Appunti del corso di:

Introduzione alle tecniche informatiche per la fisica

Gabriele Cembalo

A.A. 2025-2026

Università degli Studi di Torino
Dipartimento di Fisica
Via Giuria, 1, Torino (TO)

Informazioni legali

Questo materiale è una rielaborazione personale del corso di **Introduzione alle tecniche informatiche per la fisica**, tenuto dal **Prof. Stefano Trogolo** presso **l'Università degli Studi di Torino**.

Il contenuto riportato non rappresenta materiale ufficiale del docente né dell'università, e può contenere interpretazioni soggettive o errori. Tutti i diritti su slide, dispense o altri materiali forniti dal docente restano riservati ai rispettivi autori e non sono inclusi in questi appunti. Questi appunti sono condivisi a solo scopo didattico e divulgativo, senza fini di lucro, e sono destinati a supportare lo studio personale degli studenti.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

Prefazione

In questo documento voglio raccorgliere le mie note rispetto gli appunti relativi al corso di "**Introduzione alle tecniche informatiche per la fisica**" svolto dal professor S. Trogolo e seguito all'*Università degli studi di Torino* nell'a.a. 2025-2026 aggiungendo eventualmente i riferimenti a vari libri (più o meno utili a seconda della volontà di approfondire). Questi appunti sono una riscrittura degli appunti presi in aula, quindi la fonte principale sono le note del/la professore/ssa, ma è stato anche indicato un testo [1] che si potrebbe seguire e scaricabile al link.

Il corso nasce dalle difficoltà degli studenti emerse durante diversi corsi di calcolo nelle diverse lauree magistrali, i quali richiedono l'uso o sviluppo di piccoli programmi. Il corso si propone di fornire un'introduzione all'utilizzo e la gestione di OS linux-based e all'interfaccia tramite comandi di linea. Nel capitolo §10 ho scritto un riassunti dei comandi visti nel corso.

Chiaramente sono da intendere come degli appunti personali scritti in bella, eventuali sviste, errori o inesattezze sono dovute alla mia ignoranza, ma soprattuto ho scritto questi appunti in modo da *"spiegare"* a me stesso l'argomento, quindi alcune parti potrebbero sembrare troppo prolisse o troppo superficiali per alcuni. In ogni caso fa piacere se possono aiutare qualcun'altro. Spero in ogni caso di esser riuscito a scrivere un documento chiaro e ben strutturato.

Alcune volte posso non far riferimento ad un particolare testo o corso passato, in questi casi mi sto riferendo ai MIEI appunti riguardanti quell'argomento. Una mia collezione di appunti è presente nella mia pagina personale di GitHub: gCembalo.github.io.

Qualsiasi errore/refuso può essere inviato alla mia mail personale:
gabriele.cembalo02@gmail.com.

Ultimo aggiornamento: 30/09/2025

Indice

Capitolo 1

Introduzione

Daremo in questo capitolo una breve introduzione al corso, che sarà molto simile ad una carrellata di informazioni, comandi ed esempi da utilizzare direttamente sul proprio pc.

1.1 Kernel

Per prima cosa dobbiamo capire che cosa intendiamo quando parliamo di kernel. Immaginiamo di recarci in un negozio di elettronica e di acquistare un nuovo pc. Cosa abbiamo comprato? Ovviamente teniamo tra le mani l'*hardware*, ovvero la parte fisica del pc, quello che possiamo toccare con le mani, e poi sappiamo che appena clicchiamo il tasto ON vedremo aprirsi varie finestre, gestite dal *software* che è l'insieme dei programmi, delle istruzioni e dei dati intangibili.

Però potremmo chiederci come facciano a parlarsi la parte fisica e i programmi del software. È proprio il **kernel** che permette di collegare queste due parti. Infatti, in informatica, un kernel costituisce il nucleo di un sistema operativo (OS) e ciò che fa è:

- Fornisce ai processi in esecuzione sul computer un accesso sicuro e controllato dell'hardware.
- Assegna il tempo-macchina (scheduling).
- Da l'accesso all'hardware a ciascun programma (multitasking).

1.1.1 Classificazione

Esistono in commercio diversi tipi di kernel, che si differenziano sostanzialmente dalla quantità di libertà operativa lasciata all'utente - la cui consapevolezza dev'essere proporzionale alla libertà di movimento all'interno del kernel. Potremmo trovare:

- **Kernel monolitico**: implementa direttamente una completa astrazione dell'hardware sottostante; è l'unico responsabile della gestione della memoria, dei processi e della comunicazione tra processi, oltre a fornire funzioni per il supporto di driver e hardware.
- **Microkernel**: fornisce un insieme ristretto e semplice di astrazione dell'hardware e usa device driver (o server) per fornire maggiori funzionalità.
- **Kernel ibrido**: si differenzia dai microkernel puri per l'implementazione di alcune funzioni aggiuntive al fine di incrementare le prestazioni.
- **Esokernel**: rimuove tutte le limitazioni legate all'astrazione dell'hardware e si limita a garantire l'accesso concorrente allo stesso.

Puoi vedere nella figura 1.1 una rappresentazione delle differenze tra kernel monolitico, microkernel e kernel ibrido.

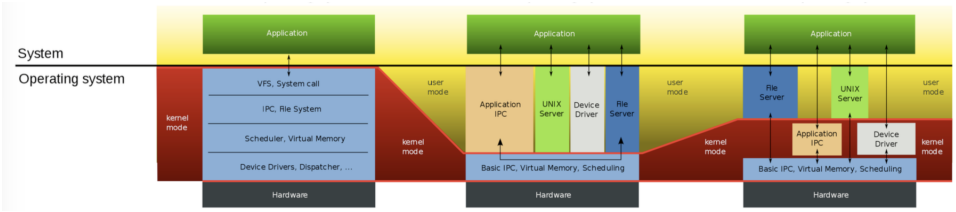

Figura 1.1: Differenze tra kernel monolitico, microkernel e kernel ibrido.

1.2 Breve storia di Linux

Linux è un kernel monolitico creato nel 1991 da Linus Torvalds e derivato da UNIX. È distribuito sotto licenza di software libero GNU GPLv2. GNU/-Linux (ossia quello a cui ci riferiamo quando diciamo Linux) è una famiglia di OS free e open source di tipo Unix-like, che hanno la caratteristica di utilizzare il kernel Linux.

Quando diciamo **Linux** potremmo riferirci a molte cose, ma solitamente si fa riferimento al kernel. Infatti, tutto il resto, compreso software e graphical desktop, costituisce il sistema operativo. Dunque, non è propiamente corretto parlare di sistemi Linux, piuttosto dovremmo dire *sistemi Linux-based.*

Nota anche che il kernel Linux insieme a componenti aggiuntive, quali librerie, graphical environment etc. costituiscono una **Linux distribution**. Le particolari *famiglie di distribuzioni* sono:

- **Red Hat**: RHEL, CentOS, Fedora.
- **Debian**: Debian, Ubuntu, Linux Mint.
- **SUSE**: SUSE, OpenSUSE.

Le più utilizzate sono CentOS e OpenSUSE per quanto riguarda Big Tech e Ubuntu e Linux Mint per la ricerca scientifica. Una lista completa delle distribuzioni è reperibile al link.

1.3 Intallare Linux

In giro per il web sono presenti moltissime guide per l'installazione di Linux. Le opzioni più utilizzate sono:

- Dual boot installation: sostanzialmente fornisce la possibilità di scegliere tra due (o più) sistemi operativi al momento dell'avvio. È la scelta più solida a lungo termine.
- Virtual Machine: si installa sul proprio pc un software che permette di emulare il comportamento di una macchina fisica attraverso l'assegnazione di risorse hardware. Se vuoi ne ho scritto una breve guida nella mia pagina personale.
- Windows Subsystem for Linux (WSL): è un layer di compatibilità che permette l'esecuzione dei binari Linux in modo nativo su Windows senza una macchina virtuale.

1.4 Consigli di naming

In questa rapida sezione voglio raccogliere pochi consigli di naming di files e directories:

- Assegnare nomi *meaningful*.
- Nomi troppo complicati o poco intuitivi rendono complicato lavorare con command line.
- Non utilizzare spazi, piuttosto utilizzare dash (-) o underscore (_). Attenzione però che i dash non possono stare davanti, poiché come vedreme saranno una caratteristica delle opzioni.
- Non utilizzare caratteri speciali che potrebbero essere dei comandi nella command line.

1.5 Coding best practices

In analogia alla sezione precedente voglio presentare alcuni buoni consigli per un buon coding. Prima però vorrei chiarire dove si scrive il codice, ovvero nell'**editor di testo**. Parliamo di editor di testo quando ci riferiamo a programmi che lavorano con dati di carattere semplice, dunque no tabelle, immagini o altri human-friendly media. Uno dei più famosi editor di testo più utilizzati è VScode.

I buoni consigli di coding sono:

- Scrivere meno righe possibile.
- Buona convenzione di nomi.
- Segmentare opportunamente il codice. Conviene sempre creare più funzioni diverse che fanno operazioni diverse, piuttosto che avere un unica funzione lunghissima.
- Indentazione.

Capitolo 2

La Unix shell

Cominciamo a questo punto ad entrare un po' nel vivo del corso e parliamo di cos'è la **shell**. Quando si accende un pc si può interagire con esso in diversi modi, il più classico e comune è tramite tastiera, mouse, touch screen etc. Più in generale parliamo di **Graphical User Interface (GUI)**. Il problema di questo metodo che è tanto intuitivo quanto soggetto ad errori umani e a perdite di tempo nel momento in cui si devono fare task ripetitivi.

Per questo ci viene in soccorso la **Unix shell**, o meglio la **command-line interface (CLI)** e scripting language. La shell è un'interfaccia utente di tipo testuale in cui si possono dare dei comandi da eserguire al computer. Ad esempio possiamo lanciare un complicato programma di simulazione di collisioni adroniche, ma in egual maniera possiamo semplicemente creare una cartella vuota. La Unix shell più popolare è **Bash** (Bourne Again SHell, inventata da Stephen Bourne), che si trova di default in molte implementazioni moderne di Unix.

Ovviamente siamo tutti abituati all'interfaccia grafica del computer ed imparare ad usare la shell richiede impegno e tempo. Una volta presa la mano con la linea di comando la GUI risulterà ancora più facile, ma ci renderemo conto che ci presenta le scelte da selezionare ma è limitata. In CLI dobbiamo sapere noi che cosa vogliamo fare e dobbiamo conoscere i comandi (i.e. vocabolario), però allo stesso tempo pochi potenti comandi ci permettono tante operazioni e sequenze di comandi possono essere combinate negli script (concede ripetibilità e gestione di grandi moli di dati).

2.1 Piccola nota per MacOS

Se possedete un pc che monta il sistema operativo MacOS, allora probabilmente avrete come shell di default **zsh**. Nel corso utilizziamo la shell **bash**, ma il passaggio per passare dall'una all'altra è molto semplice, basta aprire una finestra del terminale e scrivere:

```
exec bash
```

per passare da zsh a bash. Allo stesso modo possiamo tornare indietro e passare da bash a zsh con:

```
exec zsh
```

Il problema di questa cosa è che nel momento in cui si chiude la finestra di lavoro la shell torna quella di default. Per rendere definitiva la schelta della shell occorre lanciare il seguente comando:

```
chsh -s /bin/bash
```

per rendere bash definitivo, e:

```
chsh -s /bin/zsh
```

per settare zsh.

Si faccia riferimento alla discussione su stackoverflow.

2.2 Prompt

Se proviamo ad aprire il terminale sul nostro pc ci comparirà una cosa del genere:

```
$
```

con eventualmente il cursore che ci indica dove stiamo scrivendo. Il simbolo $ lo chiamiamo **prompt** e la shell mostrandocelo ci sta dicendo che è pronta a ricevere comandi.

Ricorda che le diverse shell possono utilizzare prompt diversi. In ogni caso il prompt può essere modificato sia in maniera definitiva sia in temporanea.

2.2.1 Modifiche permanenti

Per modificare il prompt in modo permanente dobbiamo agire sul file `.bashrc`. Tale file non è presente di default nel nostro computer, ma dobbiamo crearlo noi in modo da poter modificare la shell bash. Per prima cosa, dopo aver aperto il terminale, creiamo un file vuoto di nome `.bashrc` eseguendo il comando:

```
touch ~/.bashrc
```

dopodiché possiamo eseguire:

```
open -e ~/.bashrc
```

per aprirlo con un editor di testo e modificarlo. Scrivendo all'interno di `.bashrc` il comando:

```
PS1="MyTestPrompt > "
```

e facendo il `source` del `.bashrc`, ovvero eseguire:

```
source ~/.bashrc
```

possiamo notare che il prompt della nostra shell risulta:

```
MyTestPrompt >
```

2.2.2 Modifiche temporanee

Per modificare il prompt solo temporaneamente possiamo eseguire il comando:

```
export PS1="MyTestPrompt > "
```

2.2.3 Customisation

A prescindere del metodo che si utilizza per modificare il prompt dei comandi si possono utilizzare specifici comandi per ottenere determinati risultati. Faccio un rapido elenco:

- `"\u"` → only username.
- `"\H"` → full hostname.
- `"@ $"` → add special character $.
- `"\s\v"` → shell name and version character.
- `"\d"` → today's date.
- `"\t"` → time 24-hour format.
- `"\T"` → time 12-hour format.
- `"\A"` → time 24-hour format, just hour and minutes.

- `"\W"` → hide everythig.

Questi comandi vanno inseriti all'interno delle virgolette del comando `PS1`, ad esempio:

```
export PS1="\u >\s\v "
```

2.3 Sintassi

La sintassi generale delle shell prevede che i comandi siano tutti intervallati da degli spazi. La struttura generica di un comando è:

```
$ ls -F /
```

in cui $ è il prompt, `ls` è il comando, `-F` l'opzione di esso e `/` l'argomento. Nota che le opzioni possono essere *short option*, scritte con un solo dash (-), oppure *long option* scritte con due dash (–). Quest'ultime si scrivono per esteso; vedi ad esempio l'opzione `--help`.

Le short option sono utilizzate per velocizzare le operazioni, mentre le long option per maggiore chiarezza di codice.

Delle volte, soprattutto i fisici pigri, non si ha molta voglia di scrivere e per questo ci viene in aiuto la **tab completion**. Iniziando a scrivere un comando e premendo Tab sulla tastiera il terminale scriverà per noi il resto. Se premendo Tab non succede nulla vuol dire che ci sono possibilità multiple; premendo due volte consecutive Tab compariranno tutte le scelte possibili.

2.4 Shortcut per la navigazione tra i comandi

Una volta che eseguiamo un comando possiamo andarlo a rivedere e rieseguirlo utilizzando le frecce direzionali. In più se eseguiamo il comando `history` esso restituisce la lista delle ultime centinaia di comandi eseguiti; una volta che individuiamo il comando che ci interessa possiamo digitare `!123` e verrà eseguito il comando numero 123.

2.5 Esercizio

Per esercitarti puoi provare a customizzare il `prompt` in modo che mostri:

- solo username
- full hostname
- aggiungete un carattere speciale

- username, shell name e versione
- la data di oggi
- l'ora in formato 24 ore e 12 ore
- l'ora in formato 24 ore nascondendo i secondi
- nascondendo tutto

Capitolo 3

Navigating files and directories

Vediamo in questo capitolo come navigare in cartelle e files tramite comandi di linea.

3.1 Relative and absolute path

Per raggiungere ogni directory del nostro pc, ovviamente, la raggiungiamo facendo un certo percorso che viene chiamato **path**. Il path di un file o directory può essere *assoluto* o *relativo*. La differenza tra i due diversi path è il punto da cui partono: l'absolute path comincia dalla root directory, mentre il relative path dalla current working directory. È buona pratica utilizzare l'absolute path per avere maggiore trasparenza nei casi in cui il codice si in collaborazione con altri users.

Una piccola shortcut è che la shell vede il carattere tilde $\sim$ all'inizio di un path come *the current user's home directory*, per cui può essere utilizzato per velocizzare alcune scritture.

3.2 File system

La parte del sistema operativo responsabile della gestione di files e directories si chiama **file system**. Permette di organizzare i nostri dati nei file, organizzare i file nelle directories (anche chiamate *folders*) e di gestire la struttura directories-subdirectories.

Molti comandi sono spesso usati per creare, ispezionare, rinominare, modificare e cancellare files e directories; ad esempio ne abbiamo usati un paio per creare e aprire il file `.bashrc`. Ci sarà tempo per conoscere i comandi uno per uno e ne capirne l'utilizzo.

Apriamo il terminale del nostro pc, se scriviamo un qualsiasi comando, ad esempio creare un file vuoto, dove viene posizionato? La shell esegue i

comandi che gli vengono dati nel posto in cui ci troviamo in quel momento. Ora il punto sarebbe capire come vediamo dove ci troviamo. Se proviamo ad eseguire il comando `pwd` (print working directory) la risposta che ci viene data è il percorso (path) assoluto della directory in cui ci troviamo.

```
pwd
```

Solitamente quando apriamo il terminale ci troviamo nella home directory. Per capire cos'è la home directory possiamo guardare all'intera struttura del file system; vedi figura 3.1.

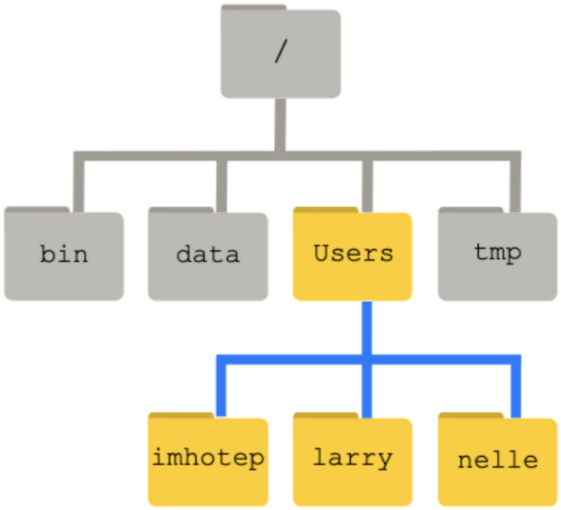

Figura 3.1: Struttura ad albero rovesciato del file system.

La cartella più in alto "/" è la **root directory** ed è colei che contiene tutto. La directory `bin` è dove sono salvati tutti i built-in programs, `data` dove ci sono i file di dati vari, in `tmp` ci sono i file temporanei che non serve storare a lungo e `Users` contiene le directory personali degli Users.

Tieni a mente che se "/" è all'inizio di un file o di una directory, allora si riferisce alla root directory, mentre se è all'interno di un path, allora è semplicemente un separatore.

Le varie cartelle `imhotep`, `larry` e `nelle` sono le rispettive home directory dei vari account. Dunque, se Larry accende il computer, entra nel terminale ed esegue `pwd`, allora il path che gli risulterà sarà:

```
/Users/Larry
```

Prestiamo attenzione che in base al sistema operativo che abbiamo possiamo avere il percorso della home directory leggermente in modo differente.

3.3 Errori

Vedremo nelle prossime sezioni una serie di comandi eseguibili a terminale. Potrebbe capitare che ci esca il seguente messaggio:

```
comm : command not found
```

dove `comm` è un qualsiasi comando abbiamo provato ad eseguire. In queste situazioni possono essere successe diverse cose. Potremmo aver scritto in maniera errata il comando, potrebbe essere che il comando non sia ancora installato oppure che il comando non possa essere lanciato in quella directory.

Può anche capitare che lanciamo per errore un comando senza argomento e questo continui ad andare e aspettare. Per interrompere il comando possiamo digitare `Cntr+C`.

3.4 Listing command

Possiamo eseguire il comando:

```
ls
```

per fare il listing dei file e delle subdirectories presenti nella nostra *current working directory*. Il comando `ls` può essere utilizzato anche in subdirectories.

Appunto importante è che possiamo anche scrivere il comando `ls` seguito dal path di una specifica subdirectory per guardarci dentro, ma senza effettivamente spostarci al suo interno.

3.4.1 Option

Possiamo aggiungere delle opzioni, precedute da un trattino, al comando listing. Come vedremo fin da subito, le opzioni possono essere combinate tra loro. Ovviamente non tutte le opzioni sono elencate in queste note e nel momento in cui si dovessero avere dei dubbi si può eseguire il comando:

```
ls --help
```

o equivalentemente:

```
man ls
```

per farci restituire dalla shell il manuale con tutte le istruzioni. Nel manuale ci si muove con le frecce direzionali per navigare line-by-line, con B e la spacebar per spostarsi di una pagina intera, "/" insieme al carattere o parola chiave per fare una ricerca e "Q" per uscire dal manuale.

L'opzione `-a`:

```
ls -a
```

fa in modo che vengano anche mostrate le directory nascoste, il cui nome inizia con "." e con ".." .

L'opzione `-A` ci permette di renderci conto che le option sono cases-sensitive:

```
ls -A
```

fa in modo che vengano anche mostrate le directory nascoste, il cui nome inizia con "." .

L'opzione `-F`:

```
ls -F
```

fa in modo che gli output vengono classificati aggiungendo un marker alla fine del nome per indicare cosa sono. "/" indica una directory, "@" indica un link e "*" un file eseguibile.

L'opzione `-l`:

```
ls -l
```

i file vengono mostrati nel long format e sono mostrate le seguenti informazioni:*file mode, number of links, owner name, group name, number of bytes, abbreviated month, day-of-month file was modified, minute file last modified, pathname.*

L'opzione `-lt`:

```
ls -lt
```

i file vengono mostrati nel long format, ma in ordine temporale dall'ultima modifica. Il più recentemente modificato appare come primo nell'elenco.

L'opzione `-lr`:

```
ls -lr
```

ha una funzione simile a `-lt`, ma elenca i file in ordine alfabetico inverso.

L'opzione `-lh`:

```
ls -lh
```

i file vengono mostrati nel long format, ma la loro dimensione viene indicata con i prefissi delle unità; ad esempio B (Byte), K (kilobyte), M (megabyte), G (gigabyte), T (terabyte), P (petabyte).

L'opzione `-R`:

```
ls -R
```

permette di listare tutte le nested (annidate) sub-directories di una directory.

3.4.2 Permission

Possiamo vedere che una linea di output tipica del comando `ls -l` è:

```
-rwxrwxrwx
```

vediamo cosa vuol dire. Il primo carattere indica il tipo di file: "-" indica un generico file; "d" è una directory. A seguire possiamo dividere l'output in 3 triplette formate da `rwx`, che stanno rispettivamente per *read*, *write* e *execute*, e indicano i tipi di permessi che si hanno. La prima tripletta si riferisce ai permessi che ha l'owner del file, la seconda tripletta ai permessi dei membri del gruppo di lavoro e la terza riguarda tutti gli altri utenti. Ad esempio un file con:

```
-rw-r--r--
```

indica un generico file di cui l'owner possiede i permessi di read and write, mentre il gruppo e tutti gli altri utenti solamente read.

Possiamo ovviamente modificare i permessi di un certo file (se siamo gli owner). Il comando `chmod` è utilizzato per cambiare i permessi. Dobbiamo immaginarci la stringa di permessi come una serie di bits, con 1 se il permesso è "acceso" e 0 se invece è negato. Ad esempio:

```
rw-rw-rw- = 110 110 110
```

Nel comando `chmod` dobbiamo scrivere le varie stringhe 110, 111, 000 in codice binario. Ricorda:

$$111 \quad \text{in binario} \quad = 7 \tag{3.4.1}$$
$$110 \quad \text{in binario} \quad = 6 \tag{3.4.2}$$
$$101 \quad \text{in binario} \quad = 5 \tag{3.4.3}$$
$$100 \quad \text{in binario} \quad = 4. \tag{3.4.4}$$

Dunque se ad esempio vogliamo che l'owner abbia il permesso `r` e `w`, allora vorremmo avere la stringa 110 nella prima tripletta, eseguiamo dunque:

```
chmod 600 some_file
```

così che i permessi dell'owner vengono cambiati in 6 binario, ma quelli del gruppo e degli altri user vengono lasciati intoccati essendoci 0.

Nota che il comando `chmod` può essere anche sulle directory, ma i permessi `r` e `w` possono essere attribuiti solo se c'è anche `x`.

3.5 Esplorando altre directories

Possiamo spostarci in un altra directory usando il comando `cd` (change directory) seguito dal nome della directory in cui vogliamo metterci. Se devi entrare dentro diverse subdirectories puoi sia eseguire più volte il comando entrando in una cartella per volta, oppure anche scrivere `cd` seguito dal path che vuoi seguire.

Il comando `cd` senza una directory specificata ci riporta nella home directory di default.

Ovviamente molte volte ci interessa "entrare" in delle directory, ma altre ci interesserebbe anche tornare indietro. In questi casi il comando `cd` seguito dalla directory obiettivo non è d'aiuto. Possiamo eseguire il comando:

```
cd ..
```

il che permette di portarci nella cartella ".." ovvero nella directory speciale che sta a rappresentare *the directory containing this one* o anche *parent of current directory*. La directory ".." possiamo vederne l'esistenza mettendoci in una randomica subdirectory e eseguendo:

```
ls -F -a
```

Una shortcut che possiamo utilizzare è il dash "-" che utilizzato nel comando `cd` viene interpretato come *into the previous directory I was in*. La differenza tra `cd ..` e `cd -` è che la prima ci porta indietro di un livello, mandandoci nella directory immediatamente precedente la corrente nel path, ma il secondo comando ci riporta nella directory in cui ci trovavamo prima di spostarci.

Capitolo 4

Working with files and directories

Abbiamo visto nel capitolo precedente come muoverci tra le cartelle e vedere al loro interno, ma non abbiamo ancora detto come creare, copiare, spostare o eliminare files e directories.

4.1 Comandi mkdir, mv, cp e rm

Il comando `mkdir` (making directory) ci permette di creare una directory. A seguito del comando `mkdir` possiamo inserire un relative o absolute path in modo da creare la cartella dove preferiamo; ad esempio:

```
mkdir Gabriele/Desktop/prova
```

crea la cartella *prova* all'interno del Desktop. Il comando `mkdir` non si limita a creare una cartella per volta, possiamo creare una cartella e le sue sub-directories in una sola operazione:

```
mkdir ../prova/data ../prova/results
```

crea la cartella prova e al suo interno le directories *data* e *results*.

A questo punto di cartelle vuote ce ne facciamo poco, vediamo come creare un file di testo. In generale scrivere `nome_editor` (ad esempio se scrivi `code` apri VScode) seguito da `nome_file.txt` permette di aprire un file di testo. Un editor direttamente nel terminale lo abbiamo compilando:

```
nano draft.txt
```

ricordati però di digitare `Ctrl+O` per scrivere sul file, assegnargli un nome ed uscire dall'editor con `Ctrl+X`. Nota che `control` è anche indicato con `^`.

Compilando il comando `touch`:

```
touch draft.txt
```

possiamo creare un file di testo vuoto, che non pesa nessun byte.

Ovviamente c'è anche un comando che ci permette di muovere files e cartelle. Sia nel senso di **spostarle**, ma anche nel senso di **renaming**. Infatti `mv` (move) permette di "muovere" nel senso di rinominare un file, proprio perché per la shell move ha lo stesso effetto di un renaming.Dobbiamo scrivere:

```
mv thesis/draft.txt thesis/quotes.txt
```

dove indichiamo con il primo termine l'oggetto da muovere (rinominare) e con il secondo dove lo muoviamo (il nuovo nome). Però il comando `mv` lavora anche sulle cartelle, infatti se scriviamo:

```
mv thesis/draft.txt .
```

quello che facciamo è prendere il file `draft.txt` e muoverlo, sta volta lo spostiamo, nella current working directory (indicata con `.`).

Possiamo anche copiare dei file o cartelle intere usando `cp` (copy). Se scriviamo:

```
cp quotes.txt thesis/quotation.txt
```

stiamo copiando il file `quotes.txt` nel file `thesis/quotation.txt`.
Il comando può essere utilizzato anche per copiare le directory e tutto il loro contenuto. L'unico vincolo è il dover utilizzare l'opzione `-r` (recursive):

```
cp - thesis thesis_backup
```

copia la cartella `thesis` nella cartella `thesis_backup`.

Possiamo anche assumere il ruolo di distruttori e cancellare file e cartelle. Ovviamente questa operazione è molto delicata e va fatta con cognizione per non perdere informazioni. Il comando `rm` (remuve) funziona uguale al comando copy:

```
rm quotes.txt
```

cancella il file `quotes.txt`, mentre:

```
rm -r thesis
```

cancella la directory `thesis`.

Nota. Tieni a mente che la shell non ha una directory cestino, per cui una volta che cancelli qualcosa non la si può recuperare e l'operazione è definitiva.

4.1.1 Attenzione

Fai sempre attenzione perché i comandi `mv`, `cp` e `rm` agiscono "silenziosamente" su qualsiasi file abbia lo stesso nome di quello messo come oggetto. Questo può portare ad una perdita di dati.

Una precauzione potrebbe essere utilizzare l'opzione `-i` o `--interactive` che permette tramite interazione di confermare il comando. Nota che solitamente per comandi a terminale non si utilizza, ma lo si usa per loop o codici lunghi.

4.2 Wildcards

Esistono questi oggetti, chiamati **wildcards**, che permettono di lasciare dei caratteri variabili per indicare più opzioni possibili. Spiegato meglio. Il carattere `*` rappresenta zero o altri caratteri, nel senso che il comando:

```
ls *.pdb
```

lista tutti i file che hanno un qualche nome (`*`) la cui estensione sia `.pdb`. Invece il comando:

```
ls p*.pdb
```

lista tutti i files che iniziano con la lettera `p`, continuano con qualcosa (`*`) e terminano con l'estensione `.pdb`.

La wildcard la possiamo usare quando preferiamo, anche combinata con altri comandi.

Invece `?` è una wildcard che rappresenta un solo preciso carattere. Ad esempio se abbiamo una directory con all'interno:

methane.pdb	octane.pdb
cubane.pdb	ethane.pdb

allora se scriviamo:

```
ls ?ethane.pdb
```

rappresenta solo methane.pdb e non ethane.pdv poiché `?` rappresenta 1 solo carattere. Invece:

```
ls ???ane.pdb
```

indica 3 caratteri seguiti da ane.pdv, ossia: cubane.pdb, ethane.pdb e octane.pdb.

Capitolo 5

Pipes and filters

Possiamo vedere alcuni comandi per ispezionare i vari file. Il comando `wc` (word count) permette di ispezionare un file `.pdb` e restituisce il numero di *righe*, *parole* e *caratteri*.

Però se volessimo solamente il numero di righe possiamo utilizzare l'opzione `-l`, così come il numero di parole `-w` o di caratteri `-m`.

Il simbolo `>` dice alla schell di *indirizzare* l'output del comando in un file di output anziché scriverlo sul terminale. Se eseguiamo:

```
wc -l *.pdb > lenghts.txt
```

stiamo indirizzando l'output del comando `wc -l *.pdb`, dunque la lista del numero di righe in ogni file `*.pdb`, in un file di testo `lenghts.txt`. Se il file `lenghts.txt` non esiste lo crea, mentre se esiste già lo sovrascirve. Il comando `>>` *appende* l'output del comando in un file di output. Per cui:

```
wc -l *.pdb >> lenghts.txt
```

prende l'output di `wc -l *.pdb` e lo appiccica a seguire di quello che già c'è scritto in `lenghts.txt`.

Possiamo stampare a terminale il contenuto di un file di testo utilizzando il comando `cat` (concatenate). Ad esempio possiamo scrivere:

```
cat lenghts.txt
```

Possiamo utilizzare il comando `sort` con l'opzione `-n` per disporre in ordine crescente su base numerica il contenuto di un file, ad esempio:

```
sort -n lenghts.txt
```

notando che il comando `sort` non modifica il contenuto del file, ma quello che fa è stampare in modo ordinato. Se utilizziamo il comando `head` allora il terminale ci stampa le prime 10 righe del file, ma se insieme ci mettiamo l'opzione `-n`, allora possiamo dire quante righe (dall'inizio) effettivamente stampare; ad esempio:

```
head -n 1 lenghts.txt
```

L'alter ego di `head` è `tail` e funziona in maniera analoga.

Nota che non esiste un comando per stampare una specifica riga di un file, ma puoi combinare `head` e `tail` in modo da ottenere il risultato. Vedi l'esempio nel capitolo §6.

La barra verticale | tra due comandi è detta **pipe** e dice alla shell che vogliamo usare l'output del comando a sinistra come input del comando a destra. Questo oggetto permette di rimuovere l'utilizzo di file o passaggi intermedi. Un esempio dell'utilizzo della pipe è:

```
sort -n lenghts.txt | head -n 1
```

ci permette di ottenere l'ordinamento di `lenghts.txt` e poi di stampare la prima riga. Un altro esempio è:

```
wc -l *.pdb | sort -n | head -n 1
```

che ci permette di vedere il numero di righe in ogni file, ordinarle e poi stamparne la prima.

Il comando `cut` è usato per rimuover una parte di ogni riga di un file. L'opzione `-d` ci permette di specificare che la virgola è il *delimiter*, mentre `-f` specifica che vogliamo estrarre la seconda colonna.

Il comando `uniq` permette di rimuovere righe adiacenti uguali. L'opzione `-c` ci dice quante volte si è ripetuta una riga.

Capitolo 6

Loops

I **loops** permettono di ripetere un comando o un set di comandi su ogni elemento della lista. La struttura di un loop è la seguente:

```
#The word "for" indicates the start of a "For-loop" command
$ for thing in list_of_things
#The word "do" indicates the start of the job execution list
> do
    #Indentation within the loop is not required, but aids legibility
>     operation_using/command $thing
#The word "done" indicates the end of a loop
> done
```

Puoi vedere in questo loop che il simbolo utilizzato per commentare del codice è `#`.

Un esempio di loop che stampa la seconda riga di una serie di file la otteniamo combinando i comandi `head` e `tail`:

```
$ for filename in basilisk.dat minotaur.dat unicorn.dat
> do
>    echo $filename
>    head -n 2 $filename | tail -n 1
> done
```

in cui `echo` lo utilizziamo per stampare il nome del file, ogni volta che il loop gira la variabile `filename` diventa uno dei file che abbiamo elencato e nel loop abbiamo richiesto che venga prima preso il valore delle prime due righe, ma che venga stampato solo l'ultimo di essi. Nota che nel loop richiediamo il valore della variabile premettendo il simbolo $.

Nota. Quando eseguiamo un loop e poi lo andiamo a rivedere utilizzando le frecce direzionali lo vediamo scritto tutto su una riga e con i vari pezzi separati da un punto e virgola (;).

6.1 Files con nomi contenenti gli spazi

Potrebbe capitare che dobbiamo elencare in un loop dei nomi contenenti degli spazi; in questo caso dobbiamo utilizzare le virgolette:

```
$ for filename in "mio basilisk.dat" "tuo minotaur.dat" "suo unicorn.dat"
> do
>    echo "$filename"
>    head -n 2 "$filename" | tail -n 1
> done
```

notando che lo dobbiamo anche inserirle all'interno del `do-done`.

6.2 Esercizi

In aula sono stati eseguiti diversi esercizi, ma per essi rimando alle slide del corso. Uno che ho trovato carino è il seguente:

Esercizio 23 Andiamo nella cartella shell-lesson-data/exercise-data/alkanes (utilizzando il file .zip inviato dal docente). Supponiamo di voler fare un esperimento misurando una serie di valori per diversi specie (cubane, ethane, methane) e diverse temperature (25, 30, 37, 40). Come possiamo creare le nostre cartelle con i loops?

Risposta Eseguendo il loop:

```
$ for temperature in 25 30 35 40
> do
>   mkdir $temperature
>   for filename in *.pdb
>   do
>       mkdir $temperature/$filename
>   done
> done
```

Capitolo 7

Shell script

Gli **shell script** erano un po' il main goal del corso fin dall'inizio, anche se non lo abbiamo detto fin dall'inizio. Gli *schell script* sono comandi scritti su un file con estensione `.sh`. Sono tipicamente programmi piccoli e snelli che permettono di automatizzare operazioni ripetitive.

Ad esempio possiamo scrivere:

```
nano middle.sh
```

e all'interno di `middle.sh`:

```
head -n 15 octane.pdb | tail -n 5
```

e poi eseguirlo come:

```
bash middle.sh
```

e ci stamperà le righe dalla 10 alla 15 del file `octane.pdb`.

Però possiamo anche non inserire all'interno di uno script un file specifico, ma inserire degli argomenti:

```
head -n 15 "$1" | tail -n 5
```

in cui mettiamo "" per precauzione e `$1` significa che prendiamo il primo argomento sulla command line. Ovviamente possiamo mettere più argomenti; ad esempio:

```
head -n "$2" "$1" | tail -n "$3"
```

in questo caso il file lo eseguiamo come:

```
bash middle.sh 1 2 3
```

Nel caso in cui volessimo porcessare tutti i file di un certo tipo possiamo utilizzare `$@$` che significa *tutti gli argomenti della command line allo shell script*.

Abbiamo già visto come il comando `history` ci permetta di vedere l'elenco dei comandi eseguiti, ma possiamo eseguire un comando per salvarli all'interno di uno script:

```
history | tail -n 5 > redo-figure-3.sh
```

che per esempio stampa gli ultimi 5 comandi all'interno dello script `redo-figure-3.sh`.

7.1 Opzioni

Abbiamo detto che con `bash` possiamo eseguire i comandi salvati nel file, ma esistono anche delle opzioni. L'opzione `-x` esegue lo script in modalità debug.

Capitolo 8

Finding things

Potremmo aver bisogno di cercare file o parole all'interno dei file e in questo capitolo analizziamo i comandi che ci permettono di fare ciò.

8.1 Grep

Il primo comando è `grep` (global/regular expression/print) che è un importante command-line program che cerca e stampa linee di un file che combacino con un determinato pattern; ad esempio:

```
grep not haiku.txt
```

cerca tutte le linee con la parola `not` all'interno di `haiku.txt`. Attenzione solo a due cose: la prima è che il comando è case-sensitive e per rimuovere questa cosa possiamo utilizzare l'opzione `-i` (che rende il comando case-insensitive); la seconda è che se cerchiamo ad esempio il pattern `the` non solo ci stampa le righe con l'articolo `the`, ma anche tutte le righe al cui interno c'è `the`, ad esempio la riga con la parola `thesis`, per rimuovere questo problema possiamo utilizzare l'opzione `-w` che introduce il word boundary, ossia che le parole devono combaciare perfettamente al pattern.

Ovviamente possiamo cercare pezzi di frase, basta utilizzare le virgolette (").

Un'altra opzione è `-n` che numera le linee e riporta i numeri delle linee corrispondenti. L'opzione `-v` ribalta la nostra ricerca e ci stampa tutte le righe che non contengono il pattern. L'opzione `-r` (recursive) cerca il pattern in maniera ricorsiva in un gruppo di file e subdirectories.

Il punto di forza di `grep` è che si possono includere le wildcard all'interno dei pattern; ad esempio:

```
grep "^.o" haiku.txt
```

cerca all'interno di `haiku.txt` tutte le righe la cui seconda lettera è una "o".

8.2 Find

Il comando `find` cerca i file stessi che corrispondono ad un certo pattern. Il comando:

```
find .
```

permette di cercare all'interno della current working directory. L'opzione `-type` ci permette di cercare cose specifiche, ad esempio:

```
find . -type d
```

cerca solo directory, mentre l'opzione `f` cerca cose che sono file.

Usando `file` possiamo cercare anche per nome; ad esempio possiamo scrivere:

```
find . -name *.txt
```

cerchiamo tutti i file, con qualsiasi nome, che hanno estensione `.txt`. Questo esempio però non permette di cercare nelle subdirectories, ma per farlo dobbiamo utilizzare le virgolette:

```
find . -name "*.txt"
```

Utilizzando quest'ultimo esempio possiamo vedere come combinare comandi diversi e ad esempio contare le linee di tutti i file `.txt`. Non possiamo utilizzare le pipe in questo caso, ma dobbiamo utilizzare `$()`, che prima di tutto fa eseguire alla shell ciò che è dentro le parentesi tonde e successivamente prende l'output del comando e lo sostituisce a `$()`. Quindi scriviamo:

```
wc -l $(find . -name *.txt)
```

Capitolo 9

Remote working

In aula è stato anche affrontato l'argomento del *remote working*, ma preferisco non includerlo nelle note. È stato menzionato anche l'utilizzo di Git (e GitHub). Se qualcuno avesse bisogno può fare riferimento alle slide del corso.

Capitolo 10

Riepilogo comandi

- `exec` : esegue una nuova shell.
- `source` : per eseguire il file `.bashrc`.
- `export` : modifica temporanea del prompt (vedi le customisation nella sezione §2.2.3).
- `ls` : listing (vedi le opzioni nella sezione §3.4.1).
- `pwd` : print working directory, ovvero ti mostra il path assoluto.
- `chmod` : cambia i permessi di un certo file/directory.
- `cd` : change directory.
- `cd ..` : permette di tornare nella *parent of current directory.*
- `cd -` : permette di tornare nella *into the previous directory I was in.*
- `mkdir` : crea una nuova directory.
- `nano` : crea un file di testo e lo apre con un editor a terminale.
- `mv` : muove dei file/directory, anche nel senso di rinominare (vedi la sezione §4.1).
- `cp` : copia file in altri. Con l'opzione `-r` permette di copiare anche cartelle.
- `rm` : elimina file. Con l'opzione `-r` elimina cartelle.
- `*` : wildcard che rappresenta zero o altri caratteri.
- `?` : wildcard che rappresenta un solo specifico carattere.
- `wc` : word count e restituisce il numero di righe, parole e caratteri.

- `>` : permette di indirizzare l'output di un comando in un file.
- `>>` : permette di appendere l'output di un comando in un file.
- `cat` : permette di stampare a terminale il contenuto di un file.
- `sort -n` : ordina in modo crescente su base numerica il contenuto di un file.
- `head -n` : stampa a terminale le prime 10 righe di default, ma se ci aggiungiamo un numero possiamo scegliere il numero di righe.
- `tail -n` : analogo ad `head -n`.
- | : permette di combinare più comandi.
- `echo` : stampa a terminale il nome del file.
- `bash [filename]` : esegue i comandi salvati nel file.
- `bash -x` : esegue i file in modalità debug.
- `grep` : per stampare linee con pattern specifici (vedi le opzioni nel capitolo §8).
- `find` : per cercare file (vedi il capitolo §8 per le opzioni).
- `$()` : mette l'output del comando.

Bibliografia

[1] William Shotts. *The Linux Command Line*.