

u2-s1-linux

Sistemi Operativi

Unità 2: Utilizzo di Linux

## Ambienti Linux

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

### Argomenti

1. Distribuzioni di Linux
2. Alternative per usare Linux

---

## Distribuzioni di Linux

---

### Distinzione tra Linux e GNU/Linux

#### **LINUX.**

Con **Linux** si intende *il Kernel UNIX-like creato da Linus Torvald.*

#### **GNU/LINUX.**

Con **GNU/Linux** si *intende una famiglia di sistemi operativi basati su Kernel Linux*

Ci sono *più di 100 SO* che sono della famiglia **GNU/Linux**. Condividono:

- Kernel Linux
  - Programmi e utility di base di GNU per gestione di file, processi, rete
-

## Distribuzioni di Linux: Ubuntu

Le famiglie principali OS Linux sono:

**Ubuntu:** attualmente il *più diffuso*.

- Basato su un'altra distribuzione chiamata **Debian**
  - Debian è il *punto di partenza* per tanti altri OS Linux (o *distribuzioni Linux*)
  - Debian contiene *solo software libero*, Ubuntu no; ad esempio in Debian *non c'è* la *decodificazione di alcuni formati audio codecs*.
- Ha lo scopo di offrire un SO completo e facile da usare per PC (e per server)
- Ne derivano *altri distribuzioni* che di differenziano per il software che gestisce l'ambiente grafico (desktop, finestre); ad esempio abbiamo *XUbuntu*, *Kali* (storicamente) eccetera...

---

## Distribuzioni di Linux: Red Hat

**Red Hat Enterprise Linux e CentOS:** *versioni professionali* di Linux, per il mercato aziendale

- *Red Hat* è la *ditta che crea queste distribuzioni*
- Particolare attenzione a *stabilità e sicurezza*
- Mantenute dall'azienda Red Hat, che offre *supporto (commerciale) a pagamento*
- *RHEL* è la versione con supporto commerciale. CentOS è la versione *consumer*
- L'OS **Fedora** è della stessa famiglia, è adotta funzionalità più innovative, sebbene meno stabili (*versione più aggiornata*)

---

## Distribuzioni di Linux: Arch, openSUSE e Mint

**Arch Linux:** distribuzione leggera, adatta a sistemi minimali e con poche risorse

- Non prevede un ambiente Desktop di default
- Utilizza la filosofia KISS (Keep It Simple, Stupid)

**openSUSE Linux:** sviluppata da volontari. Nei primi anni 2000 era molto diffusa

**Linux Mint:** basata su Ubuntu. Ha avuto momenti di celebrità nei primi anni 2010

---

## L'infinità delle distribuzioni di Linux

Impossibile enumerare tutte le distribuzioni (ce ne sono quasi *infinite!*).

Molte nascono e muoiono nel giro di pochi anni (si dicono "*abandonware*")

In questo corso utilizzeremo *Ubuntu*

- Diffuso
- Semplice
- Generico
  - Ha versione per PC e per server



# Alternative per usare Linux

---

## Alternative per usare Linux

Per utilizzare un sistema Linux, ci sono varie alternative a seconda che:

- Si abbia in PC o un MAC
  - Si abbia tanto o poco spazio su disco
  - Si sia più o meno esperti nell'utilizzo del computer
- 

## Alternative per usare Linux

**Installazione Nativa:** si installa un SO Linux su un PC.

- Necessario scaricare l'immagine dal sito in un SO Linux (e.g., Ubuntu)
  - Il PC viene formattato e il SO è installato nativamente
  - Si può mantenere Windows (o Mac OS) usando il *Dual Boot*
    - L'hard disk è partizionato in due drive logici, uno con Linux, uno con Windows
  - Operazione non facilissima, e potenzialmente distruttiva
- 

## Alternative per usare Linux

**Linux da USB "Live":**

Ogni distribuzione di Linux può essere usata Live:

- Si crea una chiavetta USB *bootable*
  - Si inserisce nel PC e lo si forza a fare *boot* da chiavetta
  - Linux gira nativamente come se fosse installato
  - Su Windows si può usare il software Rufus <https://rufus.ie/en/>
    - **Nota:** a meno che non lo si configuri esplicitamente, la chiavetta non è **persistente**. A ogni riavvio si perdono tutti i file modificati
-

## Alternative per usare Linux

**Macchina Virtuale:** utilizzando un software chiamato *virtualizzatore* è possibile creare un PC virtuale.

- E' a tutti gli effetti in PC completo di tutte le funzionalità
    - Ha una CPU, memoria e disco virtuali
  - Che gira all'interno di un'applicazione
    - Non danneggia nè impatta il SO nativo del PC
  - Tanti software per virtualizzazione
    - **VirtualBox** (consigliato)
    - **VMWare**
    - **QEMU**
- 

## Alternative per usare Linux

- Passi necessari:
    - Installare il virtualizzatore
    - Creare una nuova macchina virtuale
      - Specificare la quantità di risorse (CPU, memoria, disco) da allocare alla macchina virtuale
    - Installare il SO Linux preferito
    - Configurarlo con i software desiderati, se necessario
  - Questa è l'*opzione consigliata*:
    - Facile, stesse potenzialità di avere Linux installato nativamente
    - Il PC deve essere abbastanza potente:
      - Almeno 8 core, 8GB di RAM e 20GB (di spazio libero) su Hard Disk
- 

## Alternative per usare Linux

**Cygwin:** è un software da installare su Windows

E' un layer di compatibilità POSIX che permette di usare programmi POSIX su sistemi Windows

- Mappa le system call POSIX su quelle di Windows.
- Include i tool GNU base per gestione di file, compilazione
- Necessario compilare i programmi usando Cygwin

Facile da installare:

- Si installa come un normale programma

- Sito Web: [www.cygwin.com](http://www.cygwin.com)

---

## Alternative per usare Linux

**Windows Subsystem for Linux (WSL):** è anche esso un layer di compatibilità per programmi Linux su Windows.

- Sviluppato direttamente da Microsoft
- A partire da Windows 10
- Permette di eseguire eseguibili Linux senza ricompilare
- Si può installare tramite command line di Windows
- Dopodichè è possibile installare pacchetti di software Linux
  - Ad esempio si può installare l'applicazione "Ubuntu" tramite software center.
  - Nota: l'applicazione "Ubuntu" non è un vero SO. E' solo un pacchetto che contiene i software di base di Ubuntu e una shell

---

## Alternative per usare Linux

**Terminale via browser:**

- Tante opzioni online
  - Cercare su Google: [linux box online](#)
- Una è: <https://linuxcontainers.org/incus/try-it/>
  - Non persistente
  - Limitata a 30 minuti
  - Va bene per provare i comandi Linux *al volo*

---

## Domande

Ubuntu é un SO che utilizza il Kernel:

• **Linux** • **UNIX** • **POSIX**

Red Hat é un:

• **Kernel** • **SO** • **Uno standard**

u2-s2-concetti-linux

# Sistemi Operativi

## Unità 2: Utilizzo di Linux

### Comandi di Linux

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

### Argomenti

1. Sessione Shell
2. Comandi di base
3. File System
4. Utenti e permessi
5. Processi e programmi
6. Altri comandi
7. Esercizi

---

### Sessione Shell

#### Login

Per fare login, è necessario inserire le proprie *credenziali sul terminale* (in realtà di solito viene fatta *automaticamente*):

```
login as: <username>
password: <password>
```

Per fare logout: **CTRL+D**, **exit** o **shutdown** (solo *superuser*, detto *root*)

---

### Terminali Remoti attraverso SSH

E' possibile usare un terminale *remoto* utilizzando SSH (*Secure Shell*).



Su un'altra macchina collegata *in rete*, digitare su terminale:

SHELL

```
ssh <username>@<indirizzo IP della macchina>
```

Si utilizza il protocollo *Secure Shell*, che *trasmette* in maniera cifrata i comandi e il loro output tramite la rete.

---

## Comandi di base

---



### Argomenti e Opzioni dei Comandi

Vengono digitati sul terminale. Avviano il corrispondente eseguibile

Alcuni ammettono *argomenti*, ovvero gli *oggetti* su cui il comando deve agire.

- Specificati dopo il nome del comando

Alcuni ammettono *opzioni* che specificano comportamenti particolari

- Iniziano per  seguite da una singola lettera
- Oppure per  seguite da una stringa

---

### Formato dei comandi

**Formato:**

SHELL

```
comando [opzioni] [argomenti]
```

**Esempio:** stampa il contenuto di **file.txt**

SHELL

```
cat file.txt
```

**Esempio:** lista il contenuto della cartella `dir`, includendo anche i file nascosti (che iniziano per `.`):

SHELL

```
ls -a dir
```

SHELL

```
ls --all dir
```

---

## Concatenazione dei comandi

E' possibile avere più comandi con una sola riga, separandoli con `;`.

SHELL

```
comando1 ; comando2; ...
```

### Altri comportamenti:

- I comandi possono essere concatenati tramite il carattere `|` (questo concetto sarà ben noto come *pipe*).
- Si può redirezionare l'output di un comando su file tramite il carattere `>` (questo concetto sarà ben noto quando vedremo i *flussi stdin, stdout e stderr*).
- Analizzato in dettaglio più avanti

---

## Comando manuale

**Manuale in linea:** i comandi sono documentati

```
man <comando>
```

Restituisce la pagina di manuale del `<comando>`. Particolarmente utile per capire gli *argomenti* e le *opzioni* del comando.

Comandi simili:

- `apropos`: ricerca in tutti i manuali dei comandi
- `whereis`: trova il binario, il sorgente e il manuale di un comando

## Altri comandi di base utili

### Altri comandi di base:

- **date**: visualizza la *data*
- **who**: mostra gli *utenti attualmente collegati*.
- **uptime**: tempo di vita di un sistema, numero di utenti collegati, carico del sistema negli ultimi 1, 5, 15 minuti; utile nell'ambito in cui si usano i *server*, dato che di solito vanno rimasti accesi *per sempre*. Con questo comando si vede se un *server* sia stato *riavviato o meno*, che potrebbe essere sorgente di problemi.
- **hostname**: nome della macchina

---

## File System

### Organizzazione del File System

Il file system su Linux è *gerarchico*, ovvero *ad albero*.

- Organizzato in directory annidate l'una dentro l'altra
- La directory radice è **/** (detta "*root*")
- Tutte le cartelle del sistema sono contenute nella directory radice.

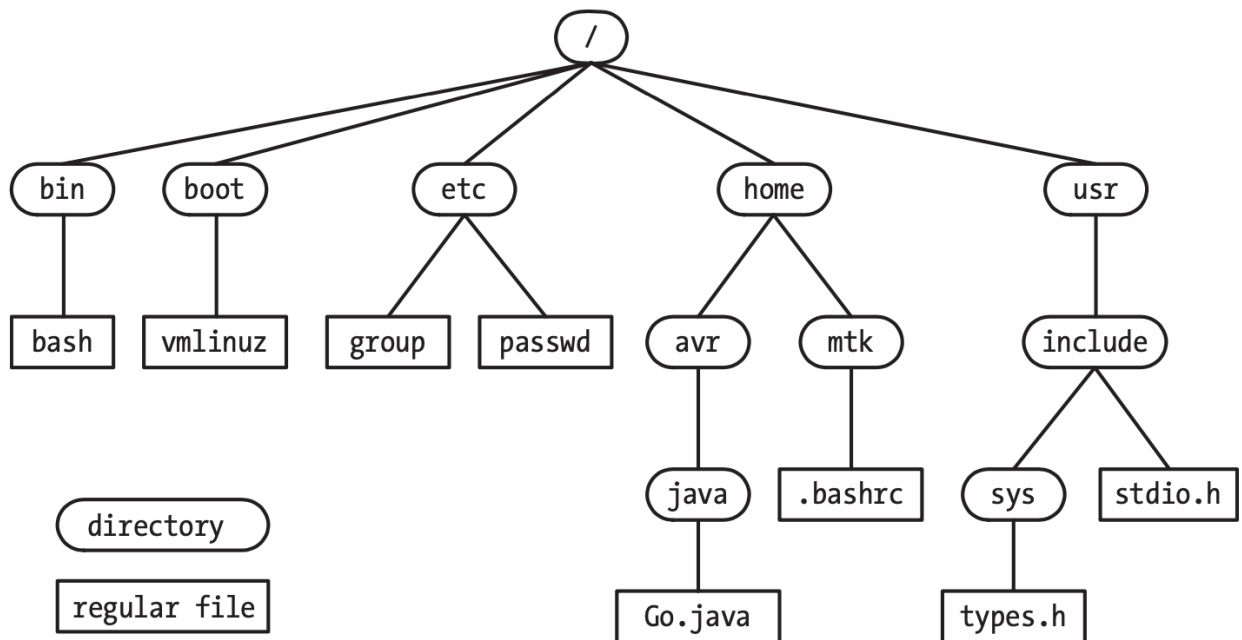
Esempio: la home degli utenti si trova in

**/home/nomeutente**

---

### Esempio di File System

Esempio di albero (*parziale*) delle cartelle di un sistema Linux



## Path

### DEFINIZIONE DI PATH.

Un *path* identifica un *file* o una *cartella*.

Un terminale (e i comandi che vi vengono lanciati) sono *sempre posizionati* su una cartella, la *Working Directory*; ovvero "*da dove li eseguo*"

- Ci muove da una cartella all'altra col comando **cd <path>**
- Un path può essere:
  - Assoluto: inizia con **/** e indica un path completo a partire dalla radice
  - Relativo: **non** inizia con **/** e indica un path *a partire dalla Working Directory*
- Ci sono dei "*path speciali*":
  - Ci si può riferire directory corrente con **.**
  - La directory padre di quella corrente è indicata con **..**

## Esempio di Path

**Esempio:** si consideri il seguente albero di cartelle

```
/
├── tmp
│   ├── directory
│   └── file.txt
```

Col comando **cd /tmp** si posiziona il terminale in **/tmp**.

A questo punto:

- **directory** identifica la directory **/tmp/directory**
- **.** identifica la directory **/tmp**
- **..** identifica la directory **/**
- **directory/./..** rappresenta la directory **/**

---

## Strutturazione generale delle cartelle Linux

Su tutti i sistemi Linux, il file system è organizzato con *le seguenti cartelle di sistema*, necessari al *funzionamento del sistema operativo*. Di queste ne elenchiamo 16.

- **/**: radice
- **/bin**: file *eseguibili* (e *preinstallati*) del sistema – ls, pwd, cp, mv ....
- **/boot**: (*alcuni*) file necessari per l'avvio del sistema, *boot loader* ...
- **/dev**: file speciali che descrivono i *dispositivi* (I/O) – dischi, scheda audio, porte seriali ...
- **/etc**: file eseguibili, script, inizializzazione, *configurazione sistema* (ad esempio indirizzo di rete), file password, ...
- **/home**: directory delle *home directory degli utenti*
- **/lib**: *librerie di sistema* (ricordiamoci che queste *NON* sono i *system calls*!)
- **/lost+found**: contiene i file *danneggiati*
- **/mnt**: punto di *montaggio file system* (mount point) (per i *dispositivi rimovibili*)
- **/proc**: file system virtuale che contiene informazioni sui *programmi in esecuzione* (processi)
- **/sys**: programmi di sistema (su *informazioni del sistema*)
- **/tmp**: direttorio *temporaneo*; il sistema operativo ha diritto di *svuotare* questa cartella per *certe situazioni* (quando non si ha abbastanza spazio) o per un certo *intervallo di tempo*.
- **/usr**: file relativi alle *applicazioni installate*
- **/usr/include**: header file libreria *standard C* (come **#include stdio.h**)
- **/usr/bin** e **/usr/lib**: file binari disponibili agli utenti
- **/var**: (*molto importante!*) file di sistema che *variano con frequenza elevata*; ad esempio è utile per le *storage delle web-app installate di sistema*.

---

## Comando "list directory"

**Comando** **ls [opzioni] [dir]**: lista il contenuto della directory. Opzioni principali:

- **-1**: stampa su una colonna

- **-l**: formato lungo
- **-n**: come -l ma visualizza gli ID al posto del nome del proprietario e del gruppo
- **-t**: ordina per data
- **-s**: mostra la dimensione dei file in blocchi
- **-a**: mostra tutti i file compresi **.** e **..**
- **-R**: elenca il contenuto in modo ricorsivo

**Esempio:** **ls -ahl** è un utilizzo comune per utilizzare questo comando.

---

## Esempio di utilizzo del comando **ls**

**Esempio:** differenti forme di **ls**

```
$ ls
compile.txt style.css u1-introduzione u2-linux
```

```
$ ls -l
total 16
-rw-rw-r-- 1 martino martino 102 set 30 14:16 compile.txt
-rw-rw-r-- 1 martino martino 199 set 30 15:27 style.css
drwxrwxr-x 3 martino martino 4096 ott 1 18:33 u1-introduzione
drwxrwxr-x 3 martino martino 4096 ott 4 10:20 u2-linux
```

---

## Comando "remove"

**Comando** **rm [-rfi] [filename]**: *rimuove* il/i file selezionati. Opzioni principali:

- **-r**: rimozione *ricorsiva* del contenuto delle directories.
- **-f**: rimozione di *tutti i file* (anche *protetti in scrittura*) senza avvisare.
- **-i**: con questa opzione **rm** chiede conferma

**Esempio:** cancella tutti i file in **cartella**

```
rm cartella/*
```

**Nota:** con **\*** si intendono tutti i file dentro una cartella

**NOTA BENISSIMO!** Questo comando è *pericolosissimo*, quindi quando si scrive un comando che usa **rm** bisogna stare non attenti, ma di più! Ad esempio il comando

```
sudo rm -rf /*
```

è in grado di cancellare l'*intero computer* e tocca ri-installare un qualsiasi sistema operativo.

---

## Comandi per modificare il File System

**Comando** `cd <dir>`: *cambia* directory. ("*change directory*")

**Comando** `mkdir <dir>`: crea sub-directory. ("*make directory*")

**Comando** `rmdir <dir>`: rimuove sub-directory, solo se vuota. Altrimenti fallisce. Questa è l'opzione più sicura, rispetto a `rm`. ("*remove directory*")

**Comando** `cp <file1> <file2>` e `mv <file1> <file2>`: *copia/sposta file o cartelle*. Opzioni principali

- `-f`: effettua le operazioni senza chiederne conferma
- `-i`: chiede conferma nel caso che la copia sovrasciva il file di destinazione
- `-r`: ricorsivo. Copia/sposta la directory e tutti i suoi file, incluso le sottodirectory ed i loro file

---

## Collegamenti su Linux

**Comando** `ln [-s] <sorgente> <destinazione>`: crea un *link*. In Linux esistono due tipi di link:

- **HARD LINK**: associa un *secondo path* al contenuto del file. Se il primo file viene spostato, il link rimane valido e funzionante. E' l'*opzione di default (!!!)*
  - *Robusto*: non può mai essere invalido. Non si può usare tra dischi diversi, né per linkare cartelle
- **SOFT LINK**: è un *semplice rimando* a un altro path. Se il path destinazione non esiste o viene spostato, il link semplicemente non funziona. Si usa l'opzione `-s`. Quindi si ha una specie di *file speciale*.
  - *Flessibile*: può linkare a un altro file system o a una cartella

## Ricerca di file nelle directories

**Comando** `find [path] [-n nome] [-print]`: ricerca ricorsiva di directories

**Esempio:** cerca i file che finiscono per `.txt` nella directory `/tmp`:

SHELL

```
find /tmp -name *.txt
```

E' possibile filtrare su varie **proprietà** dei file o cartelle:

- Tempo di creazione/modifica
- Utente o gruppo proprietario
- Grandezza

**Nota:** Non effettua ricerca nel *contenuto* del file, bensì solo *attributi* del file.

---

## Stampare e creare (o toccare) file

**Comando** `cat <file>`: stampa il *contenuto* di un file

**Comando** `touch <file>`: *crea* il file se non esiste; altrimenti *modifica la data dell'ultimo accesso al file*

**Esempio:** creare un file `a.txt`, aprirlo con un editor e scrivervi dentro `ciao`, poi stampare il file

SHELL

```
$ touch a.txt
... modificare con editor
$ cat a.txt
ciao
```

---

## Visualizzare e scrivere su file

**Comando** `less <file>`: apre il file in un visualizzatore interno alla shell dove si può scorrere in entrambe le direzioni, utile per i *file lunghi*

Esistono svariati *altri comandi per visualizzare il contenuto* di un file.

- Comandi per stampare file binari (`hexdump`)



- Comandi per stampare le prime (**head**) o le ultime righe (**tail**) di un file
  - *Editor avanzati* utilizzabili dentro la shell.
    - **nano** il più semplice
    - Ne esistono molti. Sono in competizione **emacs** e **vi** (o **vim**), detta *Guerra degli editor*. Qui gli editor diventano una specie di "religione" per i programmatori.
- 

## Utenti e permessi

---

### UNIX è un sistema multiutente

Un dispositivo con OS Linux può avere più *utenti* (infatti si dice che è un *sistema multiutente*).

- Essi possono fare login su una *shell* o un *terminale remoto* (SSH)
  - Ogni utente ha la sua *Home Directory* in **/home/<utente>**  
Serve per permettere all'utente di immagazzinare file personali come documenti, immagini, programmi.
- 

### Utenti e permessi

Un utente può essere assegnato a uno o più *gruppi*.

- Ogni utente deve essere associato ad \*uno ed uno solo gruppo primario\*\*
- Eventualmente un utente può essere assegnato a più *gruppi secondari*  
Meccanismo utente-gruppi serve per gestire l'accesso a file e risorse.

L'utente *root* esiste sempre ed ha massimi privilegi

**ATTENZIONE!** Non bisogna *assolutamente* confondere l'*utente root* con il *kernel-mode*! ([Definizioni Relative ai Sistemi Operativi > ^33592a](#)) In ogni caso si avviano le applicazioni *SEMPRE* in *user-mode*!

---

### Comandi relativi agli Utenti e ai Permessi

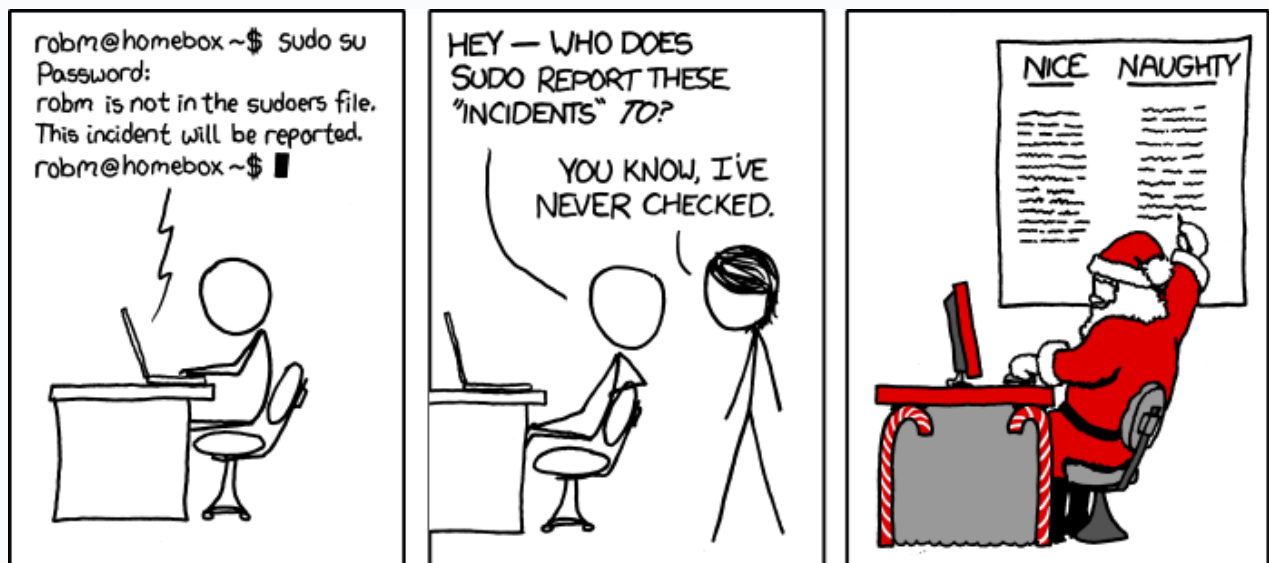
**Gestione:** Comandi per creare o rimuovere utenti e gruppi: **useradd**, **groupadd**, **userdel**, **groupdel** (*di solito sono complicati da usare*)

- Su molti OS Linux, esistono dei *comandi più facili da usare*: **adduser**, **addgroup**, **deluser**, **delgroup**

### Altri comandi:

- **groups**: stampa i gruppi ai quali appartiene l'utente corrente
- **whoami**: stampa l'*utente corrente* (per dubbi esistenziali?)
- **su <utente>**: *cambia utente* (chiede password)
- **sudo <comando>**: esegue il comando *come utente root*, dopo aver chiesto la password
  - Nota: **sudo** sta per "*super user does*"
  - Nota 2: Se si tenta di usare questo comando con un utente che non *ha il privilegio di usare sudo*, si vede questo spaventoso messaggio: **<user> is not in the sudoers file. This incident will be reported.**; ovvero si dice che l'*incidente* verrà "*segnalato*". Nel passato (fino ad un anno fa) veniva effettivamente *segnalato* tramite una *mail* all'*amministratore effettivo*. Adesso questo tentativo viene semplicemente registrato sul file **/var/log/auth.log**, se non specificato (per ulteriori dettagli vedere il commit <https://github.com/sudo-project/sudo/commit/6aa320c96a37613663e8de4c275bd6c490466b01>)

**FIGURA: Babbo natale che controlla la lista dei non-sudoers cattivi**



## Tipologie di Permessi

I file e le cartelle hanno tre tipi di *permessi*:

- 1. Permesso di **Lettura**: Per i file, *accedere a contenuto*. Per cartelle, *listare i file*.
- 2. Permesso di **Scrittura**: Per i file, *modificare il contenuto*. Per le cartelle, *creare file o cartelle in essa* (alterare la lista).
- 3. Permesso di **Esecuzione/Attraversamento**:

- Per i **file**, esiste il permesso di **esecuzione**. Necessario per *eseguire programmi*.
- Per le **cartelle**, esiste il permesso di **attraversamento**. Necessario per *accedere a sotto cartelle*.

**NOTA BENE.** Con i *permessi* non c'è *nessuna eredità*; ad esempio nel caso in cui un utente ha il permesso di *scrivere* su un file, questo non vuol dire che questo utente ha necessariamente anche il permesso di *leggere* su questo file; se l'utente ha il *solo permesso di scrivere sul file*, allora questa è l'unica cosa che può fare (anche se potrebbe risultare strana come cosa).

## Utente e Gruppo proprietario

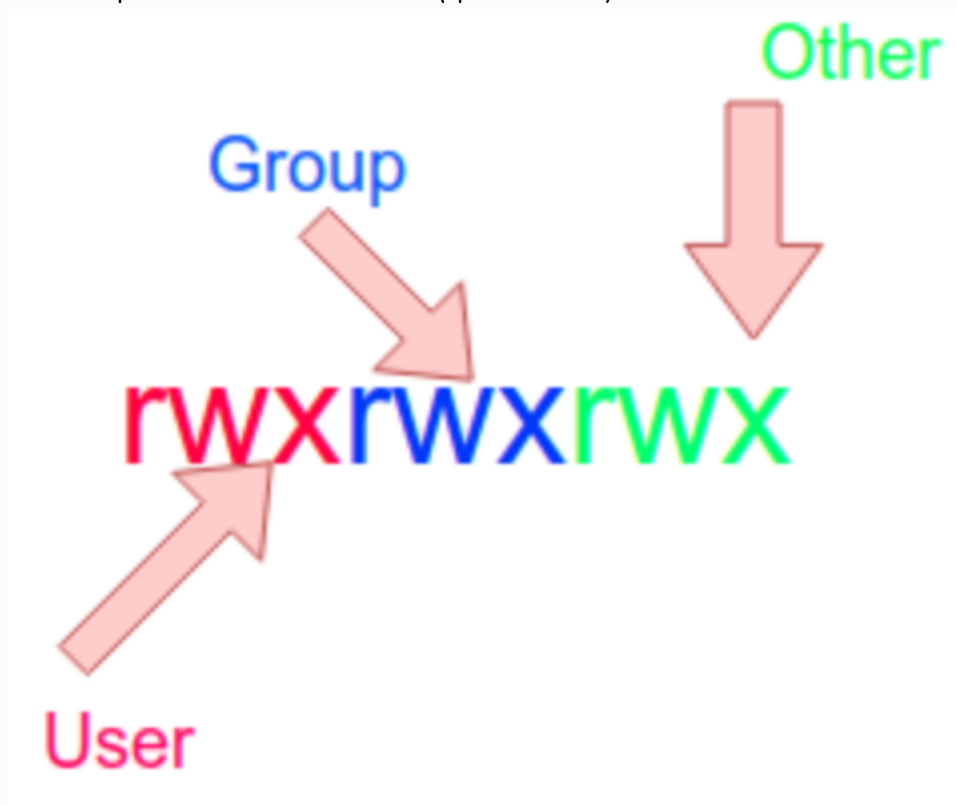
### DEFINIZIONE DI UTENTE/GRUPPO PROPRIETARIO.

I file e le cartelle hanno *uno ed uno solo utente proprietario* e un *gruppo proprietario*. I permessi su file sono gestibili separatamente per:

- Utente proprietario del file
- Utenti del gruppo proprietario del file
- Tutti gli altri utenti

Per riassumere, ad ogni file abbiamo *nove permessi*, separati per tipologia del permesso e per tipologia dell'utente: in totale ogni file o cartella ha  $3 \times 3$  permessi, in formato **rwXrwXrwX**, i *primi tre dedicati* per l'*utente proprietario*, i *secondi tre* per il *gruppo proprietario* e gli *ultimi tre* per *tutti* gli altri (quindi in ordine " $U_r U_w U_x G_r G_w G_x O_r O_w O_x$ ").

- Ogni permesso può essere attivo o no (quindi 1 o 0)



## Esempio:

```
$ ls -l
-rw-rw-r-- 1 martino docenti 102 set 30 14:16 compile.txt
-rw-rw-r-- 1 martino docenti 199 set 30 15:27 style.css
```

## Esempio generale sui file:

si considerino le seguenti informazioni sul file **my-program**:

```
-rwxr-xr-- 1 martino docenti 102 set 30 14:16 my-program
```

- L'utente **luca** del gruppo **docenti** può eseguire **my-program**?  
**SI**: il gruppo ha permessi **r-x**, quindi **luca** può eseguire **my-program**
- L'utente **marco** del gruppo **studenti** può eseguire **my-program**?  
**NO**: gli altri hanno permessi **r--**, quindi **marco** non può eseguire **my-program**

## Esempio generale sulle cartelle:

si considerino le seguenti informazioni sulla cartella **data**:

```
dr-xr--r-- 3 martino docenti 4096 ott 1 18:33 data/
```

- L'utente **luca** del gruppo **docenti** può listare i file?  
**SI**: il gruppo ha permessi **r--**
- L'utente **luca** del gruppo **docenti** può accedere alla cartella dentro **data**?  
**NO**: il gruppo ha permessi **r--**. Servirebbe **r-x**
- L'utente **martino** del gruppo **docenti** può creare file in **data**?  
**NO**: l'utente **martino** ha permessi **r-x**. Servirebbe **rwX**

## Comando per modificare i permessi di un file

**1. Modifica dei permessi di un file:** si usa il comando **chmod [-r] <permessi> <file>**

I permessi possono essere indicati con (principalmente) *due sintassi*: *assoluto* e *mirato*.

- **Assoluto**, con *tre cifre ottali*, che rappresentano rispettivamente i *permessi a utente, gruppo e altri*. Ogni cifra ha *3 bit* e rappresenta *permessi di lettura, scrittura ed esecuzione/attraversamento*.

Esempio: **chmod 750 file.txt** dà permessi totali a utente ( $7_8 = 111_2$ ), lettura/esecuzione al gruppo ( $5_8 = 101_2$ ) e niente agli altri ( $0_8 = 000_2$ )

- **Mirato**: Modifica permessi esistenti tramite una stringa composta di 3 parti:
  - Quali utenti: **u** (user), **g** (gruppo), **o** (other) (*chi?*)

- Che operazione: **+** (aggiungi), **-** (rimuovi) (*cosa?*)
- Quale permesso: **r** (lettura), **w** (scrittura), **x** (esecuzione/attraversamento) (*quale?*)

Esempio: **chmod g+w file.txt** dà permessi in scrittura agli utenti del gruppo proprietario del file

- **-r** applica il comando ricorsivamente a file e cartelle contenute
- **Chi può modificare i permessi:** *Utente proprietario* e utente **root**

**Esempio:** usi di **chmod**

- **chmod 600 file.txt**: l'utente può leggere e scrivere. Il gruppo e gli altri niente.
- **chmod 640 file.txt**: l'utente può leggere e scrivere. Il gruppo può leggere. Gli altri niente
- **chmod u+x file.txt**: Aggiungi i permessi di esecuzione all'utente
- **chmod go+w file.txt**: Aggiungi i permessi di scrittura al gruppo e a gli altri

## 2. Modifica di proprietario e gruppo di file o cartella

- **chown utente file**: modifica utente proprietario (*"change owner"*)
- **chgrp gruppo file**: (*"change group"*)
- **chown utente:gruppo file**: modifica contemporaneamente entrambi
- **Note:**
  - Posso assegnare un file *solo a un gruppo che possiedo*
  - Sulla maggior parte degli OS, solo **root** può cambiare utente proprietario; non esiste il *"give-away"* dei file.
  - Opzione **-r**: applica il comando *ricorsivamente* a cartelle e file contenuti

# Processi e programmi

## Processi e programmi

### DEFINIZIONE DI PROCESSO (RICHIAMO).

Un processo è un programma in esecuzione.

In Linux, ogni processo è identificato da un'identificatore detto *PID*.

Il *PID* si usa per *effettuare operazioni sul processo*.

## Comandi Relativi ai Processi

Abbiamo i seguenti comandi per gestire i processi.

- **kill <PID>**: termina il processo (*se ho i privilegi opportuni!*)

- **top**: mostra in maniera interattiva i *processi in esecuzione*. Simile a un Task Manager via Shell
- **ps [opzioni]**: mostra informazioni sui processi attivi.
  - **a**: informazioni su tutti i processi (non solo generati dalla sessione shell corrente)
  - **x**: mostra anche i processi in background
  - **f**: stampa i processi in modo che se ne veda il rapporto padre-figli (ovvero l'*albero dei processi*)
  - **u**: stampa più informazioni

**Esempio:** **ps fax** è un utilizzo molto comune di questo comando

  - **Nota:** **ps** è tra i pochi programmi in cui le opzioni non vanno iniziate con **-**. Ciò è un *relict* delle primissime versioni di Unix in cui le opzioni non avevano il **-**.
- **lsusb**: lista i dispositivi usb
- **lspci**: lista i dispositivi su bus pci
- **lsblk**: lista i dischi
- **ifconfig**: lista le interfacce di rete (*lo stato della rete*)
- **pwd**: stampa la *directory corrente*
- **free**: mostra quanta memoria *RAM libera* ed occupata ha il sistema
- **df [-htv]**: visualizza *informazioni sui file system del sistema* (in particolare lo *stato di occupazione dei dischi*).
  - **-t**: nr totale di blocchi e i-node liberi
  - **-v**: percentuale di blocchi e i-node
  - **-h**: stampa in GB/MB anziché in numero di byte

#### SHELL

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
udev	3923948	4	3923944	1%	/dev
tmpfs	787220	1552	785668	1%	/run
/dev/sda6	425085288	259786508	143682652	65%	/
none	4	0	4	0%	/sys/fs/cgroup
none	5120	0	5120	0%	/run/lock
none	3936080	436	3935644	1%	/run/shm
none	102400	44	102356	1%	/run/user

## Esercizi

## Esercizi

1. Stampare il contenuto del file `/etc/hosts`
2. Posizionarsi nella cartella `/tmp` e listare il contenuto della cartella `/home` usando un path relativo e uno assoluto
3. Usare l'editor nano per creare un file `file.txt` in una qualsivoglia cartella. Creare una link simbolico del file e cancellare l'originale. Cosa succede se si prova a stampare il contenuto del link? Ripetere con Hard Link
4. Creare una cartella e due file in essa. Cancellare la cartella con un unico comando.
5. Creare un nuovo gruppo `studenti` e un utente `studente` assegnato a tale gruppo.  
**Nota:** usare le opzioni `-m -g <group>` del comando `useradd`  
E' necessario usare `sudo`?  
Un utente normale può listare i file nella home della home directory di `studente`?  
Modificare i permessi della home di `utente` affinché tutti possano leggere, scrivere ed eseguire

## Soluzioni

1. Basta scrivere `cat ~/ect/hosts`
2. Indipendentemente da dove si trova, prima si scrive `cd ~/tmp`. Per il path relativo si scrive `cd ../home`; per il path assoluto si scrive `~/home`.
3. `touch file.txt`. Con un *soft link*, diventa impossibile leggere il file. Con un *hard link*, si può comunque leggere il file.
4. `rm -r(i) cartello` (la parte `-i` sarebbe opzionale, anche se è saggio usarlo)
5. Sì, è necessario usare `sudo` per creare il gruppo e l'utente. No, l'utente normale non può listare i file nella home della home directory di studente. Basta scrivere `chmod 777 /home`

## u2-s3-programmazione-bash

### Sistemi Operativi

### Unità 2: Utilizzo di Linux

## Programmi in Bash

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

## Argomenti

1. Script Bash
2. Variabili

3. Strutture di controllo
4. Esercizi

---

## Script Bash

---

### Definizione di Bash

Con *Bash* si intende il *software shell di default* in GNU/Linux  
Permette di:

- Eseguire comandi (già visto)
- Definire variabili e utilizzarle
- Controllare il flusso (**if**, **while**, ecc...) con i vari *costrutti*

E' un *linguaggio completo* di tutti i costrutti.

Ha una sintassi particolare e *problematica*; nel senso che è molto "*particolare*" e "*pedantica*" nella sua *sintassi*.

Una lista di comandi può essere racchiusa in un file detto *script*.

---

### Esempio di Script Bash

Uno script di esempio nel file **script.sh**:

```
#!/bin/bash
# primo esempio di script
echo $RANDOM
```

SHELL

**#!/bin/bash** Indica che il file è uno *script bash*; questa è *obbligatoria*!

**# primo esempio di script** E' un commento

**echo \$RANDOM** Stampa il *contenuto* della variabile **\$RANDOM**

Per eseguire lo script.

```
./script.sh
```



Il file *deve* avere permessi di lettura ed esecuzione; quindi bisogna anche usare **chmod** **u+x script.sh** per aggiungere i permessi opportuni.

---

## Il valore di Ritorno in Bash

Uno script è una lista di comandi che vengono eseguiti più delle strutture di controllo.

SHELL

```
ls # Lista i file
./myprog # Avvia il programma myprog
```

Ogni processo in Linux/POSIX deve fornire un *Valore di Ritorno* al chiamante, ovvero:

- La *shell*
- Uno *script bash*
- Un *qualsiasi altro processo*

Il valore di ritorno è un *valore numerico intero*

- Usato dal chiamante per vedere se c'è stato errore
- Per convenzione 0 se *successo*,  $\neq$  0 in *caso di errore*

In uno script bash, si accede al Valore di Ritorno dell'ultimo comando tramite la variabile speciale **\$?**

---

## Accedere ai parametri di riga del comando

Ricordare che i *script Bash* sono dei *programmi*, ergo possono usare i *parametri* forniti dalla riga del comando.

Si *ottengono con*:

- **\$1**, **\$2**, ...: *contenuti dei parametri*
- **\$0**: *nome dello script*
- **\$#**: *numero di argomenti*

**Esempio:** il nome dello script è **script.sh** e viene eseguito come **./script.sh ciao**.

```
#!/bin/bash
```

```
echo $0 # Stampa "./script.sh"
```

```
echo $1 # Stampa "ciao"
```

```
echo $# # Stampa "1"
```

---

## Variabili

---

### Concetti preliminari per le Variabili

#### REGOLE PER LA DENOMINAZIONE DELLE VARIABILI.

1. Sono una combinazione illimitata di lettere, numeri e underscore.
2. *Non* possono *cominciare con numeri* e sono *CASE sensitive*.

#### COMANDO READ PER LEGGERE LE VARIABILI DA STDIN.

Istruzione **read nomevar** (stesso compito di **scanf**)

SHELL

```
read nome
```

#### COMANDO ECHO PER STAMPARE LE VARIABILI SU STDOUT.

Si utilizza **echo** (stesso compito di **printf**)

SHELL

```
echo "Testo su schermo"
```

#### ACCESSO ALLE VARIABILI.

Si accede con **\$nomevar**.

SHELL

```
read x # Legge X da terminale
```

```
y=$x # Assegna a y il valore di x
```

```
echo $y # Stampa quanto letto
```

## I DUE TIPI DI VARIABILI.

I tipi principali sono solo i seguenti:

- **Stringhe:** `a="testo"`
- **Interi:** `a=47`

**Nota:** non inserire spazi prima e dopo `=` durante assegnazione

## Quoting per le stringhe

Le stringhe vanno racchiuse tra `"` o tra `'`.

Il carattere `\` indica il quoting. Permette di usare nella stringa *il carattere di quoting* (un'escape se vuoi)

### Esempio:

`a="ciao"` indica la stringa `ciao`

`a='ciao a tutti'` indica la stringa `ciao a tutti`

`a="ha detto: \"ciao\""` indica la stringa `ha detto: " ciao"`

### DIFFERENZE TRA QUOTING IN `'` E IN `"`.

Le stringhe definite con `"` possono contenere delle variabili che vengono valuate.

SHELL

```
a="test"
b="this is a $a"
c='this is a $a'
```

La variabile `b` contiene `this is a test`

La variabile `c` contiene `this is a $a`

**Esempio generale:** leggere due stringhe da tastiera e stamparle.

SHELL

```
read a
read b
echo $a $b
```

# Operazioni Matematiche con le Variabili

Solo numeri interi con segno (ovvero *non* esistono i *float*)

- Se si usano valori floating non segnala errore ma fa i *calcoli con numeri interi* (ovvero il risultato verrà *troncato*)

Operazioni ammesse: `+` `-` `*` `/` `%` `<<` `>>` `&` `^` (or esc.) `|`

## SINTASSI.

L'espressione `$(( var1 + var2 ))` restituisce la somma di due variabili

Alternativamente scrivere `$(( $var1 + $var2 ))` è equivalente (quindi il *dollaro* qui non è importante)

**ESEMPIO:** Si scriva un programma che legge due interi da tastiera e stampa il prodotto

SHELL

```
#!/bin/bash
read a
read b
c=$(( a * b ))
echo "Il prodotto è $c"
```

## Note:

Osservare la forma `c=$(( a * b ))`

Osservare la concatenazione naturale in `echo "Il prodotto è $c"`

---

## Strutture di controllo

---

### Struttura if-then-elif-else-fi

Le *condizioni* hanno forme

```

if condizione then
    ramo 1
elif condizione2 then
    ramo 2
else
    ramo alternativo
fi

```

Esistono molte *sintassi alternative* per esprimere le *condizioni*. Ne vedremo una parte.

## Condizioni tra Numeri

Si utilizza la sintassi **(( espressione ))** (*attenzione che qui NON c'è il dollaro!*)

Gli operatori di confronto sono i classici: **=** **≠** **<** **>** **<=** **>=**

### Esempio:

```

read n1 n2
if (( n1<n2 ))
then
    echo "$n1 minore di $n2"
elif (( n1==n2 )) then
    echo "$n1 uguale a $n2"
else
    echo "$n1 maggiore di $n2"
fi

```

## Condizioni tra Stringhe

Si utilizza la *sintassi* **[[ espressione]]**

Gli operatori di *confronto* sono:

- =** **≠**: uguaglianza o differenza (*ATTENZIONE! QUI SI USA UNA SOLA =*)
- >** **<**: ordinamento alfabetico
- z**: vero se la stringa è vuota (*o la variabile non è definita*; questa è unica per Bash);  
**!-z** è vero se la variabile non è vuota (o se è *definita*)
- E' *necessario* usare l'operatore **\$** e mettere *spazi tra operandi*

Esempio: **if [[ \$a ≠ \$b ]]**

Esempio: **if [[ ! -z \$var ]]**: vero se **var** esiste e non è vuota (questo è molto comune in Bash, dal momento che *non esistono* le eccezioni)

### Esempio:

SHELL

```
#!/bin/bash
read s1
read s2
if [[ $s1 = $s2 ]]
then
    echo "Le stringhe sono uguali"
else
    echo "Le stringhe sono diverse"
fi
```

## Condizioni su File

E' molto semplice testare *se un file (ovvero sui path) esiste, è vuoto o è una cartella*

- (*esiste?*): **-a path**: vero se **path** esiste
- (*è un file o una cartella?*):
  - **-f path**: vero se **path** è un file
  - **-c path**: vero se **path** è una cartella
- (*non è vuoto?*): **-s path**: vero se **path** non è vuoto
- (*ho i permessi?*)
  - **-r path**: vero se posso leggere **path**
  - **-w path**: vero se posso scrivere **path**
  - **-x path**: vero se eseguire/attraversare leggere **path**

**N.B.** Per le condizioni si usano le parentesi quadre **[[ ... ]]**, dato che si trattano di stringhe.

**Esempio:** si scriva un programma che legge due path da tastiera. Se sono uguali, controlla che il path corrisponda a una cartella. Se affermativo, stampa il path.

```
#!/bin/bash
echo "Inserisci il primo path:"
read s1
echo "Inserisci il secondo path:"
read s2
if [[ $s1 = $s2 ]]
then
    if [[ -d $s1 ]]
    then
        echo "$s1 è una cartella"
    else
        echo "$s1 non è una cartella"
    fi
else
    echo "Le due stringhe non sono uguali"
fi
```

## Operatori logici

Si possono creare *condizioni composte* con gli *operatori booleani*

- **&&**: and
- **||**: or
- **!**: not

Sintassi: **if condizione1 && condizione2**

Esempio: **if (( a>b )) && [[ \$c="hello" ]]**

**OSS.** Il Bash è un *linguaggio "lazy"*, ovvero nel senso che se abbiamo condizioni composte, verranno eseguite in una maniera più *"ottimale"*; nel senso che se, ad esempio abbiamo **if condizione1 && condizione2** e abbiamo che la *prima condizione* è *falsa*, allora Bash non verrà *mai controllato* (dunque eseguito). Vedremo a seguito come sarà utile questa caratteristica.

**OSS 2.** Ricordandoci che ogni programma deve *fornire un valore di ritorno*, posso utilizzare la precedenza osservazione per eseguire comandi *secondo una logica voluta*

- Ogni comando/programma avviato in bash fornisce alla shell/script chiamante un valore di ritorno
- Per convenzione un comando ritorna: 0 se successo,  $\neq 0$  in caso di errore
  - In bash, il valore 0 è interpretato come **true**; un valore  $\neq 0$  come **false**

- **NOTA:** diverso da altri linguaggio come C o Java!

### CONSEGUENZA: Utilizzo in espressioni di comandi

Ora, combinando il fatto che Bash è un *linguaggio lazy* e che i programmi *devono fornire un valore di ritorno*, abbiamo il seguente utilizzo di comandi.

Esegue **comando2** se **comando1** non dà errore

```
comando1 && comando2
```

SHELL

Esegue **comando2** se **comando1** dà errore

```
comando1 || comando2
```

SHELL

**Esempio:** eseguo **myprog** solo se la compilazione è andata a buon fine

```
gcc myprog.c -o myprog && ./myprog
```

SHELL

**Esempio:** eseguo un gestore dell'errore **gestione\_err** se un'istruzione **istruzione** non è andata a buon fine

```
istruzione || gestione_err
```

SHELL

---

## Cicli **for**

Abbiamo *due modi* per esprimere un *ciclo* **for** in Bash.

### 1) Versione semplice

Scelgo un *"iterabile"* (una successione di numeri oppure una stringa) e ci itero, come se fossimo su Python



SHELL

```
for n in 1 2 3 4
do
  echo "valore di n = $n"
done
```

SHELL

```
for nome in mario giuseppe vittorio
do
  echo "$nome"
done
```

## 2) Versione completa

Questa sintassi si dice *"stile-C"*.

### Sintassi:

SHELL

```
for ((INITIALIZATION; TEST; STEP))
do
  [COMMANDS]
done
```

### Esempio:

SHELL

```
for ((i = 0 ; i < 1000 ; i++))
do
  echo "Counter: $i"
done
```

---

## Cicli **while**

### SINTASSI:

```
while condizione
do
...
done
```

### ESEMPI:

```
n=0
while ((n<4))
do
    ((n=n+1))
    echo $n
done
```

```
n=0
until((n>4))
do
    ((n=n+1))
done
```

---

## Esercizi Bash

---

1. Si scriva un programma che riceve due argomenti. Se entrambi sono dei file, stampa il contenuto di entrambi
2. Si scriva un programma che per ogni file/cartella nella cartella corrente dice se esso è un file o una cartella.
3. Si scriva un programma che riceve un intero come argomento. Se esso è minore di 10, crei i file **0.txt**, ..., **9.txt**

## Soluzioni agli Esercizi

1. Esercizio 1

```
#!/bin/bash
if [[ "$#" ≠ "2" ]]
then
    echo "Servono due argomenti"
else
    if [[ -f $1 ]] && [[ -f $2 ]]
    then
        cat $1
        cat $2
    else
        echo "Non sono due file"
    fi
fi
```

## 2. Esercizio 2

```
#!/bin/bash
for file in *
do
    if [[ -f $file ]]
    then
        echo "$file è un file"
    elif [[ -d $file ]]
    then
        echo "$file è una cartella"
    fi
done
```

## 3. Esercizio 3

```
#!/bin/bash
if [ "$#" != "1" ]
then
    echo "Serve un argomento"
else
    if (( $1 < 10 ))
    then
        for (( i=0; i<$1; i++))
        do
            touch $i.txt
        done
    else
        echo "L'argomento non è minore di 10"
    fi
fi
```

Dataview (inline field '='): Error:

```
-- PARSING FAILED -----
-----
```

```
> 1 | =
   | ^
```

Expected one of the following:

```
('', 'null', boolean, date, duration, file link, list ('[1,
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),
string, variable
```

## u2-s4-comandi-bash

### Unità 2: Utilizzo di Linux

## Comandi in Bash

## Argomenti

1. Pipe e redirect
  2. Filtri e simili
  3. Esercizi
- 

## Pipe e Redirect

---

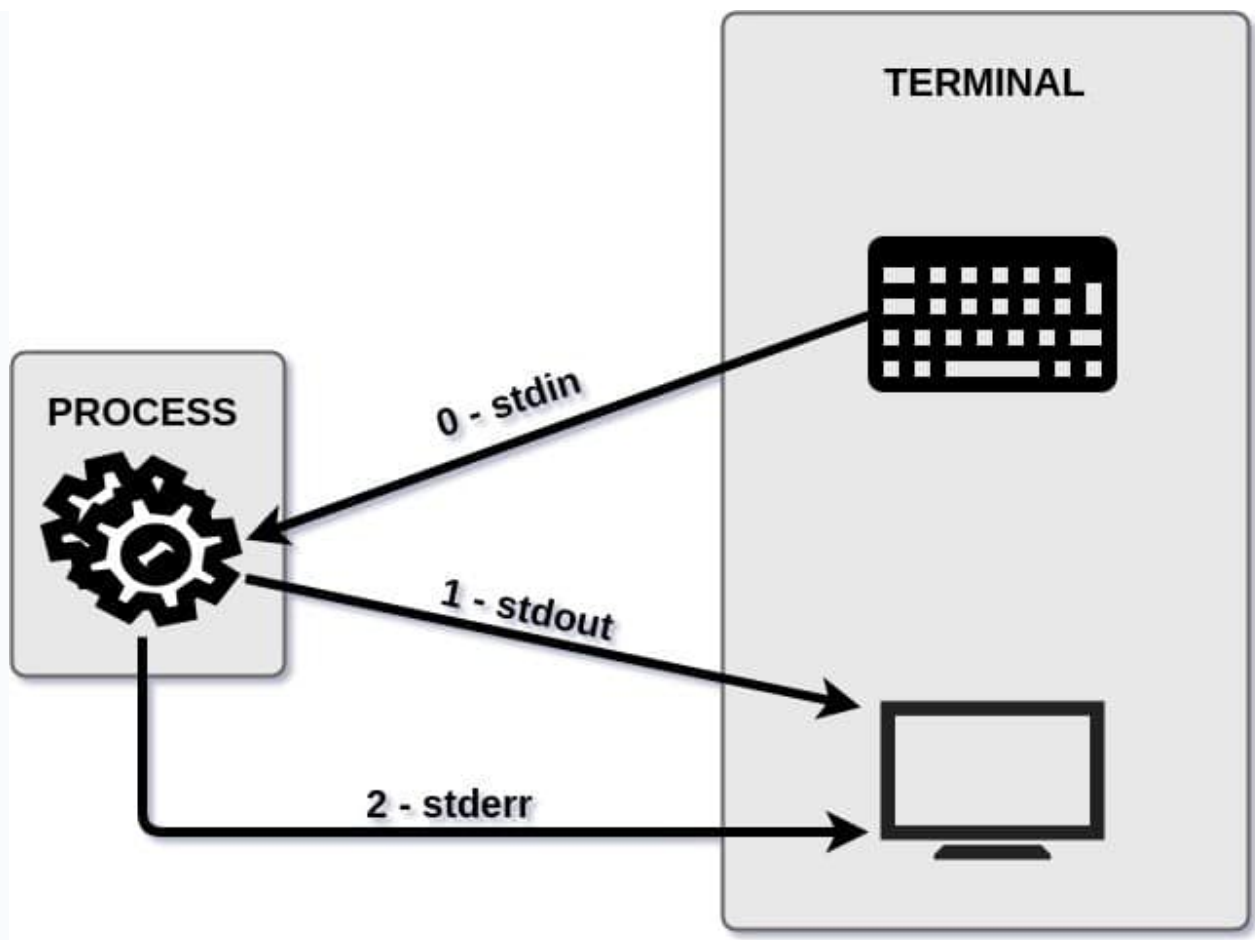
### I standard di comunicazione in Linux

**NOZIONE PRELIMINARE.** In Linux, ogni processo, ha a disposizione *3 canali standard di comunicazione* con l'*esterno* (ovvero i dispositivi I/O).

- **Standard Input** (**stdin**): per ricevere dati in ingresso.
  - **Standard Output** (**stdout**): per stampare l'output
  - **Standard Error** (**stderr**): per stampare eventuali errori
- Questa convenzione è *caratteristica* per i sistemi *POSIX-like* ([Storia e Definizione di Linux > ^1ad28a](#)).

Di default, un programma riceve lo Standard Input da tastiera, e stampa Standard Output e Standard Error su console.

**FIGURA: Illustrazione grafica dei canali di comunicazione**



**CONSEGUENZE.** Questo implica quello che già abbiamo visto:

- **read** legge da **stdin**, quindi *di default da tastiera* (quindi non *direttamente* dalla *tastiera*!)
  - **echo** stampa su **stdout** che *di default è console* (quindi non *direttamente* sullo *schermo*!)
  - Per stampare su **stderr**, si può usare: **echo "An error!" >&2**. Di default, lo **stderr** è visualizzato a schermo
- Tutti i programmi ben scritti, devono attenersi a usare questi *canali standard*.
- Ciò permette una *grande flessibilità* (flessibilità)
  - *Tutti* i programmi di default di Linux *lo fanno* (universalità)

## Redirezionare i canali (redirect)

**1. Redirezione **stdout** su file:** è possibile eseguire un programma e *redirezionare* lo **stdout** su file anziché stamparlo sul terminale (come di default)

- **Formato:** **comando > file** oppure **comando 1> file**
- Questo perché 1 indica **stdout** mentre 2 indica **stderr**

**Esempio:** **date > data.txt** La data corrente viene salvata in **data.txt** e non stampata ad output

**Nota:** se **file** esiste, il contenuto viene sovrascritto, a meno che si usa la *modalità append*.

**2. Append **stdout** su file:** simile alla *redirezione*. Il file non viene cancellato, ma lo **stdout** del programma *viene aggiunto in coda*.

- **Formato:** **comando >> file** oppure **comando 1>> file**

**Esempio:** Scrivere su un file la data due volte, con un intervallo di 5 secondi

SHELL

```
date > file.txt
sleep 5 # Pausa di 5 secondi
date >> file.txt # "Appende" a file.txt
```

**Esempio:** Si scriva un programma che riceve due argomenti. Ricerca nella folder corrente tutti i file che hanno il nome del primo argomento e salva la lista nel file il cui nome è il secondo argomento

SHELL

```
#!/bin/bash
if (( "$#" ≠ "2" ))
then
    echo "Servono due argomenti"
else
    find . -name $1 > $2
fi
```

**3. **stderr** su file:** permette di redirigere lo **stderr** su un file.

- **Formato:** **comando 2> file**
- Questo perchè 1 indica **stdout** mentre 2 indica **stderr**

**4. **stdin** da file:** permette di prelevare da file anziché da tastiera lo **stdin** un programma; quindi si *"dirotta"* il flusso di **stdin**, rimpiazzando lo *default* (ovvero la tastiera) con il *file*.

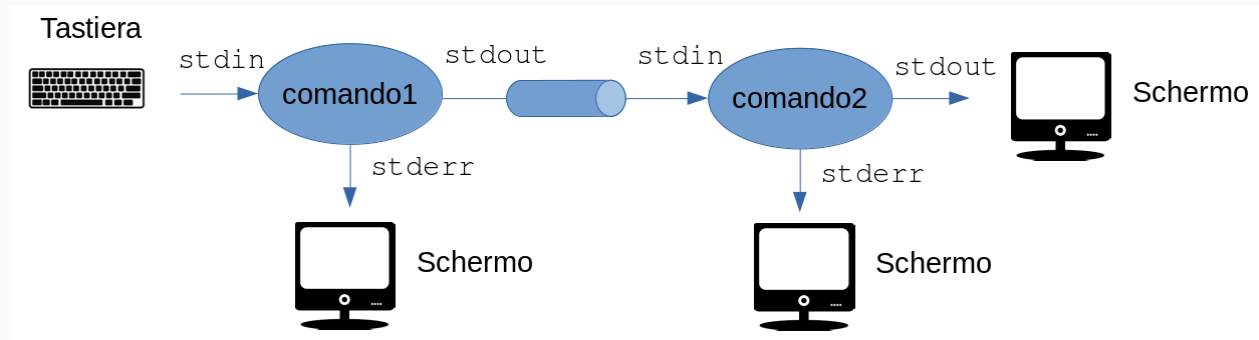
- **Formato:** **comando < file**

**5. Pipe:** è possibile redirezionare lo **stdout** di un primo comando nello **stdin** di un secondo. Ovvero avrà una situazione del tipo *...→comando1→stdout→stdin→comando2->....* In essenza, sto *concatenando* due comandi, creando così una *catena di processi*.

- **Formato:** **comando1 | comando2**

- **NOTA.** E' uno dei costrutti più potenti del bash, siccome permette di svolgere *compiti complessi con poco codice*: ne vedremo le potenzialità assieme ai comuni comandi bash

### FIGURA: Illustrazione del concetto di pipe



**6. Sostituzione in variabile:** è possibile usare lo **stdout** di un comando come una variabile.

- **Formato:** **`$(comando)`** oppure **``comando``**

#### Esempi:

- **`a=$(ls /tmp)`**: la stringa **`a`** contiene l'elenco dei file in **`/tmp`**
- **`rm $( find / -name "*.tmp" )`**: rimuove tutti i file nel sistema che terminano per **`.tmp`**

---

## Filtri e simili

---

### L'idea di filtrare testi

**L'IDEA PRELIMINARE.** Negli OS Linux esistono una serie di comandi per manipolare testo.

- Filtrare, ordinare, comporre in una *maniera arbitraria*

Essi si aspettano di lavorare *su dati testuali organizzati in righe*, come normali file di testo (o di configurazione) (o file *.csv*)

Permettono di fare operazioni complesse con *poco codice*

- Spesso si usano assieme alle **pipe** al fine di creare *pipeline di processing*

### Comando **grep**

Il comando **grep** è uno dei comandi *utili per filtrare* dei *contenuti*.



**grep [options] pattern [file...]**: stampa le linee del file che contengono il *pattern*. Se non metto il file usa lo standard input: posso usare grep in pipe. Alcune opzioni:

- **-n**: stampa il numero di riga
- **-i**: case insensitive
- **-c**: stampa il numero di match
- **-v**: stampa solo le linee che *non* contengono il pattern
- **-e**: interpreta i pattern come delle *espressioni regolari*

**Esempio dell'utilizzo di grep:**

- **grep main \*.c**: stampa le linee che contengono **main** in tutti i file che finiscono in **.c**
- **ps -ef | grep bash**: stampa tutti i processi che sono istanze del programma **bash**  
Osserviamo che ci sono dei *caratteri speciali* che vengono processati in qualche modo da Bash. Questo è *Bash Expansion*.

## Bash Expansion

**OSS.** Osservare il comando **grep main \*.c**:

- E esso ricerca il pattern **main** in *tutti* i file che terminano con **\*.c**

La *bash* (quindi NON il comando/programma grep!) *espande* il termine **\*.c** in tutti i file che matchano l'espressione **prima** di eseguire il comando

- **grep** non riceve la stringa **\*.c** come argomento
- **grep** riceve già la lista di file

**Esempio:** la cartella corrente contiene **prog.c** e **module.c**

- Il comando **grep main \*.c**
- Viene trasformato dalla bash nel comando **grep main module.c prog.c**

Questo è un *meccanismo* flessibile per operare su file:

- Agisce *prima* di avviare il comando
- Il testo non deve essere quotato né con **\*** né con **'**
- **\*** matcha qualsiasi numero di ogni caratteri (*wildcard*)
- **?** matcha un solo carattere (*single wildcard*)
- **~** rappresenta la home directory:
  - **~/file.txt** equivale a **/home/martino/file.txt** se l'utente è **martino**
  - **NOTA!** **.** e **..** non vengono espansi, *sono propriamente parte di un path*
- Liste racchiuse tra **{ ... }** vengono espanse
  - **mkdir /tmp/{dir1,dir2}** viene "espanso" in **mkdir /tmp/dir1 /tmp/dir2** (attenzione che in realtà *non sono equivalenti*, tuttavia hanno lo stesso effetto).

**Esempio:** si scriva un programma che riceve due argomenti: il primo argomento è una cartella, il secondo argomento un pattern. Il programma trova tutte le linee dei file `.c` o `.h` nella cartella, che contengono il pattern. Le linee vengono salvate nel file `/tmp/output.txt`.

SHELL

```
#!/bin/bash
if (( "$#" != "2" ))
then
    echo "Servono due argomenti"
else
    cat $1/*.c $1/*.h | grep $2 > /tmp/output.txt
fi
```

## Comando `cut`

**cut**: estrarre colonne (o campi) dall'input. Ha *diverse modalità*, di cui ne vedremo *due*; modalità *byte* e modalità *campi*.

- **Modalità byte**: estrae i byte specificati da ogni riga. Si utilizza l'opzione `-b byterange`
- **Modalità campi**: estrae i campi specificati, delimitati da un separatore specifico. Si utilizza l'opzione `-d delimitatore -c campi`. Questo è particolarmente utile per i file `.csv`.

**Esempio:** il file `file.txt` contiene:

```
luca 1985 milano
martino 1990 torino
```

`cat file.txt | cut -b1-2` estrae i primi *2 byte* (caratteri) da ogni riga, e stampa su **stdout**:

```
lu
ma
```

`cat file.txt | cut -d " " -f 2` estrae il *secondo campo del file*, delimitato da uno spazio. Stampa su **stdout**:

1985

1990

---

## Comando **tr** (translate)

**tr [-c ds] [set1] [set2]**: legge dei dati e *sostituisce* i caratteri specificati con altri caratteri (quindi fa una specie di *"translate"* automatica). Opzioni comuni:

- **-d**: (del) *cancella* tutti i caratteri specificati. E' necessario *un solo set* come argomento
- **-s**: *sostituisce* le ripetizioni del carattere specificato *con un solo carattere*

**Esempio:** **tr a A < file1 > file2**: sostituisce le **a** minuscole con **A** maiuscole. Notare lo **stdin** di **tr** è letto da file con l'operatore **<**. Qui si ha una situazione del tipo *(file1→tr)->file2*

---

## Comando **sort** (ordinamento)

**sort [-dfnru] [-o outfile] [file...]**: Ordina i dati del file o dello **stdin**. *Di default* si usa l'ordinamento *alfabetico*. Opzioni principali:

- **-f**: tratta maiuscole come minuscole (*case insensitivity*).
- **-n**: riconosce i numeri e li ordina in modo *numerico* (*numerical*).
- **-r**: ordina i dati in modo inverso (*reverse*).
- **-k**: ordina secondo il numero di colonna dato dopo il k (*su file a campi*)
- **-t SEP**: usa un separatore di campo diverso da quello di default (una *non-blank to blank transition*)
- **-u**: ordina e rimuove linee duplicate

**Esempio:** il file **file.txt** contiene:

```
luca 1985 milano
martino 1990 torino
giovanni 1971 trieste
```

**sort < file.txt > sorted.txt** ordina le righe e stampa nel file **sorted.txt**, che conterrà:

```
giovanni 1971 trieste
luca 1985 milano
martino 1990 torino
```

**cat file.txt | sort -k 2 -n** ordina le righe *per anno* (secondo campo) e stampa su **stdout**:

```
giovanni 1971 trieste
luca 1985 milano
martino 1990 torino
```

---

## Comando **uniq** (duplicati)

**uniq [-cdu]**: *esamina* i dati linea per linea *cercando linee duplicate* e può:

- Di default *elimina duplicati*
- **-c** per ogni riga prepende il *numero di occorrenze* (*dice quanti ce n'erano*)
- **-d** stampa solo le linee duplicate (*solo duplicate*)
- **-u** stampa solo le linee uniche (*solo uniche*)

**NOTA.** Il comando **uniq** non ordina le righe. E' necessario fornirle già ordinate!

## Comando **wc** (conteggio parole)

**wc [-lwc] [file]**: conta linee (**l**), parole(**w**) e caratteri(**c**) dello **stdin** o del file

**Altri comandi utili** (non per esame):

- **sed**: ricerca e sostituzione di *espressioni regolari*
- **awk**: esecuzione di script (stile C) sulle righe di un file
- **comm**: trova le linee in comune (uguali) tra due file
- **paste**: concatena le linee di più file
- **rev**: scrive l'input in ordine inverso di caratteri, linea per linea

---

## Esercizi

---

## Esercizi

Dato il file **vini.txt** contenente il nome, l'anno, la città e il prezzo di alcune bottiglie di vino:

```
ribolla 2012 udine 21
prosecco 2018 trieste 15
barbera 2009 torino 20
freisa 2010 torino 18
barbera 2013 torino 14
barolo 1984 alba 45
```

Si trovino il nome e l'anno del vino più caro:

SHELL

```
$ sort -k4 -r < vini.txt | head -1 | cut -d " " -f 1-2
barolo 1984
```

Si trovino i nomi dei vini prodotti a Torino:

SHELL

```
$ cat vini.txt | grep torino | cut -d " " -f 1 | sort | uniq
barbera
freisa
```

---

## Esercizi

Utilizzando lo stesso file dell'esercizio precedente:

1. Si calcoli quanti vini sono presenti per ogni città:

SHELL

```
$ cat vini.txt | cut -d " " -f 3 | sort | uniq -c
1 alba
3 torino
1 trieste
1 udine
```

2. Si calcoli quanti anni passano tra il vino più vecchio e più nuovo:

SHELL

```
$ min=$(sort -k2 < vini.txt | cut -d " " -f 2 | head -n 1)
$ max=$(sort -k2 < vini.txt | cut -d " " -f 2 | tail -n 1)
$ echo "Intercorrono $((max-min)) anni"
Intercorrono 34 anni
```

---

## Esercizi

Dato il file **file.txt** contenente:

```
luca 1985 milano
martino 1990 torino
giovanni 1971 trieste
andrea 1984 milano
```

Si calcoli il numero di righe nel file:

SHELL

```
wc -l < file.txt # Output 4
```

Si calcoli quante città sono incluse nel file:

SHELL

```
cat file.txt | cut -d " " -f 3 | sort | uniq | wc -l # Output 3
```

Si trovi la città che appare il maggior numero di volte e il numero di occorrenze:

SHELL

```
cat file.txt | cut -d " " -f 3 | sort | uniq -c | sort | tail -n 1 # Output 2 milano
```

*Ricorda:* il comando **tail -n N** stampa le ultime **N** righe di un file o dello **stdin**

---

## Esercizi

Si crei un programma che cerca ricorsivamente tutti i file presenti in una cartella passata come primo argomento.

Collochi quei file in una cartella ricevuta come secondo argomento, suddividendoli in sottocartelle separate per estensione.

**Nota:** si assuma che i nomi di cartelle non contengano `.` e i file ne contengano uno solo, nella forma **nome.estensione**

SHELL

```
#!/bin/bash
```

```
if (( $# ≠ 2 ))
then
    echo "Usage: $0 indir outdir"
    echo "    The program assumes directories do not contain '.'"
    echo "    and files contain one"
    exit 1 # Signal error to the caller
fi

for f in $( find $1 -type f )
do
    folder=$2/$( echo $f | cut -d . -f 2 )
    mkdir -p $folder
    cp $f $folder
done
```