $lname","$fname
done < namefile
```

`read` Examples

Examples Following is a piece of shell script code that reads a file by line:

```
while read -r line
do
    printf 'Line: %s\n' "$line"
done < /etc/network/interfaces
```

`tee`

`tee` to execute a command and want its output redirected to multiple files in addition to the standard output, use the tee command. The tee command accepts input from the standard input, so it is possible to pipe another command to the tee command.

Following is an optional flag that can be used with the tee command:
-a to append to the end of the specified file. The default of the tee command is to overwrite the specified file.

`tee` Examples (try)

use the cat command on file1 to display on the screen, but you want to make a copy of file2, use the tee command as follows:

```
cat file1 | tee file2 | more
```

append file1 to the end of an already existing file2, use the flag -a as in the following command:

```
cat file1 | tee -a file2 | more
```

File Content Search Commands

For searching for a pattern in one or more files, use the grep series of commands. The grep commands search for a string in the specified files and display the output on standard output.

`egrep`

`egrep` extended version of grep command. This command searches for a specified pattern in one or more files and displays the output to standard output. The pattern can be a regular expression to match any single character.

***** to match one or more single characters that precede the asterisk.

^ to match the regular expression at the beginning of a line.

$ to match the regular expression at the end of a line.

+ to match one or more occurrences of a preceding regular expression.

? to match zero or more occurrences of a preceding regular expression.

[ ] to match any of the characters specified within the brackets.

`egrep` Examples

Let us assume that we have a file called file1 whose contents are shown below using the more command:

more file1

***** *This file is a dummy file* *****

which has been created

to run a test for egrep

grep series of commands are used by the following types of people

programmers

end users

Believe it or not, grep series of commands are used by pros and novices alike

***** *THIS FILE IS A DUMMY FILE* *****

`egrep` Examples

- If you are just interested in finding the number of lines in which the specified pattern occurs, use the -c flag as in the following command:

  egrep -i -c dummy file1
- If you want to get a list of all lines that do not contain the specified pattern, use the -v flag as in the following command:

  egrep -i -v dummy file1
- If you are interested in searching for a pattern that you want to search as a word, use the -w flag as in the following command:

  egrep -w grep file1

`egrep` Examples

- If you want to find all occurrences of dummy, use the following command:

```
egrep dummy file1
*****  This file is a dummy file *****
```

- If you want to find all occurrences of dummy, irrespective of the case, use the -i flag as in the following command:

```
egrep -i dummy file1
*****  This file is a dummy file *****
*****  THIS FILE IS A DUMMY FILE *****
```

- If you want to display the relative line number of the line that contains the pattern being searched, use the -n flag as in the following command:

```
egrep -i -n dummy file1
1:*****  This file is a dummy file *****
8:*****  THIS FILE IS A DUMMY FILE *****
```

Bash configuration

Bash has more configuration startup files.
They are executed at bash start-up time.
The files and sequence of the files executed differ from the type of shell. Shell can be:

- ★Interactive
- ★Non-interactive
- ★Login shell
- ★Non-login shell

Bash types

- ★Interactive: means that the commands are run with user-interaction from keyboard. E.g. the shell can prompt the user to enter input.
- ★Non-interactive: the shell is probably run from an automated process so it can't assume if can request input or that someone will see the output. E.g Maybe it is best to write output to a log-file.
- ★Login: shell is run as part of the login of the user to the system. Typically used to do any configuration that a user needs/wants to establish his work-environment.
- ★Non-login: any other shell run by the user after logging on, or which is run by any automated process not coupled to a logged in user.

Bash startup files (1)

http://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

★Interactive login shell, or with –login:

/etc/profile, if that file exists

~/.bash_profile,

~/.bash_login

~/.profile, in that order, and reads and executes

--noprofile option may be used to inhibit this behavior.

When an interactive login shell exits, or a non-interactive login shell executes the exit builtin command, Bash reads and executes commands from the file ~/.bash_logout, if it exists.

Bash startup files (2)

http://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

★interactive non-login shell

[Example: when you open a new terminal window by pressing Ctrl+Alt+T, or just open a new terminal tab.]

bash reads and executes commands from ~/.bashrc, if that file exists.

--norc option ti inhibit this behaviour.

--rcfile file option will force Bash to read and execute commands from file instead of ~/.bashrc.

So, typically, your ~/.bash_profile contains the line

if [-f ~/.bashrc]; then . ~/.bashrc; fi

after (or before) any login-specific initializations.

Bash startup files (3)

http://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

★Invoked non-interactively

[Example: to run a shell script]

bash looks for the variable BASH_ENV in the environment, expands its value if it appears there, and uses the expanded value as the name of a file to read and execute. Bash behaves as if the following command were executed:

if [-n "$BASH_ENV"]; then . "$BASH_ENV"; fi

but the value of the PATH variable is not used to search for the filename.

If a non-interactive shell is invoked with the --login option, Bash attempts to read and execute commands from the login shell startup files.

Set user's PATH environment variable

```
$ cat .profile
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or
# ~/.bash_login exists.

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
. "$HOME/.bashrc"
    fi
fi
# set PATH so it includes user's private bin directories
PATH="$HOME/bin:$HOME/.local/bin:$PATH"
```

~/.bashrc file

User specific, hidden by default.

~/.bashrc

If not there simply create one.

System wide:

/etc/bash.bashrc

Status commands

★Several commands that display the status of various parts of the system. These commands can be used to monitor the system status at any point in time.

`date`

`date` command to display the current date and time in a specified format. If you are root user, use the date command to set the system date.
To display the date and time, you must specify a + (plus) sign followed by the format. The format can be as follows:
%A to display date complete with weekday name.
%b or %h to display short month name.
%B to display complete month name.
%c to display default date and time representation.
%d to display the day of the month as a number from 1 through 31.
%D to display the date in mm/dd/yy format.
%H to display the hour as a number from 00 through 23.
%I to display the hour as a number from 00 through 12.
%j to display the day of year as a number from 1 through 366.
%m to display the month as a number from 1 through 12.
%M to display the minutes as a number from 0 through 59.
%p to display AM or PM appropriately.
%r to display 12-hour clock time (01-12) using the AM-PM notation.
%S to display the seconds as a number from 0 through 59.

`date`

Other format flags:
%T to display the time in hh:mm:ss format for 24 hour clock.
%U to display the week number of the year as a number from 1 through 53 counting Sunday as first day of the week.
%w to display the day of the week as a number from 0 through 6 with Sunday counted as 0.
%W to display the week number of the year as a number from 1 through 53 counting Monday as first day of the week.
%x to display the default date format.
%X to display the time format.
%y to display the last two digits of the year from 00 through 99.
%Y to display the year with century as a decimal number.
%Z to display the time-zone name, if available.

`date`: Exercises

Try some example of `date` command usage with different display of day, month, year

- ★If you want to display the date without formatting, use date without any formatting descriptor as follows:

```
date
Sat Dec  7 11:50:59 EST 1996
```

- ★If you want to display only the date in mm/dd/yy format, use the following commands:

```
date +%m/%d/%y
12/07/96
```

- ★If you want to format the date in yy/mm/dd format and time in hh:mm:ss format, use the following command:

```
date "+%y/%m/%d %H:%M:%S"
96/12/07 11:57:27
```

- ★Following is another way of formatting the date:

```
date +%A","%B" "%d","%Y
Sunday,December 15,1996
```

Variables (1)

★Variables are a way of passing information from the shell to programs when you run them.
Programs look "in the environment" for particular variables and if they are found will use the values stored.

★Variables can be set:

by the system,

by you,

by the shell,

by any program that loads another program.

Variables (2)

★Standard UNIX variables are split into two categories:

- environment variables:

  if set at login, are valid for the duration of the session

- shell variables:

  apply only to the current instance of the shell and are used to set short-term working conditions;

By convention, environment variables have UPPER CASE and shell variables have lower case names.

bash variables

Variable in bash are untyped.

A variable in bash can contain a number, a character, a string of characters. You have no need to declare a variable, just assigning a value to its reference will create it.

Example (try):

1) STR="Hello World!"
   echo $STR
2) Try assignment and echo the variable content:

   a=5324

   a=(“1, 3, 4, 6, 5, “otto”) array
3) Very simple backup script example:
   OF=/var/my-backup-$(date +%Y%m%d).tgz
      tar -cZf $OF /home/me/

Bash Arithmetic Expansion (1)

★Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations. Translating a string into a numerical expression is relatively straightforward using backticks, double parentheses, or let.

★ Backticks examples:

```
z=15
z=`expr $z + 3`
echo $z
```

Demonstrating some of the uses of 'expr' (1)

```
# Arithmetic Operators
a=`expr 5 + 3`
echo "5 + 3 = $a"

a=`expr $a + 1`
# incrementing a variable
echo "a + 1 = $a"

# modulo
a=`expr 5 % 3`
echo "5 mod 3 = $a"
```

• Demonstrating some of the uses of 'expr' (2)

```
# Logical Operators
#  Returns 1 if true, 0 if false,
#+ opposite of normal Bash convention.
x=24
y=25
b=`expr $x = $y`          # Test equality.
echo "b = $b"             # 0  ( $x -ne $y )


a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, therefore...'
  ★echo "If a > 10, b = 0 (false)"
```

• Demonstrating some of the uses of 'expr' (3)

```
a=3
b=`expr $a \> 10`
# echo "If a > 10, b = 0 (false)"
echo "b = $b"              # 0  ( 3 ! -gt 10 )
b=`expr $a \< 10`
# "If a < 10, b = 1 (true)"
echo "b = $b"              # 1  ( 3 -lt 10 )
# Note escaping of operators.
b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (true)"
echo "b = $b"              # 1  ( 3 -le 3 )
# There is also a "\>=" operator (greater than or equal to).
```

Demonstrating some of the uses of 'expr' (4)

```
# String Operators
# ------ ---------
a=1234zipper43231

# length: length of string
b=`expr length $a`
echo "Length of \"$a\" is $b."

# index: position of first character in substring
#        that matches a character in string
b=`expr index $a 23`
echo "Numerical position of first \"2\" in \"$a\" is \"$b\"."
```

Bash Arithmetic Expansion (2)

★Parentheses examples:

$((EXPRESSION)) is arithmetic expansion.

Not to be confused with + command substitution.

Examples:

```
n=0
echo "n = $n"                    # n = 0
(( n += 1 ))                     # Increment.
echo "n = $n"                    # n = 1

# (( $n += 1 ))                  # is incorrect!
echo (( $n += 1 ))
```

Bash Arithmetic Expansion (3)

★Parentheses examples:

$((EXPRESSION)) is arithmetic expansion.

Not to be confused with + command substitution.

Examples:

```
n=0
echo "n = $n"                    # n = 0
(( n += 1 ))                     # Increment.
echo "n = $n"                    # n = 1

# (( $n += 1 ))                  # is incorrect!
echo (( $n += 1 ))
```

Bash Arithmetic Expansion (4)

★`let` examples:

```
let z=z+3
let "z += 3"  #  Quotes permit the use of spaces in
              #  variable assignment.
              #  The 'let' operator actually performs
              #  arithmetic evaluation,
              #  rather than expansion.
echo $z
```

Exercise

★Try to translate some of the examples seen for `expr` in example of double quotes or `let` usage

```
z=15
z=$((z + 3))        # (($z+3)) wrong!!!
echo $z

a=3
(($a>10))           # right!!!
echo $?
(($a>2))
echo $?
```

`set`

`set` sets shell attributes, for example, the positional parameters.

Example:

```
$ set foo=baz
$ echo "$1"
foo=baz
```

Note that baz is not assigned to foo, it simply becomes a literal positional parameter.
`set` also prints variables that are not exported.
To see other possible operations: `help set`.

Shell scripting abilities

Many shells have scripting abilities:

Put multiple commands in a script and the shell executes them as if they were typed from the keyboard.

Most shells offer additional programming constructs that extend the scripting feature into a programming language.

Main Emacs commands

Principali comandi di `emacs`	
Undo	`CTRL-x u` oppure `CTRL-_`
Salva il file	`CTRL-x CTRL-s`
Salva con nome diverso	`CTRL-x CTRL-w` *nome*
Apre un nuovo file	`CTRL-x CTRL-f` *nome*
Inserisce un file	`CTRL-x i` *nome*
Passa ad un altro buffer	`CTRL-x b`
Chiude un buffer	`CTRL-x k`
Divide la finestra in due	`CTRL-x 2`
Passa da una metà all'altra	`CTRL-x o`
Riunifica la finestra	`CTRL-x 1`
Refresh della finestra	`CTRL-l`
Quit da `emacs`	`CTRL-x CTRL-c`
Cursore a fine riga	`CTRL-e`
Cursore a inizio riga	`CTRL-a`
Cursore giù una pagina	`CTRL-v`
Cursore su una pagina	`ESC v`
Inizio del buffer	`ESC <`
Fine del buffer	`ESC >`
Vai alla linea...	`ESC x goto-line` *numero*

Cerca testo	`CTRL-s` *testo*
Sostituisce testo	`ESC %` *testo1 testo2*
Marca inizio di un blocco	`CTRL-SPACE`
Marca fine blocco e taglia	`CTRL-w`
Marca fine blocco e copia	`ALT-w`
Incolla blocco	`CTRL-y`
Pagina di aiuto	`CTRL-h CTRL-h`
Significato di un tasto	`CTRL-h k` *tasto*
Significato di tutti i tasti	`CTRL-h b`
Interrompe comandi complessi	`CTRL-g`
Apre una shell dentro `emacs`	`ESC x shell`
Aiuto psicologico	`ESC x doctor`
Torri di Hanoi	`ESC x hanoi`

http://www.di.unipi.it/~bozzo/fino/appunti/node2.htmla