Lezione 5
BASH

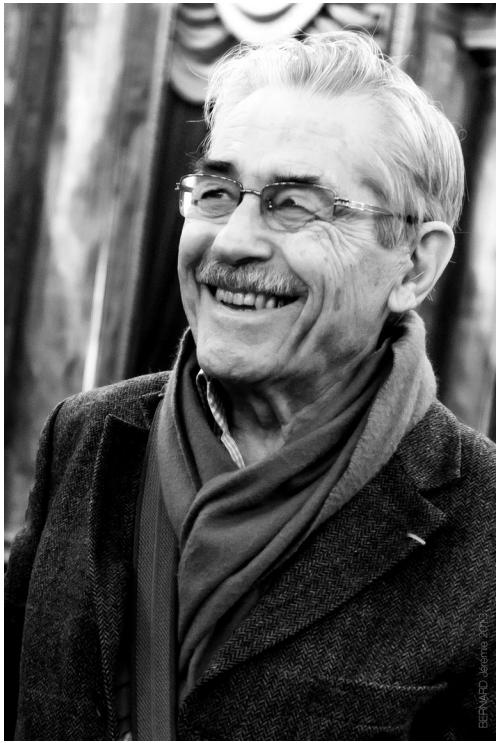
Sistemi Operativi (9 CFU), CdL Informatica, A. A. 2022/2023
Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Università di Modena e Reggio Emilia
http://weblab.ing.unimo.it/people/andreolini/didattica/sistemi-operativi

Quote of the day

(Meditate, gente, meditate…)

"… I felt that commands should be usable as building blocks for writing more commands, just like subroutine libraries. Hence, I wrote "RUNCOM", a sort of shell driving the execution of command scripts, with argument substitution.

Louis Pouzin (1931-)



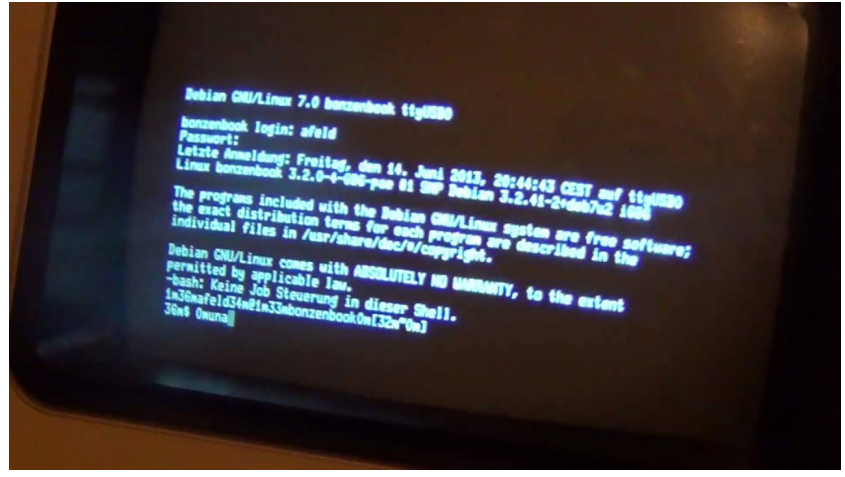
Creatore di RUNCOM (prima shell in MULTICS)
Inventore del termine "shell"
Progettista di CYCLADES (prima rete ad adottare i datagrammi)

INTRODUZIONE

Lo scenario

(Uno studente che sa usare GNOME vuole imparare la linea di comando)

Uno studente sa avviare Debian GNU/Linux e sa usare GNOME.
Lo studente, armato di coraggio e intraprendenza, decide di imparare ad usare l'interfaccia testuale.

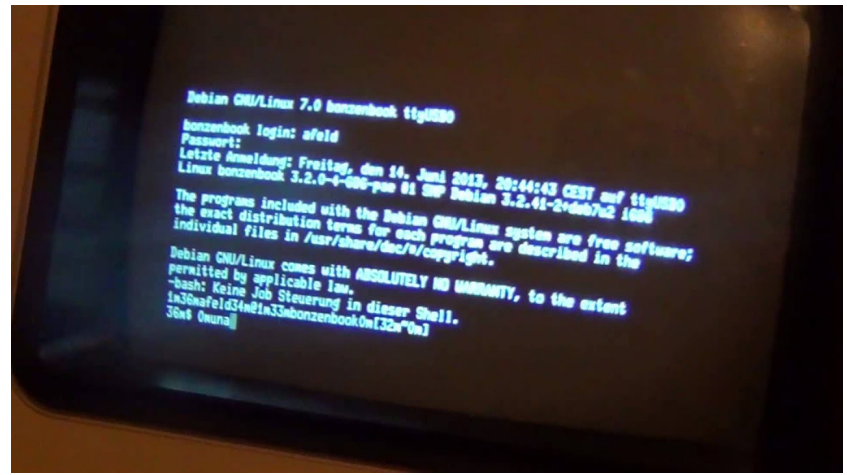


Lo scenario

(Uno studente che sa usare GNOME vuole imparare ad usare BASH)

Motivazioni per approfondire lo studio delle interfacce testuali:

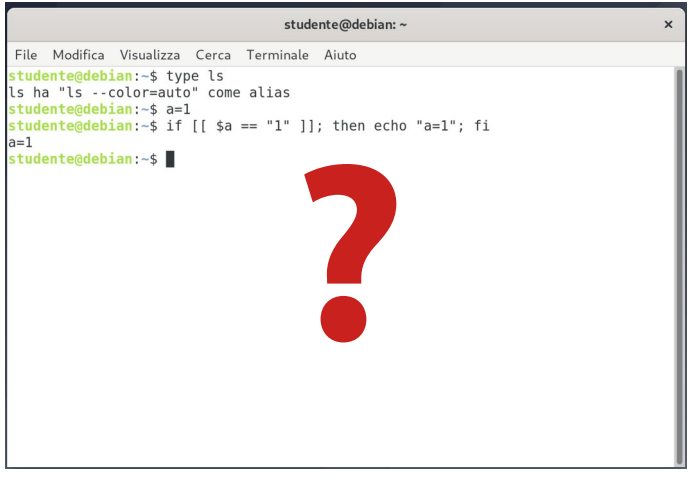
- lo studente ben comprende l'importanza di padroneggiare il terminale (e gradirebbe meno ciance);
- lo studente fa tutto quello che dice il docente, purché passi l'esame;
- lo studente inizia a comprende-re i propri limiti e piano piano comincia a fidarsi del docente.

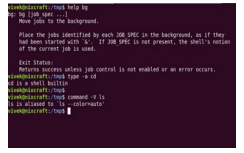


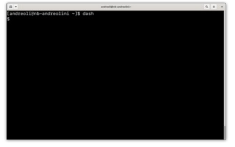
Interrogativi

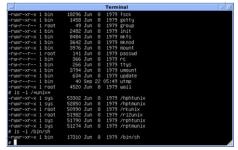
(Quale interprete studiare? Quali costrutti base offre l'interprete dei comandi?)

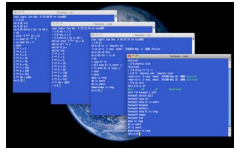
Quale interprete dei comandi si deve approfondire?
Quali costrutti di base mette a disposizione l'interprete dei comandi scelto?

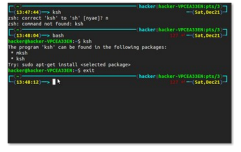


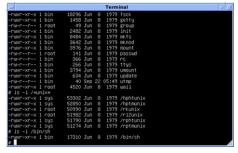


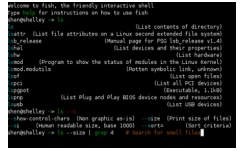












SCELTA INTERPRETE DEI COMANDI

Una amara constatazione

(Non si può sapere tutto di tutto)

Esistono tanti interpreti da linea di comando.

`sh`, **`bash`**, **`dash`**, **`csh`**, **`tcsh`**, **`ksh`**, **`zsh`**, **`fish`**.

E non finiscono qui.

Studiarli tutti non ha molto senso.

Il tempo a disposizione è limitato.

Le energie a disposizione sono limitate.

Solitamente si usa un interprete da linea di comando.

→ È necessario puntare su un interprete, cercando di massimizzare i profitti legati al suo studio.

La scelta più ovvia

(L'interprete dei comandi di default)

La scelta più ovvia ricade sull'interprete dei comandi di default nei SO GNU/Linux.

- È l'interprete usato da un terminale (grafico o testuale che sia).
- È l'interprete usato per eseguire script.
- È l'interprete in cui sono scritte diverse utility.

L'interprete di default nei SO GNU/Linux è **BASH**.

Funzionalità di BASH

(Tante; alcuni dicono troppe)

Configurabilità tramite file.
Job control.
Aritmetica intera.
Liste e Array.
Costrutti di controllo del flusso logico.
Costrutti iterativi.
Espansione e pattern matching degli argomenti.
Shortcut da tastiera per l'uso efficiente dell'interprete.
Supporto per il debugging degli script.
Modularità (funzioni, alias, plugin).

Una avvertenza

(Non imparerete tutto oggi)

Quella che segue è una introduzione di base.
Oggi non saranno esaminate tutte le funzionalità ora esposte.
Le funzionalità saranno riprese durante l'intero arco della Parte 1.

VARIABILI

Definizione

(Partiamo dal principio)

Una **variabile** è una associazione tra due elementi:

un **nome simbolico**;

una **cella di memoria**.

Nome simbolico: è un nome usato per riferire la variabile in una sessione di lavoro.

Primo carattere: deve essere alfabetico o un underscore.

Altri caratteri: possono essere alfanumerici o underscore.

Cella di memoria: è una porzione della memoria disponibile, deputata a contenere il valore associato al nome simbolico.

Assegnazione

(Operatore =)

L'assegnazione di una variabile **`var`** al valore **`val`** avviene tramite l'operatore di assegnazione **=**:

```
var=val
```

Ad esempio, per assegnare alla variabile **`a`** il valore **`1`**:

```
a=1
```

Non si usino spazi per separare nome simbolico, valore e operatore. Il seguente è un errore:

```
a = 1
```

Accesso al valore

(Si prefigge la variabile con il simbolo **$**)

Per accedere al valore di una variabile **var**, è sufficiente prefiggere al suo nome simbolico il simbolo di espansione **$**.

```
$var
```

Ad esempio, se la variabile **a** contiene il valore **1**, **$a** viene interpretato come **1**.

Errori tipici 1/1

(Immettendo **`$a`** si ottiene un errore)

Si effettua un primo, rozzo tentativo di stampa del valore di una variabile. A tal scopo, si digita:

```
$a
```

Purtroppo si ottiene un messaggio di errore:

```
bash: 1: command not found
```

Che cosa è successo?

- L'interprete ha convertito **`$a`** in **`1`**.
- L'interprete ha provato ad eseguire il comando **`1`**.
- Il comando **`1`** non esiste, di solito.
- L'interprete segnala l'impossibilità di eseguire **`1`**.

Soluzione

(Bisogna stampare a video **`$a`**)

Questo problema si risolve facilmente; è sufficiente trovare un modo per stampare il valore della variabile **`a`**.

Ci sono diversi modi per farlo.

Per ora, fidatevi; l'assegnazione è andata a buon fine.

Ve ne accorgete dal fatto che nel messaggio di errore compare il valore della variabile.

Distruzione

(Si usa il comando **`unset`**)

Per cancellare una variabile **`var`**, si usa il comando **`unset`**.

```
unset var
```

Ad esempio, per cancellare la variabile **`a`**:

```
unset a
```

Errori tipici 1/2

(Si assegna il valore nullo alla variabile)

Si prova a passare ad **`unset`** il valore della variabile, e non il suo nome simbolico. A tal scopo, si digita:

```
unset $a
```

Purtroppo si ottiene un messaggio di errore:

```
bash: unset: "1": non è un identificatore valido
```

Che cosa è successo?

- L'interprete ha convertito **`$a`** in **`1`**.
- L'interprete ha provato ad eseguire il comando **`unset 1`**.
- Un nome simbolico non può iniziare con un numero.
- L'interprete segnala il nome invalido di variabile.

Errori tipici 2/2

(Si assegna il valore nullo alla variabile)

Si potrebbe provare a cancellare **a** semplicemente impostandola al valore nullo.

```
a=
```

Il comando è corretto, ma non sortisce l'effetto sperato.
La variabile **a** è ancora presente, ma con un valore non assegnato!

Tipizzazione

(Non esiste; le variabili sono viste come stringhe)

Le variabili NON sono tipizzate; sono tutte considerate come stringhe.
A puro titolo di esempio, si provi ad assegnare il valore **`8/4`** alla variabile **`a`**:

```
a=8/4
```

Immettendo il comando **`$a`**, si ottiene l'errore seguente, da cui si deduce l'amara verità:

```
bash: 8/4: File o directory non esistente
```

Se **`a`** fosse stata intera,
ci si sarebbe aspettati
il valore **`2`**.

Attributi

(Come disse Oscar Panno: "Mettiamoci una pezza")

Una variabile può essere arricchita con uno o più **attributi**, che ne specializzano il comportamento.

Esempi di attributi:

- sola lettura;
- interpretazione di uno specifico tipo (intero, array, array associativo, …);
- conversione a lower/upper case;
- …

Forma limitata di tipizzazione.

Gestione degli attributi

(Si usa il comando **`declare`**)

Il comando **`declare`** gestisce gli attributi di una variabile.

Che si intende esattamente con "gestisce"?

- Stampa degli attributi.
- Impostazione degli attributi.
- Cancellazione degli attributi.

Stampa degli attributi

(Si usa l'opzione **`-p`** del comando **`declare`**)

Per stampare gli attributi di una variabile **`var`** si esegue il comando seguente:

```
declare -p var
```

Ad esempio, si imposti **`a=1`**:

```
a=1
```

Po si stampino gli attributi della variabile **`a`**:

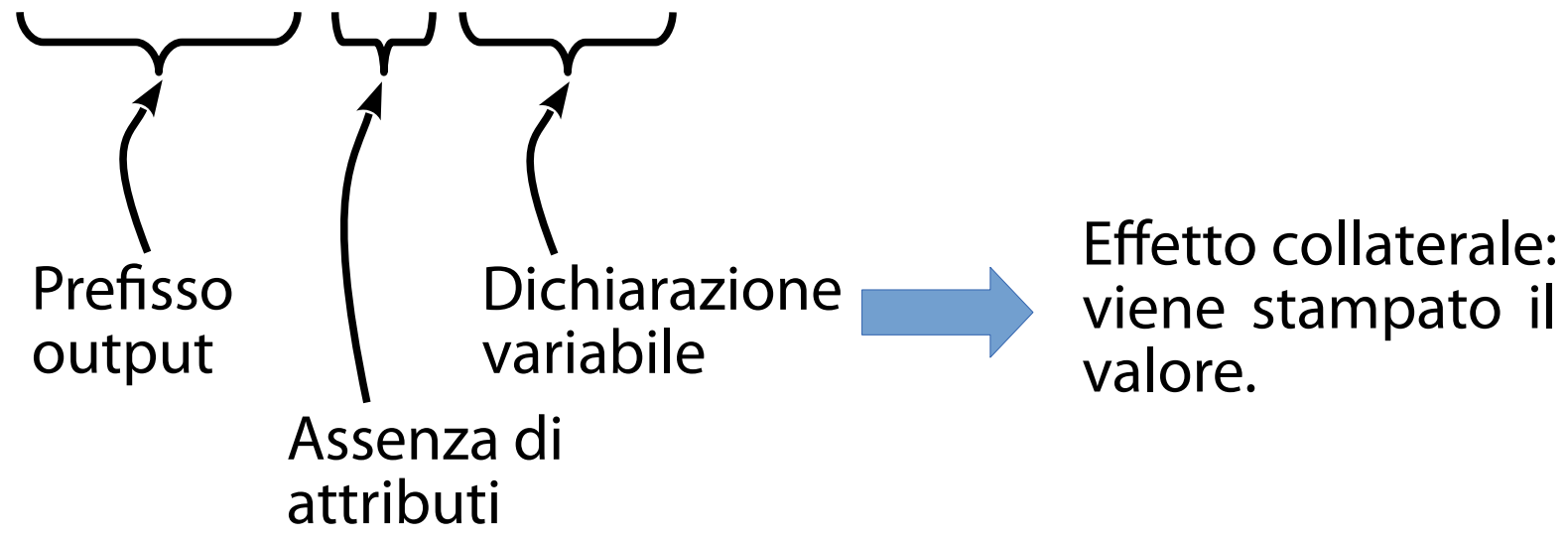
```
declare -p a
```

L'output del comando

(Semplice e immediato)

Si ottiene l'output seguente:

```
declare -- a="1"
```



Impostazione attributo "intero"

(Si usa l'opzione **`-i`** del comando **`declare`**)

Per impostare l'attributo "intero" ad una variabile **`var`** si esegue il comando seguente:

```
declare -i var
```

Ad esempio, per impostare l'attributo "intero" alla variabile **`a`**:

```
declare -i a
```

Verifica impostazione attributo

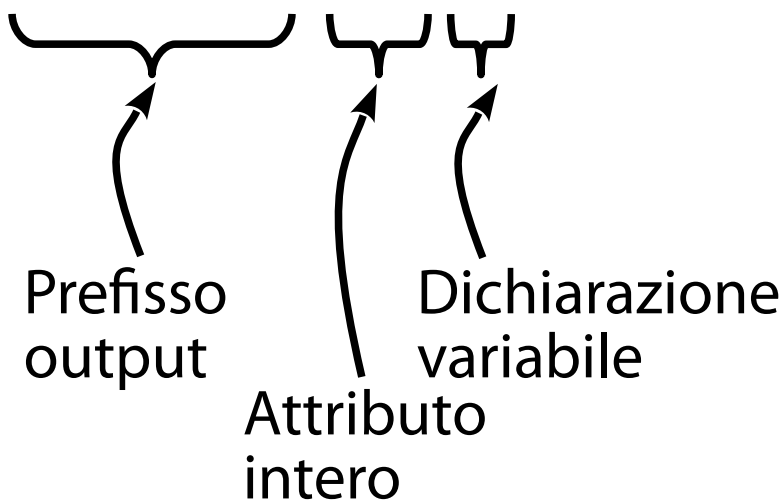
(Si stampa l'attributo con **`declare -p`**)

Si stampano gli attributi della variabile **`a`**:

```
declare -p a
```

Si ottiene l'output seguente:

```
declare -i a
```



Impostazione valore intero

(È un'assegnazione classica)

Ora si può assegnare un valore intero alla variabile **`a`**:

```
a=8/4
```

Verifica impostazione valore

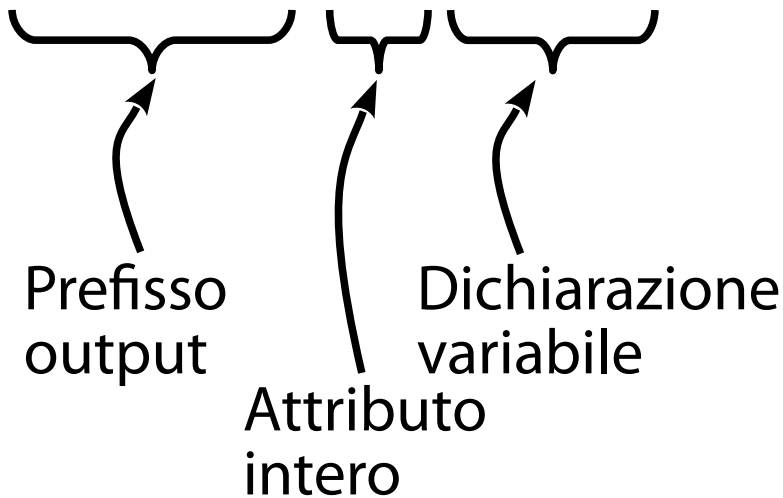
(Si stampa l'attributo con **`declare -p`**)

Si stampano gli attributi della variabile **`a`**:

```
declare -p a
```

Si ottiene l'output seguente:

```
declare -i a="2"
```



Effetto collaterale: viene stampato il valore intero risultato dell'espressione 8/4.

Cancellazione attributo

(Si annulla l'opzione con **+** al posto di **–**)

È possibile cancellare l'attributo (ad esempio, "intero") per la variabile **`var`** invertendo il segno dell'opzione:

```
declare +i var
```

Ad esempio, per rimuovere l'attributo "intero" alla variabile **`a`**:

```
declare +i a
```

D'ora in avanti, il valore della variabile **`a`** viene trattato nuovamente alla stregua di una stringa.

Esercizio 1 (2 min.)

Aprite una nuova finestra di terminale.
Dichiarate una variabile di nome **`a`** con attributo intero e valore pari a **`8/3`**.
Stampate il valore della variabile. Notate qualcosa di strano?

I/O SU TERMINALE

Lettura di un input

(Per il momento, da tastiera)

Il comando **`read`** legge un input utente e lo memorizza in una variabile.

```
read var
```

Si noti il blocco dovuto all'attesa di input utente.
Il valore immesso è interpretato in funzione degli attributi della variabile.

Ad esempio, per leggere una stringa e memorizzarla nella variabile **`a`**:

```
read a
```

Scrittura di un output

(Per il momento, su terminale)

Il comando **`echo`** stampa una espressione su terminale.
È possibile stampare stringhe costanti:

```
echo testo semplice
```

È anche possibile stampare il valore di una variabile **`var`**:

```
echo $var
```

Esercizio 2 (2 min.)

Aprite una nuova finestra di terminale.
Dichiarate una variabile **`a`** con attributo intero.
Leggete il valore **`8/3`** tramite terminale nella variabile.
Stampate il valore della variabile.

ESCAPING E QUOTING

Caratteri speciali

(Sono usati da BASH; guai a toccarli!)

Alcuni caratteri, detti **caratteri speciali**, sono riservati all'uso dei costrutti di shell.

Un carattere è stato già conosciuto: **`$`**.

I caratteri sono tanti: **`$#|();&"'` `\!{}<>~SPACE`**

Questi caratteri non si possono usare, se non nei costrutti appositi.

Corollario: se stampati, potrebbero non fornire in output la loro rappresentazione testuale.

Un esempio concreto

(Stampa della stringa **`$a`**)

Si apre un nuovo terminale.

Si prova a stampare la stringa **`$a`** con il comando seguente:

```
echo $a
```

Non si ottiene output, perché **`$a`** è un'espressione usata per recuperare il valore della variabile **`a`**, che non è definita.

Si vorrebbe invece ottenere l'output letterale:

```
$a
```

Escaping di un carattere speciale

(Disabilita l'interpretazione del carattere speciale)

Il carattere speciale **** (backslash) introduce l'operazione di **escaping** di un carattere speciale.
Escaping: il carattere successivo non viene interpretato come speciale, bensì come carattere stampabile.

Con riferimento all'esempio precedente, l'espressione seguente disabilita il meccanismo di recupero del valore della variabile **a**:

```
\$a
```

La conseguenza dell'escaping

(Ora si riesce a stampare la stringa **`$a`**)

Si prova il comando seguente:

```
echo \$a
```

Si ottiene l'output seguente:

```
$a
```

Funziona!

Una scomodità

(Stampa di una stringa contenente tanti caratteri speciali)

Si vuole stampare la stringa seguente:

```
$a$a$a$a$a$a$a$a$a$a
```

Si può eseguire il comando seguente:

```
echo \$a\$a\$a\$a\$a\$a\$a\$a\$a\$a
```

Si ottiene l'output corretto; tuttavia, l'immissione di ben dieci caratteri di escape è alquanto scomoda.

Quoting forte

(Si incastra una espressione tra singoli apici)

Il carattere ‘ (apice singolo) permette di effettuare il quoting forte di una espressione.

Quoting forte (o singolo): incastrando una espressione tra due apici singoli, si disabilita ogni forma di interpretazione dei caratteri speciali.

Con riferimento all’esempio precedente, l’espressione seguente disabilita tutti i tentativi di recupero del valore della variabile **a**:

```
'$a$a$a$a$a$a$a$a$a$a'
```

La conseguenza del quoting forte

(Ora si riesce a stampare la stringa **`$a$a$a$a$a$a$a$a$a$a`**)

Si prova il comando seguente:

echo **`'$a$a$a$a$a$a$a$a$a$a'`**

Si ottiene l'output seguente:

`$a$a$a$a$a$a$a$a$a$a`

Funziona!

Un altro vantaggio del quoting forte

(L'intera espressione è considerata come un singolo argomento)

Il quoting forte ha un'altra implicazione.
L'intera espressione quotata forte è considerata come un singolo argomento.

Scrivendo questo comando:

```
echo 'Un argomento'
```

il comando **echo** riceve un singolo argomento.
Scrivendo questo comando:

```
echo Due argomenti
```

il comando **echo** riceve due argomenti.

Che cosa cambia?

(Quoting forte o no in **`echo`**)

Nel caso del comando **`echo`** non cambia sostanzialmente nulla.
Il comando **`echo`** stampa tutti gli argomenti ricevuti, uno dopo l'altro. Quotare o no l'argomento non altera la sua semantica.

Purtroppo non è sempre così.

L'utility di sistema **`printf`**

(Stampa output formattate)

L'utility di sistema **`printf`** stampa output formattato, facendo uso delle stringhe di formato tipiche del linguaggio C.
Il primo argomento è una stringa di formato C.
Ogni argomento successivo è stampato con la stringa di formato richiesta.

Un esempio:

```
printf '%s\n' stringa
```

Si ottiene l'output seguente:

```
stringa
```

L'utility di sistema `printf`

(Stampa senza quoting forte)

Si prova a stampare una frase composta da più parole senza adottare il quoting.

```
printf '%s\n' questa è una stringa
```

Si ottiene l'output seguente:

```
questa
è
una
stringa
```

`%s`	→	formatta come stringa
`\n`	→	va a capo

Ogni argomento dopo la stringa di formato C è stampato come stringa e viene seguito da un ritorno a capo.

L'utility di sistema `printf`

(Stampa con quoting forte)

Si prova a stampare una frase composta da più parole adottando il quoting forte.

```
printf '%s\n' 'questa è una stringa'
```

Si ottiene l'output seguente:

```
questa è una stringa
```

L'unico argomento dopo la stringa di formato C è stampato come stringa e viene seguito da un ritorno a capo.

Un problema

(Il quoting forte inibisce ogni costrutto; è troppo restrittivo)

Alcune volte è desiderabile la capacità di passare un singolo argomento complesso complesso ad una applicazione, ma non l'inibizione dei meccanismi di BASH.

Classico esempio: creazione di un file con un nome dinamico, costruito a partire dal contenuto di una variabile.

L'utility di sistema **`touch`**

(Creazione file vuoti)

L'utility di sistema **`touch`**, fra le altre cose, crea file vuoti. Ogni argomento è interpretato come il nome di un nuovo file vuoto da creare.

Un esempio:

```
touch file.txt
```

Elencando i file della directory attuale con **`ls -l`**, si dovrebbe ottenere un file vuoto di nome **`file.txt`**.

L'utility di sistema `touch`

(Creazione file vuoti con quoting forte)

Si prova a creare un file contenente uno spazio e il contenuto di una variabile.

```
f='nuovo'
touch 'file $f.txt'
```

Tra i nuovi file creati ne compare uno con un nome veramente strano:

```
file $f.txt
```

Il quoting forte ha inibito ogni sorta di trasformazione di **`$var`** in un valore.

Quoting debole

(Si incastra una espressione tra doppi apici)

Il carattere “ (apice doppio) permette di effettuare il quoting debole di una espressione.

Quoting debole (o doppio): incastrando una espressione tra due apici doppi, si mantiene l’interpretazione di alcuni costrutti.

Quelli iniziati dai caratteri speciali **`$ ` \`** ****.

L'utility di sistema `touch`

(Creazione file vuoti con quoting debole)

Si prova a creare un file contenente uno spazio e il contenuto di una variabile. Si usa il quoting debole per passare un singolo argomento e permettere l'espansione del valore **`$f`**.

```
f='nuovo'
touch "file $f.txt"
```

Tra i nuovi file creati compare:

```
file nuovo.txt
```

Funziona!

Esercizio 3 (2 min.)

Eseguite i due comandi seguenti e spiegate la differenza di comportamento:

```
echo -e a\n
echo -e a\\n
```

MANIPOLAZIONE DI VARIABILI

Definizione

(Insieme di variabili, usate per personalizzare le applicazioni)

BASH mette a disposizione una serie di espansioni atte a **manipolare** il valore di una variabile.

Le manipolazioni possibili sono molte, fra cui:

- usare un valore di default se la stringa non è definita;
- ottenere la lunghezza di una stringa o sottostringa;
- estrarre, sostituire e rimuovere una sottostringa;
- …

Una doverosa premessa

(Si imposti la variabile **`s`** su cui effettuare le trasformazioni)

In quasi ognuno degli esercizi che segue si usa una variabile **`s`** contenente il valore **`abcABC123ABCabc`**.
Prima di ogni esercizio, se necessario ci si assicuri che tale variabile abbia il valore corretto con il comando seguente:

```
s=abcABC123ABCabc
```

La manipolazione nulla

(`${var}`)

L'espansione **`${var}`** introduce la manipolazione nulla. Non viene applicata alcuna manipolazione a **`var`**.

```
echo ${s}
abcABC123ABCabc
```

La manipolazione nulla è usata per evitare ambiguità nella rappresentazione di variabili.

“Evitare ambiguità”? Eh?
(WAT?)

Un esempio concreto

(Così Kermit si tranquillizza)

Si supponga di voler stampare il contenuto di **`s`**, seguito immediatamente dalla stringa OK. A tal scopo, si esegue:

```
echo $sOK
```

L'espressione **`$sOK`** presenta una certa ambiguità.

Ci si riferisce al valore della variabile **`$s`**, seguito da OK?

Ci si riferisce al valore della variabile **`sOK`**?

Per rimuovere l'ambiguità, BASH considera come nome della variabile la stringa più lunga possibile, ovvero **`sOK`**.

→ Il comando echo **`$sOK`** è sbagliato.

Un esempio concreto

(Così Kermit si tranquillizza)

La manipolazione nulla consente di "isolare" il nome della variabile s, rimuovendo ogni ambiguità:

```
echo ${s}OK
abcABC123ABCabcOK
```

Ad essere onesti, basta anche il quoting debole:

```
echo "$s"OK
abcABC123ABCabcOK
```

Uso di un valore di default

(`${var:-val}`)

L'espansione **`${var:-val}`** effettua le operazioni seguenti.

Se **`var`** è definita e non nulla, viene stampata.

Altrimenti viene stampata la stringa **`val`**.

```
echo ${s:-1}
abcABC123ABCabc

echo ${nonesistente:-1}
1
echo $nonesistente
```

Uso di un valore di default

(`${var:-val}`)

L'espansione **`${var:-val}`** effettua le operazioni seguenti.

La variabile **`var`** non è definita o alterata in alcun modo.

```
echo $s
abcABC123ABCabc

echo $nonesistente
```

Impostazione di un valore di default

(`${var:=val}`)

L'espansione **`${var:=val}`** effettua le operazioni seguenti.

Se **`var`** è definita e non nulla, viene stampata.

```
echo ${s:=1}
abcABC123ABCabc
echo $s
abcABC123ABCabc
```

Impostazione di un valore di default

(`${var:=val}`)

L'espansione **`${var:=val}`** effettua le operazioni seguenti.

Altrimenti, **`var`** viene definita (se non lo è già) e impostata al valore **`val`**.

```
echo ${nonesistente:=1}
1
echo $nonesistente
1
```

Lunghezza
(`${#var}`)

L'espansione **`${#var}`** ritorna la lunghezza della variabile **`var`**.

```
echo ${#s}
15
```

Estrazione di una sottostringa

(`${var:start}`)

L'espansione **`${var:start}`** estrae da **`var`** la sotto-stringa che inizia all'indice **`start`**.
L'indice comincia da 0.

```
echo ${s:0}
abcABC123ABCabc
echo ${s:7}
23ABCabc
```

Estrazione di una sottostringa

(`${var:start:len}`)

L'espansione **`${var:start:len}`** estrae da **`var`** la sottostringa che inizia all'indice start ed è lunga **`len`** caratteri.

L'indice comincia da 0.

```
echo ${s:7:3}
23A
```

Rimozione di una sottostringa

(`${var#pattern}`)

L'espansione **`${var#pattern}`** rimuove il match più piccolo (**non greedy removal**) di **`pattern`** all'inizio di **`var`**.

La variabile **`var`** non è definita o alterata in alcun modo.

Se **`pattern`** non corrisponde con l'inizio di **`var`**, viene stampata integralmente **`var`**.

Se **`pattern`** corrisponde con l'inizio di **`var`**, si rimuove **`pattern`** da **`var`** e si stampa la stringa risultante.

Rimozione di una sottostringa

(${var#pattern})

Il **pattern** è una rappresentazione efficiente di una sottostringa.

- Stringa esatta: **abc**.
- Stringa con caratteri speciali di match (**a*c**, **a?c**).
  - ***** (matcha tutto, incluso il nulla).
  - **?** (matcha un singolo, qualunque carattere).

Rimozione di una sottostringa

(`${var#pattern}`)

Esempio: rimozione di una stringa inesistente.

```
echo ${s#def}
abcABC123ABCabc
```

```
s: abcABC123ABCabc
```

Il pattern è **`def`**. Esso non è presente in **`s`**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa inesistente.

```
echo ${s#def}
abcABC123ABCabc
```

```
s: abcABC123ABCabc
```

Il pattern non corrisponde con l'inizio della stringa.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa inesistente.

```
echo ${s#def}
abcABC123ABCabc
```

s: abcABC123ABCabc

Pertanto, viene stampata l'intera stringa **s**, ovvero **abcABC123ABCabc**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa non corrispondente.

```
echo ${s#ABC}
abcABC123ABCabc
```

s: abcABC123ABCabc

Il pattern è **ABC**. Esso è presente in diversi punti di **s**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa non corrispondente.

```
echo ${s#ABC}
abcABC123ABCabc
```

```
s: abcABC123ABCabc
```

Il pattern non corrisponde con l'inizio della stringa.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa non corrispondente.

```
echo ${s#ABC}
abcABC123ABCabc
```

s: abcABC123ABCabc

Pertanto, viene stampata l'intera stringa **s**, ovvero **abcABC123ABCabc**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa esatta.

```
echo ${s#abc}
ABC123ABCabc
```

s: **abc**ABC123ABC**abc**

Il pattern è **abc**. Esso è presente in diversi punti di **s**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa esatta.

```
echo ${s#abc}
ABC123ABCabc
```

```
s: abcABC123ABCabc
```

La ricerca di **abc** avviene all'inizio di **s**.

Rimozione di una sottostringa

(`${var#pattern}`)

Esempio: rimozione di una stringa esatta.

```
echo ${s#abc}
ABC123ABCabc
```

`s:` **abc**`ABC123ABCabc`

Il primo pattern **`abc`** trovato è questo.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa esatta.

```
echo ${s#abc}
ABC123ABCabc
```

s: **abc**ABC123ABCabc

Il match è minimale; pertanto, ci si ferma alla più piccola stringa possibile che contiene **abc**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa esatta.

```
echo ${s#abc}
ABC123ABCabc
```

```
s: abcABC123ABCabc
```

Si rimuove il match, ottenendo la stringa che sarà stampata.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s#a*c}
ABC123ABCabc
```

```
s: abcABC123ABCabc
```

Il pattern è **a*c**. Esso rappresenta tutte le sottostringhe che:

- iniziano con la lettera **a**;
- continuano con una sequenza anche nulla di caratteri;
- terminano con la lettera **c**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s#a*c}
ABC123ABCabc
```

Diverse sottostringhe soddisfano il pattern.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s#a*c}
ABC123ABCabc
```

```
s: abcABC123ABCabc
```

La ricerca di **abc** inizia all'inizio di **s**.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s#a*c}
ABC123ABCabc
```

Esistono due pattern **a*c** candidati.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s#a*c}
ABC123ABCabc
```

```
s: abcABC123ABCabc
```

Il match è minimale; pertanto, si seleziona la sottostringa più piccola che soddisfa il pattern.

Rimozione di una sottostringa

(${var#pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s#a*c}
ABC123ABCabc
```

```
s: abcABC123ABCabc
```

Si rimuove il match, ottenendo la stringa che sarà stampata.

Rimozione di una sottostringa

(`${var#pattern}`)

La variabile **`s`** è rimasta invariata.

```
echo $s
abcABC123ABCabc
```

Rimozione di una sottostringa

(`${var##pattern}`)

L'espansione **`${var##pattern}`** rimuove il match più grande (**greedy removal**) di **`pattern`** all'inizio di **`var`**. Per il resto, è equivalente a **`${var#pattern}`**.

Rimozione di una sottostringa

(${var##pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s##a*c}
```

Qui, tra i due candidati si sceglie il più lungo, che viene poi rimosso. Si ottiene la stringa nulla.

Rimozione di una sottostringa

(${var%pattern}, ${var%%pattern})

Le due espansioni seguenti rimuovono rispettivamente il match più piccolo (non greedy) e più grande (greedy) di **pattern** alla fine di **var**.

${var%pattern}

${var%%pattern}

La ricerca di **pattern** parte dalla fine di **var**.

Per il resto, **%** è equivalente a **#**.

Rimozione di una sottostringa

(${var%pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s%a*c}
abcABC123ABC
```

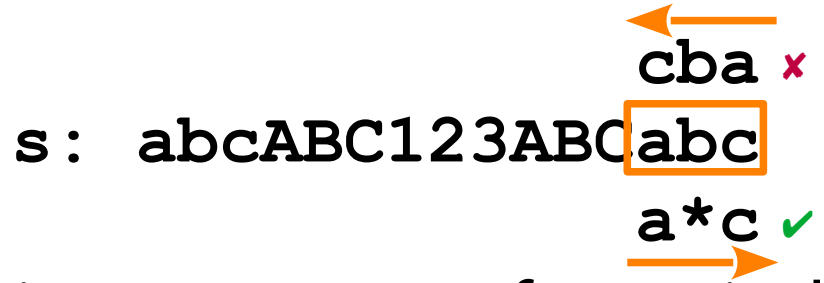
Qui, tra i due candidati si sceglie il più breve, che viene poi rimosso. Si ottiene la stringa **abcABC123ABC**.

Rimozione di una sottostringa

(${var%pattern})

Esempio: rimozione di una stringa descritta da pattern.

```
echo ${s%a*c}
abcABC123ABC
```



ATTENZIONE! Pattern e stringa sono confrontati da sinistra verso destra.

Esercizio 4 (2 min.)

Sia **`f`** una variabile contenente il valore **`image.jpg`**. Si chiede di produrre una serie di trasformazioni che cambino il valore di **`f`** in **`image.png`**.

OPERAZIONI MATEMATICHE

Definizione

(Insieme di variabili, usate per personalizzare le applicazioni)

BASH mette a disposizione una **espansione aritmetica** per interpretare espressioni aritmetiche.

La matematica gestita da BASH è esclusivamente intera. Qualunque operazione più complessa richiede applicazioni esterne.

- GNOME Calculator (calcolatrice grafica).
- **`bc`** (interprete matematico da linea di comando).
- Octave (clone di Matlab).
- …

L'operatore **$ ((EXPR))**

(Interpreta **EXPR** in senso aritmetico)

L'operatore **$ ((EXPR))** interpreta l'espressione aritmetica **EXPR** e ne restituisce il risultato.

Le operazioni aritmetiche disponibili sono le seguenti.

Addizione (**+**), Sottrazione (**–**), Moltiplicazione (*****), Divisione (**/**), Resto (**%**), elevazione a potenza (******).

Pre/Post incremento/decremento (**++**, **––**).

Shift bit a bit sinistro (**<<**) e destro (**>>**).

Uguaglianza (**==**) e disuguaglianza (**!=**).

Confronto (**<**, **>**, **<=**, **>=**).

Operatori bit a bit AND (**&**), OR (**|**) , XOR (**^**), NOT (**~**).

Operatori logici AND (&&), OR (**| |**), NOT (**!**).

L'operatore `$ ( (EXPR) )`

(Interpreta **`EXPR`** in senso aritmetico)

Gli operandi possono essere costanti.

```
echo $((1+2))
3
```

Gli operandi possono essere anche valori di variabili (non necessariamente con attributo intero).

```
a=1
b=2
echo $((a+b))
3
```

L'operatore `$((EXPR))`

(Interpreta **`EXPR`** in senso aritmetico)

Si possono annidare le espressioni.

```
echo $((1+$((1+1))))
3
```

Assegnazione ad una variabile

(Si usa lo statement **`let`**)

Per assegnare ad una variabile **`var`** il risultato di una espressione aritmetica si può applicare l'espansione aritmetica.

```
var=$((1+2))
```

Lo statement **`let`**, del tutto equivalente, è di gran lunga più elegante.

```
let var=1+2
a=1;b=2;let c=a+b
```

Esercizio 5 (2 min.)

Definite una variabile **`a`** con attributo intero e valore a vostra scelta. Scrivete uno statement che permetta di stabilire se **`$a`** sia un numero pari o dispari.

AMBIENTE

Definizione

(Insieme di variabili, usate per personalizzare le applicazioni)

BASH mette a disposizione un **ambiente**.
Ambiente: insieme di **variabili** (dette **variabili di ambiente**) che configurano il comportamento delle applicazioni.

Le variabili di ambiente possono essere **builtin** (definite direttamente da BASH) oppure **esterne** (impostate esternamente dall'utente per specifiche applicazioni esterne).

Quando parte un'applicazione, essa eredita l'intero ambiente.

Alcune variabili di ambiente builtin

(Configurazione di BASH)

`PATH:` elenco di percorsi di ricerca degli eseguibili (separato dal carattere **`:`**).

`PS1:` prompt dei comandi.

`PWD:` directory corrente.

`SHELL`: il percorso dell'interprete corrente.

`USERNAME:` lo username dell'utente corrente.

`HOSTNAME:` il nome dell'host corrente.

`HOME:` il percorso della home directory dell'utente.

`LANG:` nazionalità delle impostazioni locali.

Alcune variabili di ambiente esterne

(Configurazione di applicazioni esterne a BASH)

`LESS`: elenco di opzioni del visore **`less`**.

`DISPLAY`: schermo usato dal server grafico **`XOrg`**.

`TZ`: fuso orario usato dall'utility di sistema **`date`**.

…

Stampa variabili

(Sono variabili; si usa il comando **`echo`**)

Le variabili di ambiente sono variabili come tutte le altre. Possono essere stampate con il comando **`echo`**.

Ad esempio, per stampare il valore della variabile di ambiente **`PS1`**:

```
echo $PS1
```

Stampa intero ambiente

(Si usa il comando **`env`**)

Per stampare l'intero ambiente si può usare il comando **`env`**.

```
env
```

L'output del comando è un elenco di variabili di ambiente interne ed esterne (una per riga).

Esercizio 6 (2 min.)

Eseguite i due comandi seguenti:

```
ls file_non_esistente
LANG=C ls file_non_esistente
```

Notate qualcosa di diverso nell'output dei due programmi?

OPERATORI CONDIZIONALI E LOGICI

Definizione

(Insieme di variabili, usate per personalizzare le applicazioni)

BASH mette a disposizione **operatori condizionali**.
Operatore condizionale: operatore unario o binario che ritorna vero o falso a seconda del verificarsi di una condizione logica.

1 < 2 → Vero
2 < 1 → Falso
…

Espressione condizionale

(Espressione con uno o più operatori condizionali)

La sintassi più generale di una espressione condizionale è la seguente:

`EXPR1 OPERATORE EXPR2` (due argomenti)

`OPERATORE EXPR1` (un argomento)

dove:

`EXPR1`, **`EXPR2`** sono operandi (costanti, variabili);

`OPERATORE` è uno specifico operatore condizionale.

Alcuni esempi
(Di espressione condizionale)

Confronto aritmetico.

`ARG1 -eq ARG2`	VERO se ARG1=ARG2
`ARG1 -ne ARG2`	VERO se ARG1≠ARG2
`ARG1 -lt ARG2`	VERO se ARG1<ARG2
`ARG1 -gt ARG2`	VERO se ARG1>ARG2
`ARG1 -le ARG2`	VERO se ARG1≤ARG2
`ARG1 -ge ARG2`	VERO se ARG1≥ARG2

Alcuni esempi

(Di espressione condizionale)

Confronto tra stringhe.

`-z STR`	VERO se STR ha lunghezza zero
`-n STR`	VERO se STR è non nulla
`STR`	VERO se STR non ha lunghezza zero
`STR1 == STR2`	VERO se STR1 e STR2 sono uguali
`STR1 != STR2`	VERO se STR1 è diversa da STR2
`STR1 \< STR2`	VERO se STR1<STR2
`STR1 \> STR2`	VERO se STR1>STR2

Alcuni esempi

(Di espressione condizionale)

Confronto logico (booleano).

`EXPR`	ritorna il valore logico di EXPR
`! EXPR`	VERO se EXPR ha valore FALSO
`EXPR1 -a EXPR2`	VERO se EXPR1 e EXPR2 lo sono
`EXPR1 -o EXPR2`	VERO se EXPR1 o EXPR2 lo sono

Test di una condizione

(Si può svolgere in diversi modi)

BASH mette a disposizione due sintassi equivalenti per effettuare il test di una condizione logica.

Si può usare il comando **`test`**:

```
test ESPRESSIONE
```

Si può incastrare l'espressione tra parentesi quadre:

```
[ ESPRESSIONE ]
```

Risultato del test

(È memorizzato nella variabile speciale **`$?`**)

Il risultato del test è memorizzato nella variabile speciale **`$?`**, che in generale contiene il codice di uscita di un comando.

Valore 0: VERO (uscita senza errore)
Altrimenti: FALSO (uscita con errore)

Un esempio concreto

(Vale più di 1000 parole)

Per testare il valore dell'espressione condizionale **`1 < 2`**, si può digitare:

```
test 1 -lt 2
```

Per stampare il valore di verità del confronto, si può digitare:

```
echo $?
```

Si ottiene l'output seguente, corrispondente a VERO:

```
0
```

Esercizio 7 (3 min.)

Effettuate il test della condizione seguente nei due modi previsti da BASH:

`stringa1 > stringa2`.

Stampate il risultato del test in entrambi i casi.

CONTROLLO DI FLUSSO

Definizione

(Ordine in cui le istruzioni di un programma sono eseguite)

BASH mette a disposizione diversi costrutti per il **controllo del flusso**.
Controllo del flusso: è l'ordine in cui le istruzioni di un programma sono eseguite.

Il linguaggio offerto da BASH è **imperativo**.

Il controllo del flusso è esplicitato dall'utente tramite i costrutti.

Definizione

(Ordine in cui le istruzioni di un programma sono eseguite)

BASH mette a disposizione diversi costrutti per il **controllo del flusso**.
Controllo del flusso: è l'ordine in cui le istruzioni di un programma sono eseguite.

Altri linguaggi sono **dichiarativi**.

L'utente esprime le proprietà che deve avere una soluzione.
Il controllo di flusso è regolato internamente da un risolutore che calcola la soluzione.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
```

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
```

Il costrutto è delimitato dalle stringhe **`if`** e **`fi`**.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
```

Questi sono i test condizionali visti prima, ad esempio `[ 1 -lt 2 ]`.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
...
```

Se il test **COND_TEST1** è VERO, si esegue gli statement **STATEMENTS1** e si esce.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
```

Se il test **COND_TEST1** è FALSO, si salta al ramo elif.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
...
```

Se il test **COND_TEST2** è VERO, si esegue gli statement **STATEMENTS2** e si esce.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
```

Se il test **COND_TEST2** è FALSO, si salta al ramo else.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
...
```

Si esegue gli statement **STATEMENTS3** e si esce.

Costrutto if

(L'if-then-else presente altri linguaggi)

Il costrutto if ha la seguente sintassi generale.

```
if COND_TEST1; then
   STATEMENTS1
elif COND_TEST2; then
   STATEMENTS2
else
   STATEMENTS3
fi
```

I token **`then`** separano i test condizionali dagli statement associati.

Un esempio concreto

(Contribuisce, si spera, a chiarire il costrutto)

Si definisce una variabile **a** con attributo intero e valore pari a **12**.

```
declare -i a=12
```

Si vuole stampare un messaggio diverso in funzione dell'appartenenza di **a** ad uno specifico intervallo di valori.

Ad esempio:

a ∈ [0,10] → "a è compreso tra 0 e 10";
a ∈ [11,20] → "a è compreso tra 11 e 20";
altrimenti → "a non è compreso tra 0 e 20".

Il costrutto if richesto

(Scritto per bene, su più righe)

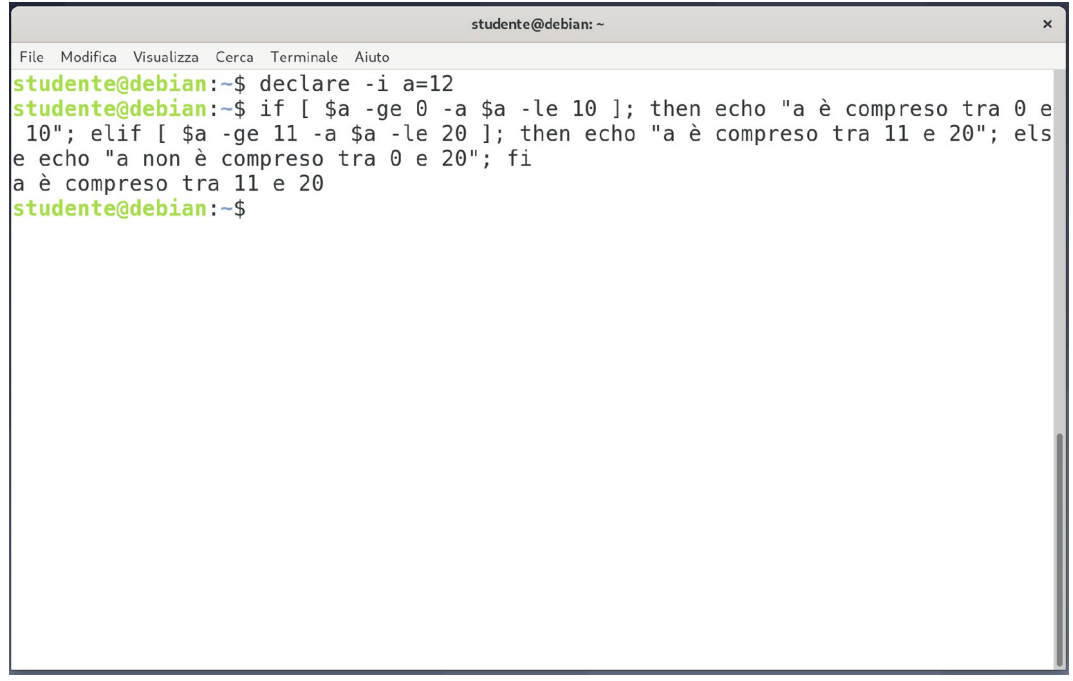
```
if [ $a -ge 0 -a $a -le 10 ]; then
  echo ”a è compreso tra 0 e 10”;
elif [ $a -ge 11 -a $a -le 20 ]; then
  echo ”a è compreso tra 11 e 20”;
else
  echo “a non è compreso tra 0 e 20”;
fi
```

Come inserire una roba del genere al terminale?

Inserimento su una singola riga

(La soluzione più immediata e più atroce)

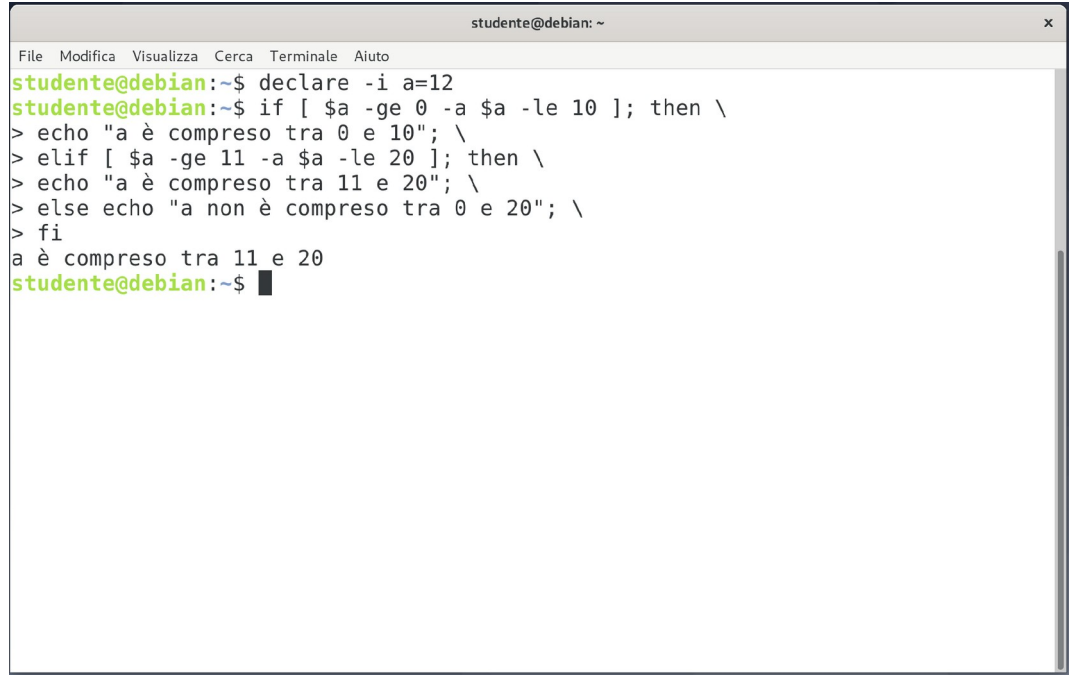
Si scrive l'intero comando su una riga.
In bocca al lupo con i costrutti annidati!



Inserimento multilinea

(Si spezza il comando in più righe con backslash)

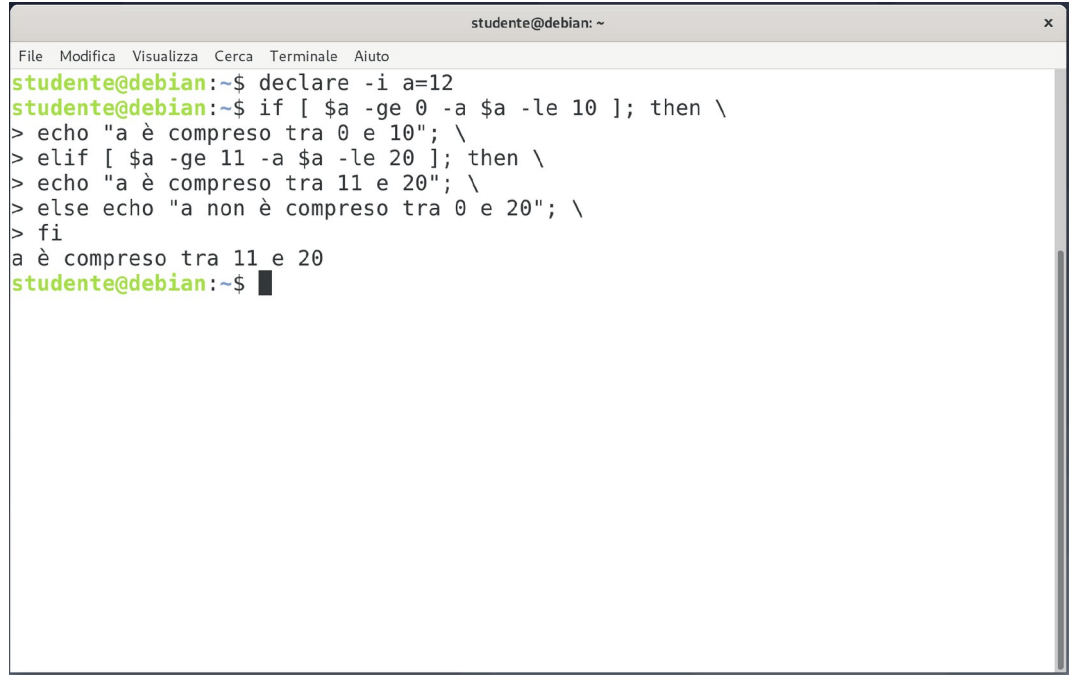
Si usa il backslash \ per spezzare il comando e indicare che continuerà alla prossima riga.



Inserimento multilinea

(Si spezza il comando in più righe con backslash)

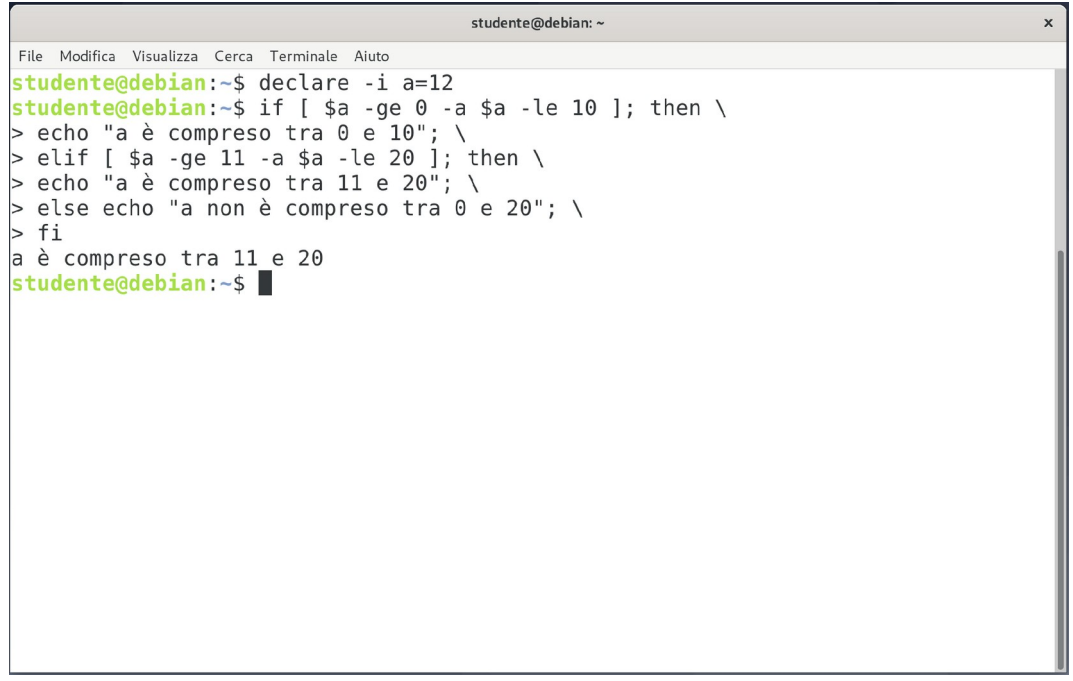
È necessario terminare gli statement del corpo con il punto e virgola `;`.



Inserimento multilinea

(Si spezza il comando in più righe con backslash)

Si noti come la continuazione del comando sia segnalata ad inizio riga dal carattere **>**.



Esercizio 8 (4 min.)

Definite una variabile **`a`** con attributo intero e valore pari a **`55`**.
Definite una variabile **`b`** con attributo intero. Leggete un valore a caso di **`b`** da terminale.
Scrivete un costrutto if con i rami seguenti:

se a > b, stampate "a è maggiore di b".
se a < b, stampate "a è minore di b".
se a=b, stampate "a è uguale a b".

Costrutto case

(Lo switch-case presente altri linguaggi)

Il costrutto case ha la seguente sintassi generale.

```
case $var in
VAL1)
   STATEMENTS1
;;
VAL2)
   STATEMENTS2
;;
VAL3)
   STATEMENTS3
;;
esac
```

Costrutto case

(Lo switch-case presente altri linguaggi)

Il costrutto case ha la seguente sintassi generale.

```
case $var in
VAL1)
   STATEMENTS1
;;
VAL2)
   STATEMENTS2
;;
VAL3)
   STATEMENTS3
;;
esac
```

Il costrutto è delimitato dalle stringhe **`case`** e **`esac`**.

Costrutto case

(Lo switch-case presente altri linguaggi)

Il costrutto case ha la seguente sintassi generale.

```
case $var in
VAL1)
   STATEMENTS1
;;
VAL2)
   STATEMENTS2
;;
VAL3)
   STATEMENTS3
;;
esac
```

var è una variabile.

Il token **in** separa la variabile dall'elenco dei valori.

Costrutto case

(Lo switch-case presente altri linguaggi)

Il costrutto case ha la seguente sintassi generale.

```
case $var in
VAL1)
    STATEMENTS1
;;
VAL2)
    STATEMENTS2
;;
VAL3)
    STATEMENTS3
;;
esac
```

Se **var==VAL1** si eseguono gli statement **STATEMENTS1** e si esce.

Costrutto case

(Lo switch-case presente altri linguaggi)

Il costrutto case ha la seguente sintassi generale.

```
case $var in
VAL1)
   STATEMENTS1
;;
VAL2)
   STATEMENTS2
;;
VAL3)
   STATEMENTS3
;;
esac
```

Se **var==VAL2** si eseguono gli statement **STATEMENTS2** e si esce.

Costrutto case

(Lo switch-case presente altri linguaggi)

Il costrutto case ha la seguente sintassi generale.

```
case $var in
VAL1)
   STATEMENTS1
;;
VAL2)
   STATEMENTS2
;;
VAL3)
   STATEMENTS3
;;
esac
```

Se **var==VAL3** si eseguono gli statement **STATEMENTS3** e si esce.

I valori nel costrutto case

(Diverse tipologie sono possibili)

I valori usabili nel costrutto case possono essere i seguenti.

Costanti alfanumeriche.

Il carattere asterisco *****, usato nell'ultimo caso di default, che matcha ogni altra stringa possibile.

Una wildcard che rappresenta un insieme di stringhe, ad es. 9[0-9].

Un esempio concreto

(Contribuisce, si spera, a chiarire il costrutto)

Si definisce una variabile **a** con attributo intero e valore pari a **12**.

```
declare -i a=12
```

Si vuole stampare un messaggio diverso in funzione del valore del primo carattere di **a**.

Ad esempio:

a inizia con 1 → ″a inizia con 1″;
a inizia con 2 → ″a inizia con 2″;
altrimenti → ″a non inizia né con 1 né con 2″.

Il costrutto case richesto

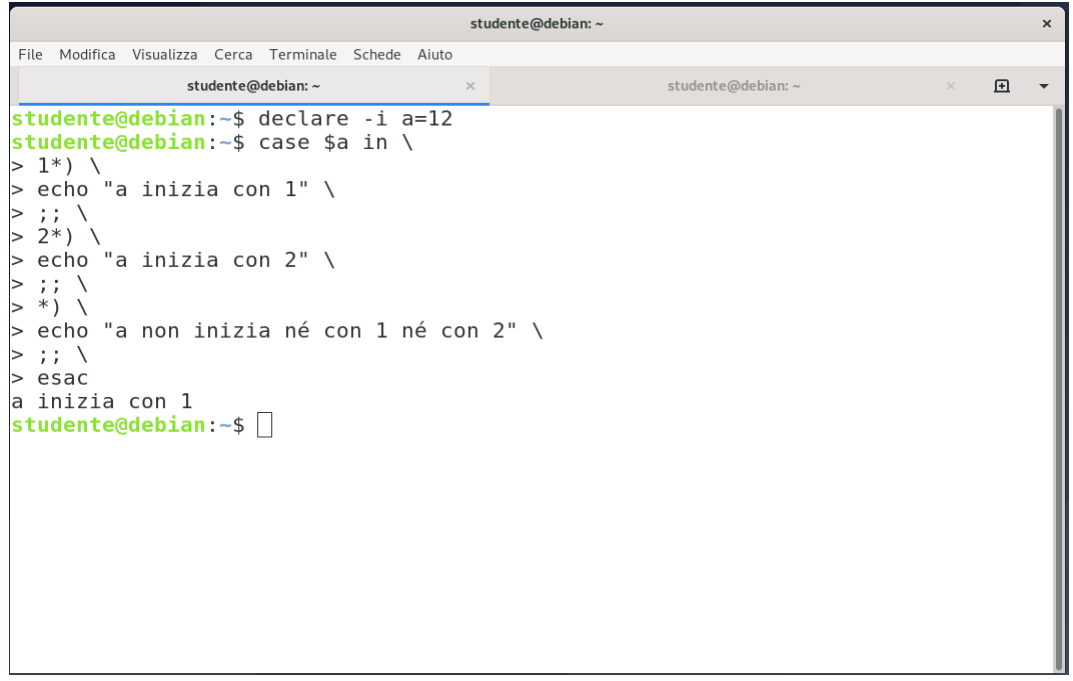
(Scritto per bene, su più righe)

```
case $a in
1*)
   echo ”a inizia con 1”
;;
2*)
   echo ”a inizia con 2”
;;
*)
   echo ”a non inizia né con 1 né con 2”;
;;
esac
```

L'output del comando

(Comando multilinea)

Si usa il backslash \ per spezzare il comando e indicare che continuerà alla prossima riga.



Esercizio 9 (4 min.)

Definite una variabile **`a`** ad un valore qualunque tra start, stop e restart.
Scrivete un costrutto case con i rami seguenti:

se **`a = start`**, stampate "system started".
se **`a = stop`**, stampate "system stopped".
se **`a = "restart"`**, stampate "system stopped" e "system started".
in tutti gli altri casi, stampate "wrong command".

Costrutto while

(Il while presente altri linguaggi)

Il costrutto while ha la seguente sintassi generale.

```
while COND_TEST1;
do
   STATEMENTS1
done
```

Costrutto while

(Il while presente altri linguaggi)

Il costrutto while ha la seguente sintassi generale.

```
while COND_TEST1;
do
   STATEMENTS1
done
```

Il corpo del ciclo è delimitato dalle stringhe **do** e **done**.

Costrutto while

(Il while presente altri linguaggi)

Il costrutto while ha la seguente sintassi generale.

```
while COND_TEST1;
do
   STATEMENTS1
done
```

COND_TEST1 è un test condizionale. Il token **;** separa la condizione dal corpo del ciclo.

Costrutto while

(Il while presente altri linguaggi)

Il costrutto while ha la seguente sintassi generale.

```
while COND_TEST1;
do
   STATEMENTS1
done
```

Fin quando **COND_TEST1** è VERO, si eseguono gli statement **STATEMENTS1**.
Il test condizionale è interpretato prima dell'eventuale esecuzione del corpo.

Costrutto while

(Il while presente altri linguaggi)

Il costrutto while ha la seguente sintassi generale.

```
while COND_TEST1;
do
   STATEMENTS1
done
...
```

Non appena **COND_TEST1** è FALSO, si esce.

Un esempio concreto

(Contribuisce, si spera, a chiarire il costrutto)

Si definisce una variabile **a** con attributo intero e valore pari a **1**.

```
declare -i a=1
```

Si vuole stampare il valore di **a** ed incrementarlo fino a quando non si verifica **a > 10**.

Il costrutto while richesto

(Scritto per bene, su più righe)

```
while [ $a -le 10 ];
do
   echo $a
   let a=++a
done
```

ATTENZIONE! Ci vuole il pre-incremento (**++a**).

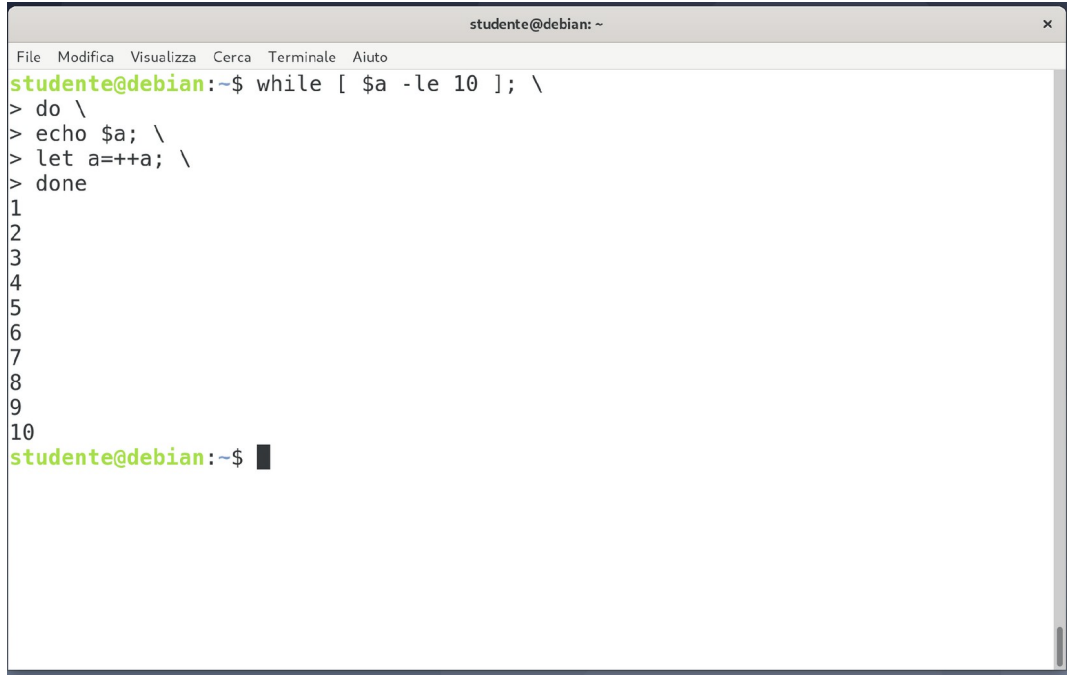
Prima si incrementa il valore di **a**, e poi lo si memorizza.

Memorizzare il valore attuale di **a** per poi incrementare non **a**, bensì il valore dell'espressione **a** (!) è un errore sottile.

L'output del comando

(Comando multilinea)

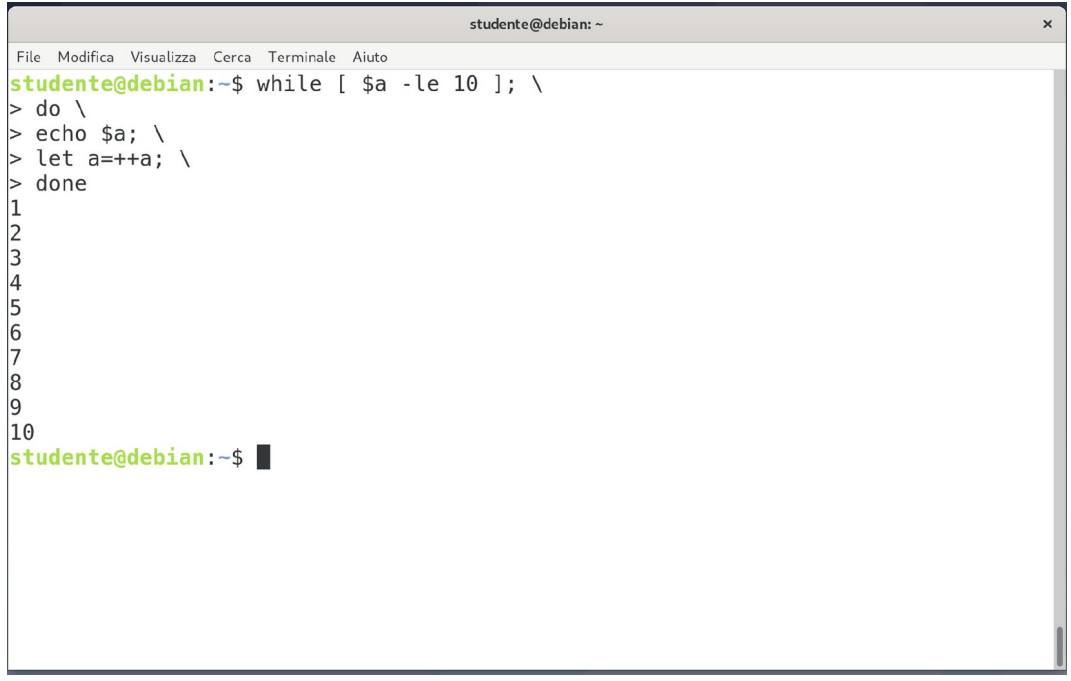
Si usa il backslash \ per spezzare il comando e indicare che continuerà alla prossima riga.



L'output del comando

(Comando multilinea)

È necessario terminare gli statement del corpo con il punto e virgola `;`.



Costrutto until

(Come l'until presente altri linguaggi)

Il costrutto until ha la seguente sintassi generale.

```
until COND_TEST1;
do
   STATEMENTS1
done
```

Costrutto until

(Come l'until presente altri linguaggi)

Il costrutto until ha la seguente sintassi generale.

```
until COND_TEST1;
do
   STATEMENTS1
done
```

Il corpo del ciclo è delimitato dalle stringhe **do** e **done**.

Costrutto until

(Come l'until presente altri linguaggi)

Il costrutto until ha la seguente sintassi generale.

```
until COND_TEST1;
do
   STATEMENTS1
done
```

COND_TEST1 è un test condizionale. Il token **;** separa la condizione dal corpo del ciclo.

Costrutto until

(Come l'until presente altri linguaggi)

Il costrutto until ha la seguente sintassi generale.

```
until COND_TEST1;
do
   STATEMENTS1
done
```

Fin quando **COND_TEST1** è FALSO, si eseguono gli statement **STATEMENTS1**.
Il test condizionale è interpretato prima dell'eventuale esecuzione del corpo.

Costrutto until

(Come l'until presente altri linguaggi)

Il costrutto until ha la seguente sintassi generale.

```
until COND_TEST1;
do
   STATEMENTS1
done
...
```

Non appena **COND_TEST1** è VERO, si esce.

Un esempio concreto

(Contribuisce, si spera, a chiarire il costrutto)

Si definisce una variabile **a** con attributo intero e valore pari a **1**.

```
declare -i a=1
```

Si vuole stampare il valore di **a** ed incrementarlo fino a quando non si verifica **a > 10**.

Il costrutto while richesto

(Scritto per bene, su più righe)

```
until [ $a -gt 10 ];
do
   echo $a
   let a=++a
done
```

ATTENZIONE! Ci vuole il pre-incremento (**++a**).

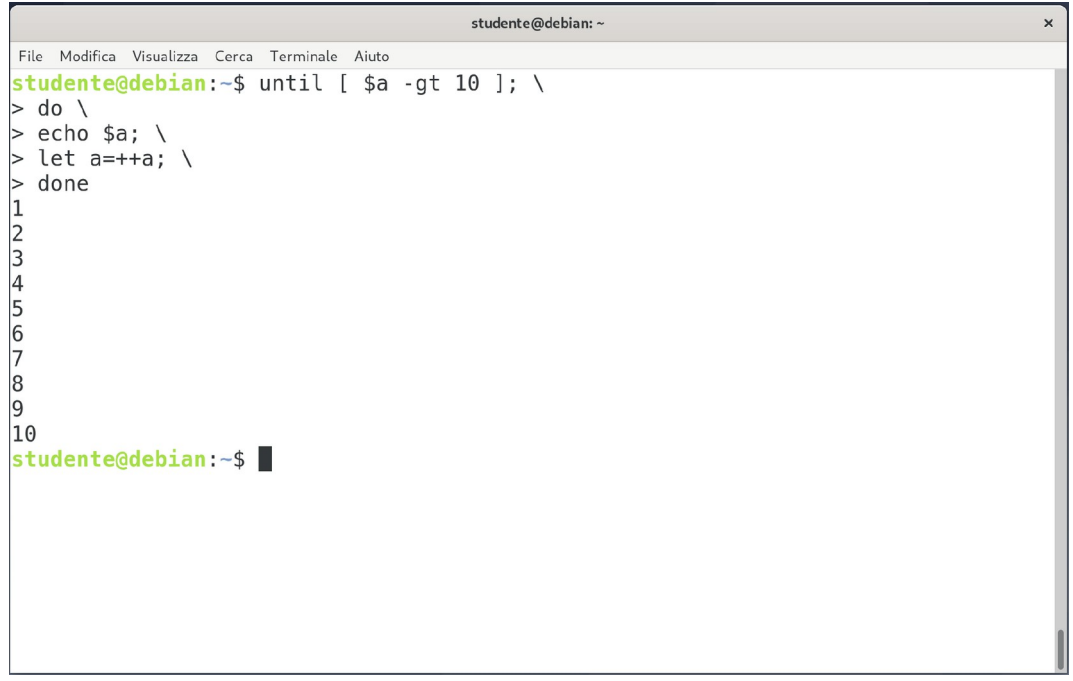
Prima si incrementa il valore di **a**, e poi lo si memorizza.

Memorizzare il valore attuale di **a** per poi incrementare non **a**, bensì il valore dell'espressione **a** (!) è un errore sottile.

L'output del comando

(Comando multilinea)

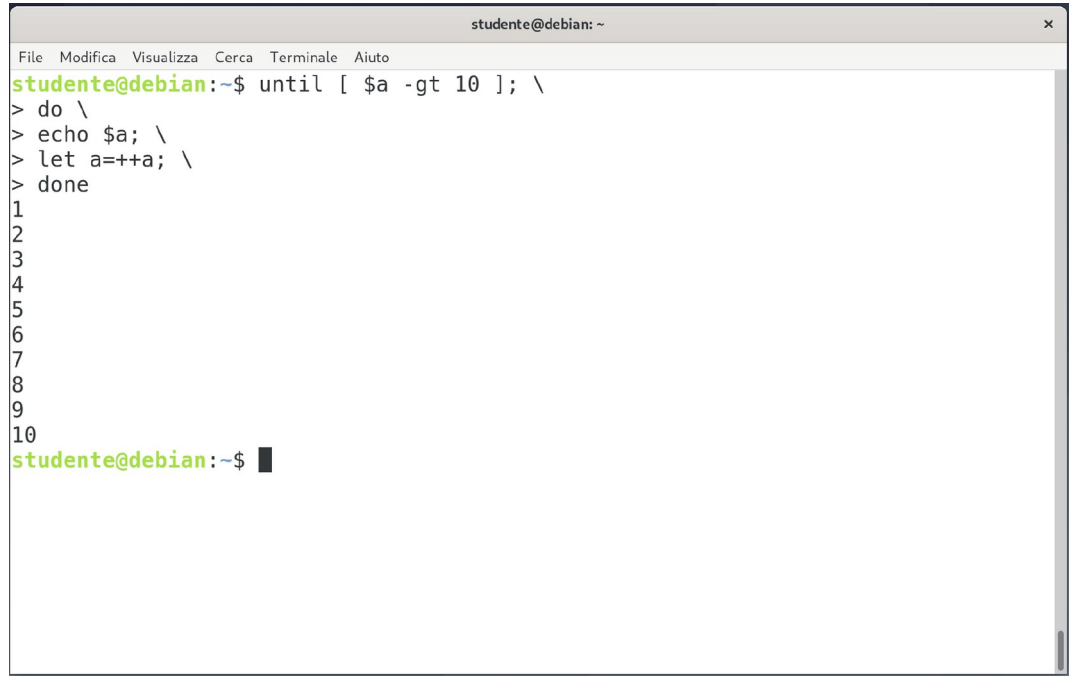
Si usa il backslash \ per spezzare il comando e indicare che continuerà alla prossima riga.



L'output del comando

(Comando multilinea)

È necessario terminare gli statement del corpo con il punto e virgola `;`.



Esercizio 10 (4 min.)

Definite una variabile **`a`** con attributo intero e valore pari a **`1`**.
Scrivete un ciclo while che stampa tutti i numeri dispari da 1 a 10.

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
```

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
```

Il corpo del ciclo è delimitato dalle stringhe **do** e **done**.

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
```

LIST è un elenco di stringhe, scritte così:
str1 str2 ... **strN**
Se contengono caratteri speciali, si possono quotare.

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
```

Il token `;` separa l'iterazione dal corpo del ciclo.

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
```

var è una variabile che riceve il valore i-mo di LIST all'i-ma iterazione del ciclo.

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
```

Ad ogni iterazione si eseguono gli statement **`STATEMENTS1`**.

Costrutto for

(Come il for presente altri linguaggi)

Il costrutto for ha la seguente sintassi generale.

```
for var in LIST;
do
   STATEMENTS1
done
...
```

Al termine dell'ultima iterazione, si esce.

Un esempio concreto

(Contribuisce, si spera, a chiarire il costrutto)

Si vogliono stampare i valori da 1 a 10.

A differenza degli esempi precedenti, qui non c'è bisogno di definire una variabile con attributo intero.

Il costrutto for itera su tutti gli elementi.

Il valore i-mo non viene confrontato con un limite superiore.

Il costrutto for richiesto

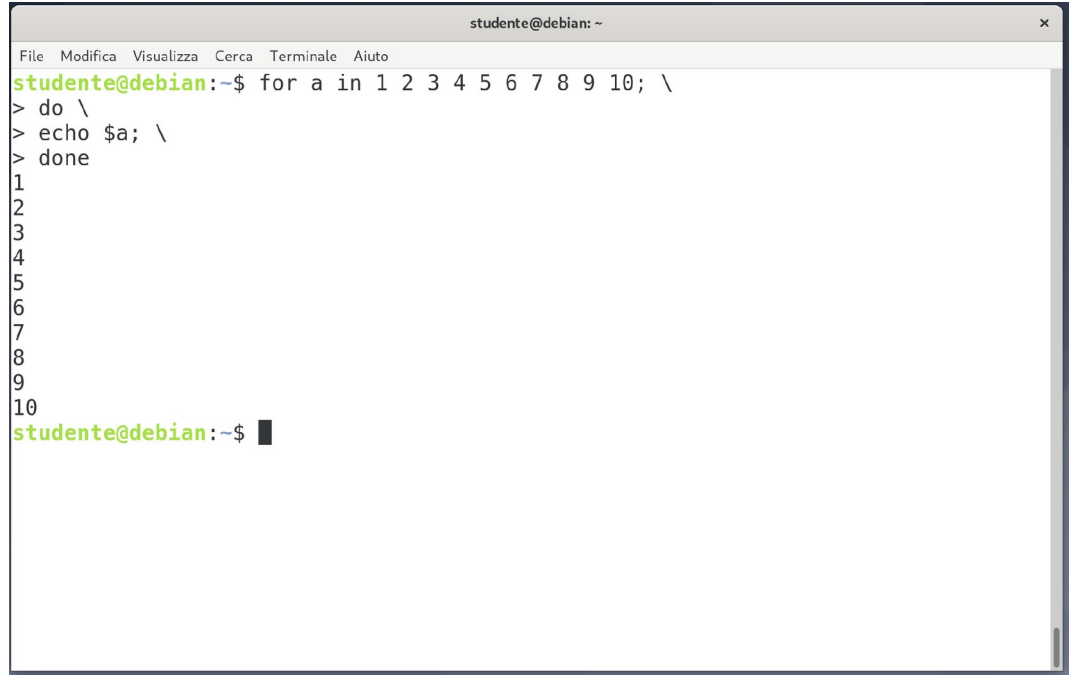
(Scritto per bene, su più righe)

```
for a in 1 2 3 4 5 6 7 8 9 10;
do
   echo $a
done
```

L'output del comando

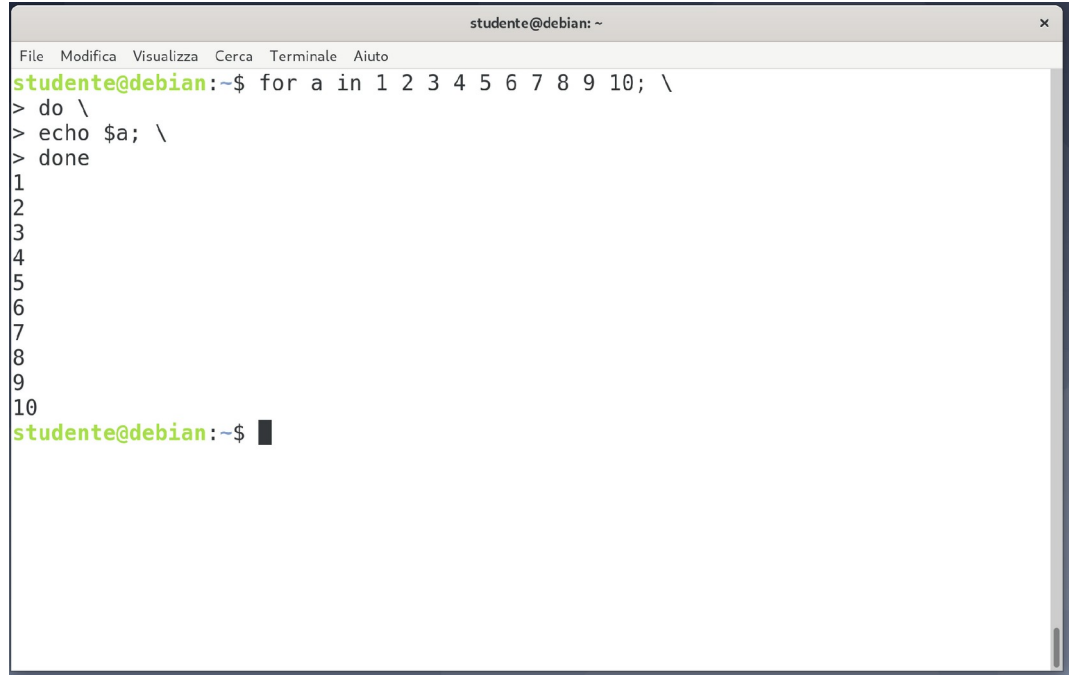
(Comando multilinea)

Si usa il backslash \ per spezzare il comando e indicare che continuerà alla prossima riga.



L'output del comando
(Comando multilinea)

È necessario terminare gli statement del corpo con il punto e virgola `;`.



Bisogna immettere tutti i valori?

(Che succede se il ciclo va da 1 a 1000?)

Una risposta confortante

(No, non è necessario!)

Non è necessario immettere a mano tutti i valori richiesti in un ciclo for. Esistono almeno due modi più comodi.

Espansione delle graffe (brace expansion).

La vediamo nella slide successiva.

Sostituzione di comando (command substitution).

La vedremo nella lezione sui processi.

Espansione delle graffe

(Utilissima per generare stringhe arbitrarie a partire da pattern)

BASH fornisce un meccanismo di **espansione tramite parentesi graffe** (**brace expansion**). Tale espansione consente di generare stringhe arbitrarie a partire da pattern. I pattern principali sono:

```
S1{S2,S3}S4   → S1S2S4 S1S3S4
{0..N}        → 0 1 2 ... N
{0..N..M}     → 0 0+M 0+2M ... N
```

L'espansione delle graffe consente di generare agevolmente sequenze di numeri e parametri con sottostringhe in comune.

Esempi di espansione delle graffe

(Molto comodi)

Per generare tutti i numeri da 0 a 100:

```
echo {0..100}
```

Per generare tutti i numeri dispari da 1 a 100:

```
echo {1..100..2}
```

Per generare le stringhe **`file-esempio.txt`**, **`file-vecchio.txt`**, ..., **`file-nuovo.txt`**:

```
echo file-{esempio,vecchio,nuovo}.txt
```

Esercizio 11 (2 min.)

Definite una variabile **`a`** con attributo intero e valore pari a **`1`**.
Scrivete un ciclo for che stampa i quadrati dei numeri da 1 a 10.

Manipolazione delle iterazioni

(AKA gli statement **`break`** e **`continue`**)

Il flusso naturale delle iterazioni in un costrutto di controllo del flusso può essere modificato con gli statement seguenti.

Break. Lo statement **`break`** esce immediatamente dal costrutto di controllo del flusso corrente.

Continue. Lo statement **`continue`** esegue immediatamente la prossima iterazione del costrutto di controllo del flusso corrente.

Un esempio concreto

(Manipolazione di un ciclo tramite **`break`**)

```
for a in 1 2 3 4 5; do
   if [ $a -eq 3 ]; then
      break
   fi
   echo $a
done
```

Output:

```
1
2
```

Un esempio concreto

(Manipolazione di un ciclo tramite **`continue`**)

```
for a in 1 2 3 4 5; do
   if [ $a -eq 3 ]; then
      continue
   fi
   echo $a
done
```

Output:

```
1
2
4
5
```

ESECUZIONE CONDIZIONATA

Definizione

(Ordine in cui le istruzioni di un programma sono eseguite)

BASH mette a disposizione diversi costrutti per l' **esecuzione condizionata** di comandi.
Esecuzione condizionata: un comando è eseguito solo se il precedente esibisce uno specifico comportamento.

I comportamenti considerati sono i seguenti.

Nessun comportamento.
Il comando precedente termina con stato nullo.
Il comando precedente termina con stato di uscita non nullo.

Concatenazione di comandi

(Si usa l'operatore `;`)

L'operatore `;` consente di concatenare più comandi. I comandi eseguono rigorosamente uno dopo l'altro, indipendentemente da terminazioni corrette o non.

```
ls nonesistente; echo prova
ls: impossibile accedere a 'nonesistente': File o directory non esistente
prova
```

Esecuzione condizionata al successo

(Si usa l'operatore **&&**)

L'operatore **&&** esegue il comando successivo solo se il precedente esce con uno stato di uscita nullo.

```
ls nonesistente && echo "Tutto OK"

ls /bin/bash && echo "Tutto OK"
/bin/bash
Tutto OK
```

Esecuzione condizionata al successo

(Si usa l'operatore **&&**)

L'operatore **||** esegue il comando successivo solo se il precedente esce con uno stato di uscita non nullo.

```
ls nonesistente || echo "Errore"
Errore

ls /bin/bash || echo "Errore"
/bin/bash
```

Esercizio 12 (1 min.)

Stampate il messaggio “Shell installate” se e solo se i due binari eseguibili **`/bin/bash`** e **`/bin/sh`** sono presenti.

SCRIPT

Definizione

(Insieme di comandi memorizzati in un file)

Immettere interattivamente una sequenza di comandi ogni volta che la si necessita è una operazione tediosa, incline agli errori ed inefficiente.
Per tale motivo, BASH permette l'esecuzione in modalità batch di una sequenza di comandi opportunamente memorizzata in un file.
Tale file prende il nome di **script di shell** (o **script**).

Linguaggi di scripting

(Permettono l'automazioni di operazioni; BASH ne fa parte)

Il concetto di script non vale solo per BASH, bensì per tutti i **linguaggi di scripting** (nei quali BASH ricade).

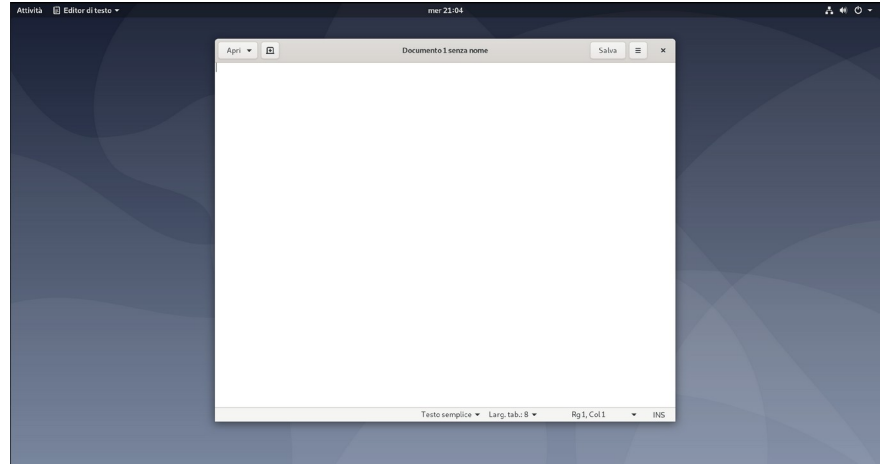
Linguaggio di scripting. È un linguaggio di programmazione che permette l'automazione di operazioni.

- Linguaggio interpretato (riga per riga o byte code).
- Linguaggio general-purpose.
- Linguaggio dinamico (diverse operazioni sono svolte a tempo di esecuzione, e non dal compilatore, ad esempio il type checking).

Creazione di uno script

(Una procedura passo passo)

Si avvia l'applicazione Gedit di GNOME.



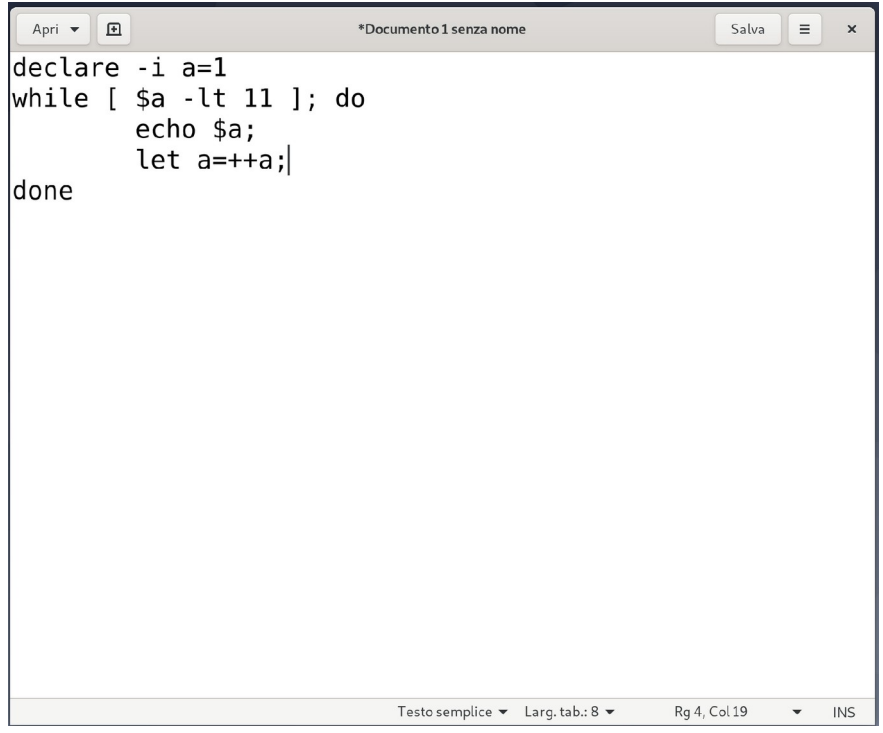
[SERMONE DA ANZIANO /on]
Sarebbe meglio se studiaste un editor come VIM o EMACS.
Gioverebbe alla vostra preparazione. Molto.
[SERMONE DA ANZIANO /off]

Creazione di uno script

(Una procedura passo passo)

Si scrivono i comandi seguenti nella finestra di Gedit.

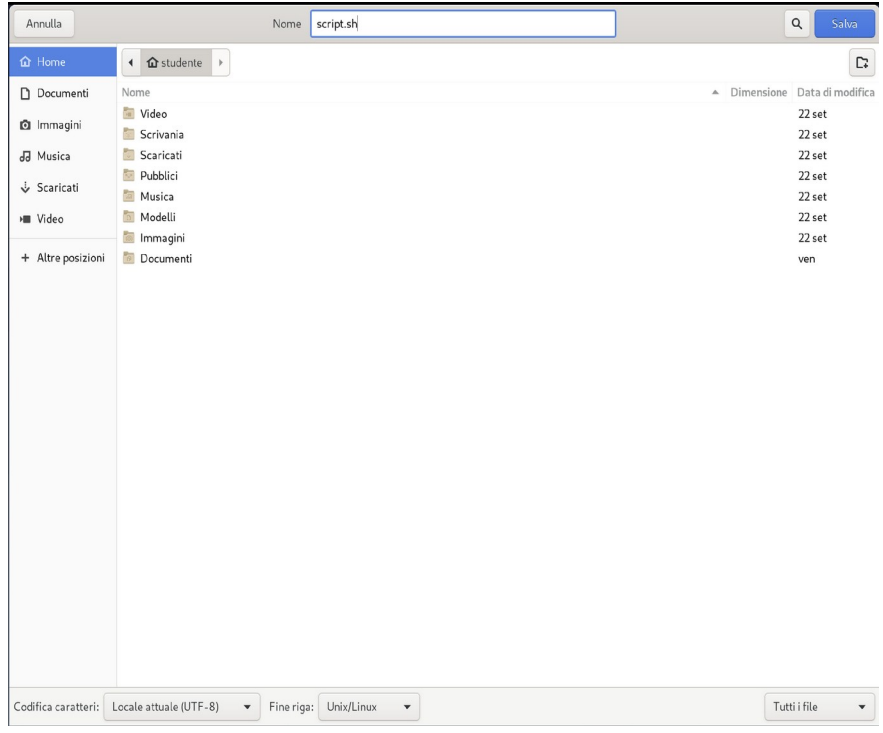
```
declare -i a=1
while [ $a -lt 11 ]; do
   echo $a;
   let a=++a;
done
```



Creazione di uno script

(Una procedura passo passo)

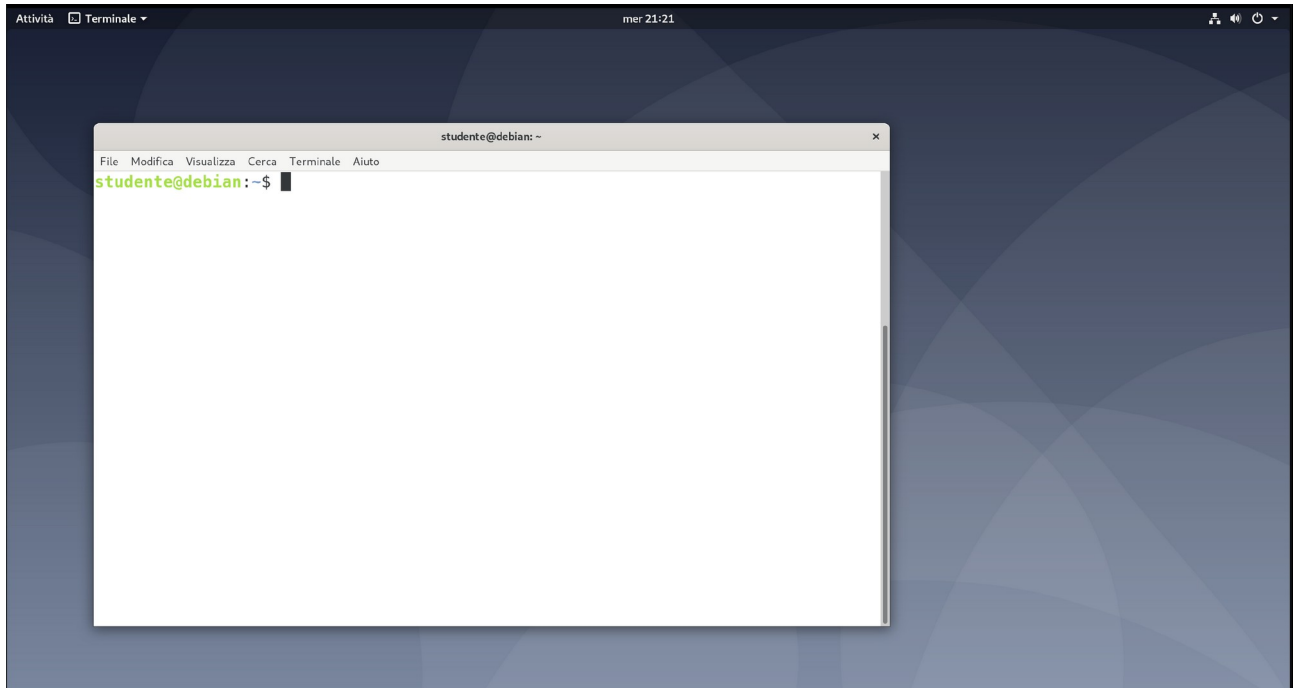
Si salva lo script con il nome di file **`script.sh`** e si esce da Gedit.



Creazione di uno script

(Una procedura passo passo)

Si avvia una istanza di GNOME Terminal.

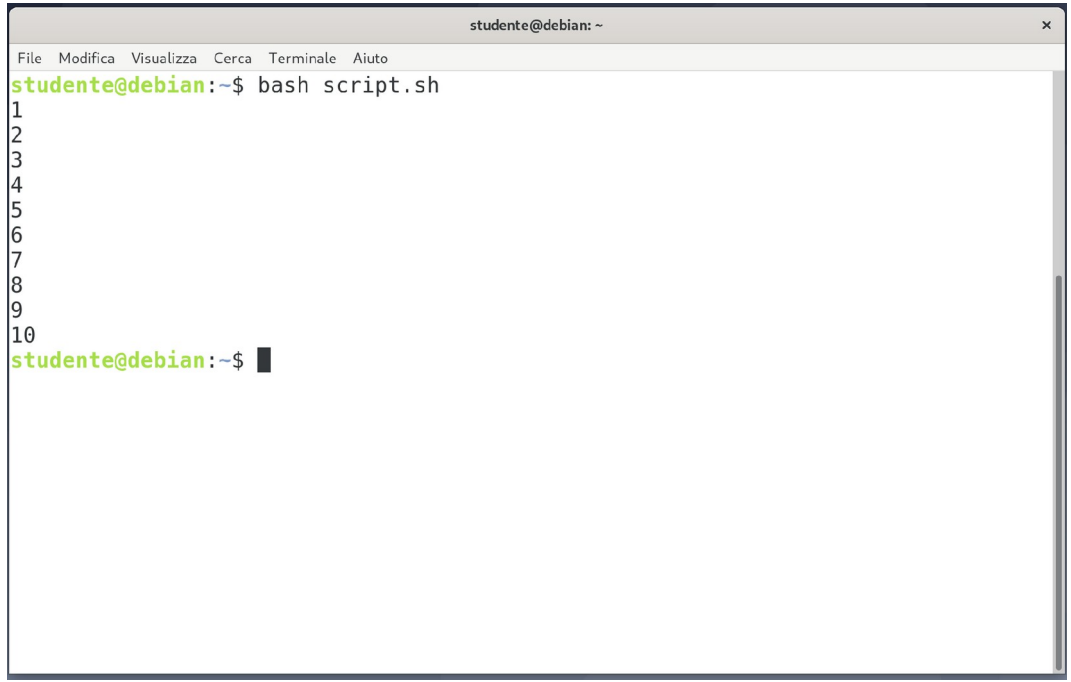


Creazione di uno script

(Una procedura passo passo)

Si esegue lo script tramite BASH:

`bash script.sh`



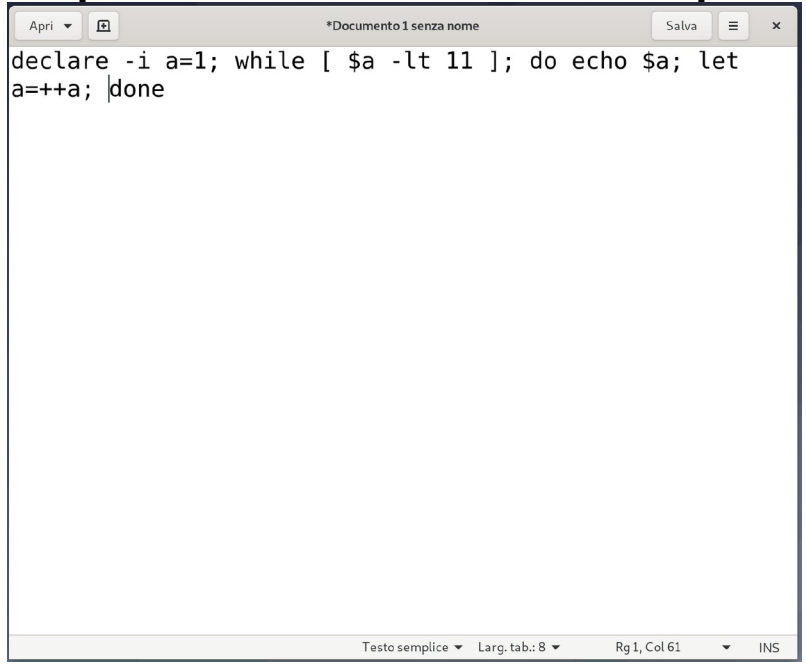
Formattazione negli script

(Atroce)

Tutti gli statement su una riga.

Ogni comando è separato dal carattere `;`.

Di solito è una pessima scelta (lo script è illeggibile).



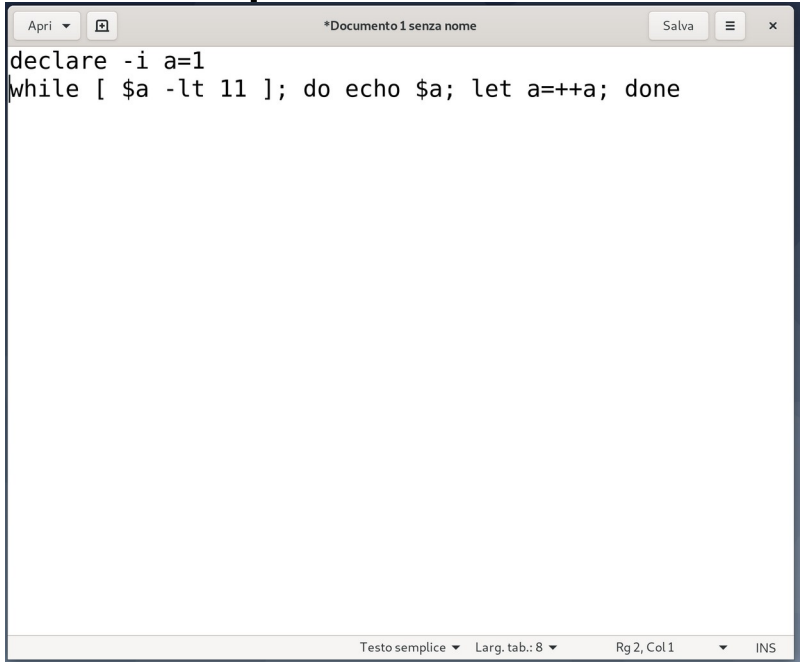
Formattazione negli script

(Mediocre)

Ogni statement su una riga, senza multiriga.

Non è necessario terminare lo statement con `;`.

Gli statement del corpo del ciclo devono terminare con `;`.

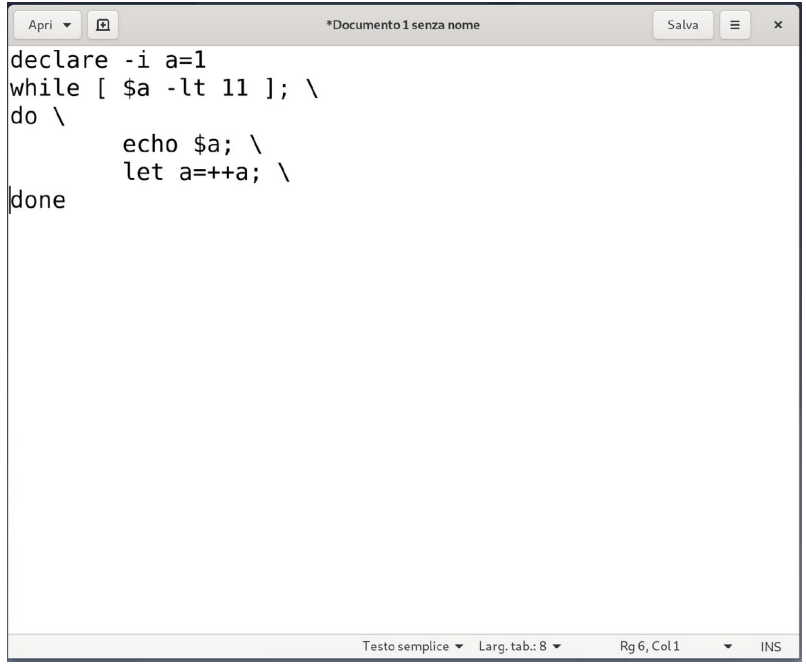


Formattazione negli script

(Normale)

Ogni statement su una riga, con multiriga.

Gli statement del corpo del ciclo devono terminare con `;`.

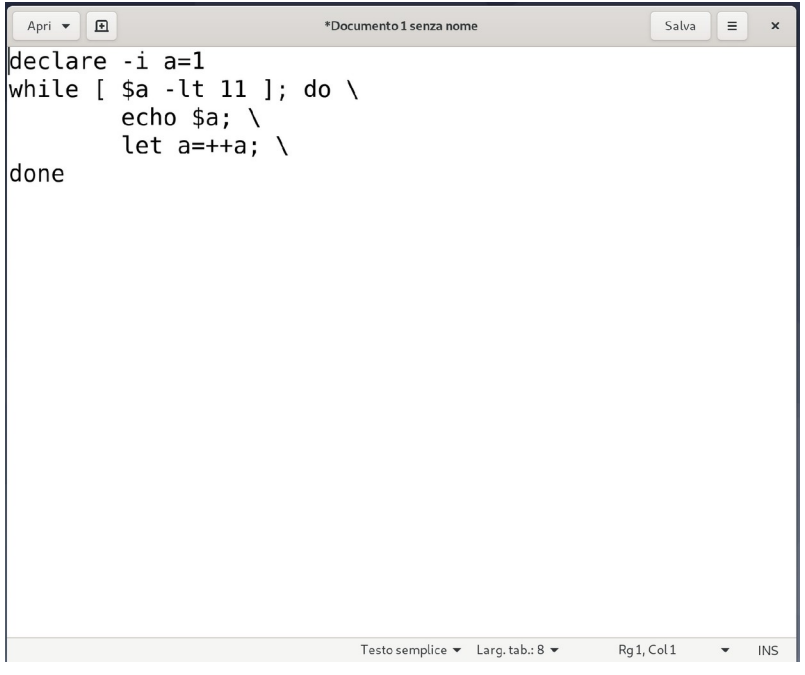


Formattazione negli script

(Normale)

Ogni statement su una riga, con multiriga.

Una variante comune nei costrutti di flusso vede il **`do`** sulla stessa riga del **`while`**/**`do`**/**`until`**.



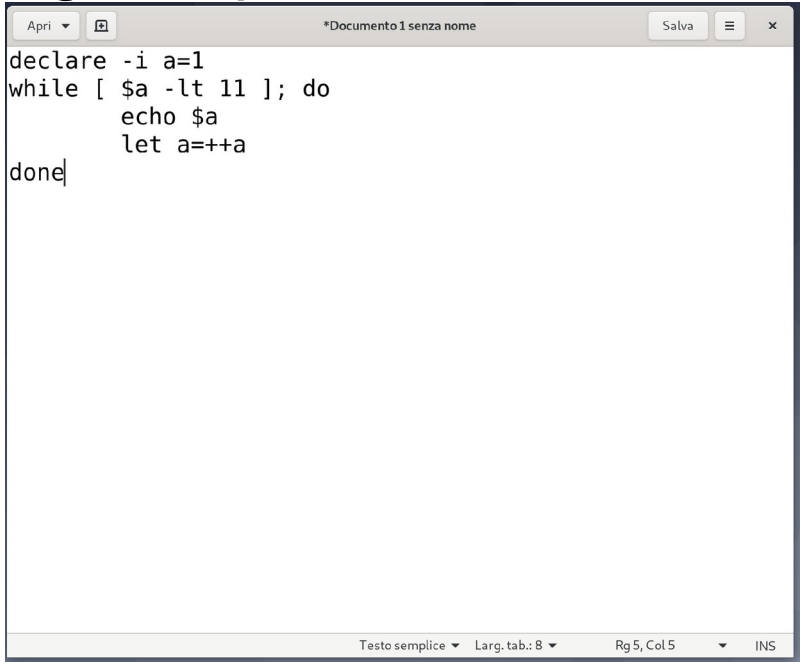
Formattazione negli script

(Ottimale)

Ogni statement su una riga, no multiriga, no `;`.

È possibile omettere multiriga e `;` nel corpo del ciclo.

Valido solo negli script e nei costrutti di controllo di flusso.

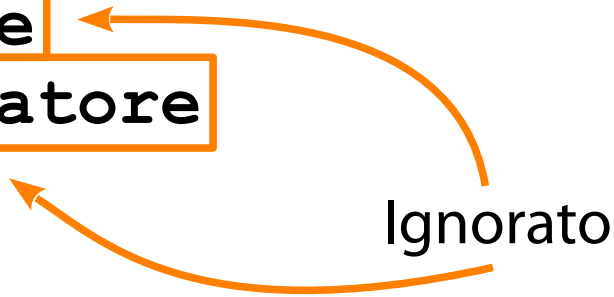


Commenti

(Introdotti dal carattere speciale #)

Il carattere **#** introduce un commento.
Tutto ciò che segue il commento fino a fine riga viene ignorato dal parser di BASH.

```
# Inizializzazione
a=1 # a è il contatore
b=2
```



I commenti si possono usare anche sul terminale.
Tuttavia, il loro uso più comune è negli script.

Parametri posizionali

(Si usano le variabili interne `$0`, `$1`, `$2`, …)

BASH mette a disposizione le variabili interne (o di builtin) `$0`, `$1`, `$2`, …, `$9`, `${10}`, `${11}`, … per la gestione dei **parametri posizionali**.

Parametro posizionale (o argomento): è una stringa che un utente fornisce ad uno script, al fine di definirne la modalità operativa e l'oggetto delle operazioni.

Il significato dei diversi parametri

(Analogo al linguaggio C)

`$0:` nome dello script.
`$1:` primo argomento.
`$2:` secondo argomento.
`$3:` terzo argomento.
…

Manipolazione dei parametri

(Analoga a quella delle variabili)

I parametri posizionali sono variabili; in quanto tali, sono soggetti a tutte le trasformazioni viste finora!

Ad esempio, l'espansione seguente stampa il primo parametro o, in sua assenza, una stringa opportuna (senza ridefinirlo):

```
echo ${1:-"valore di default"}
```

Un esempio concreto

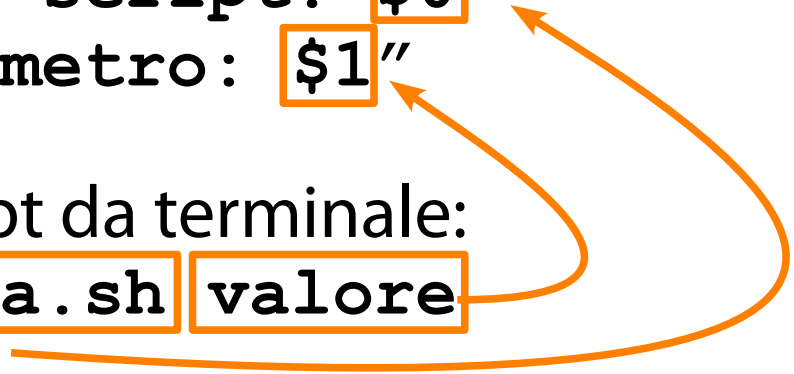
(Stampa di un parametro)

Si crei uno script **stampa.sh** con i comandi seguenti.

```
echo "Nome script: $0"
echo "Parametro: $1"
```

Si esegua lo script da terminale:

```
bash stampa.sh valore
```



Si ottiene l'output seguente:

```
Nome script: stampa.sh
Parametro: valore
```

Segnalazione stato di uscita

(0 → Tutto OK; > 0 → Errore)

Lo script è, dopotutto, un comando. È pertanto possibile (nonché buona prassi di programmazione) segnalare il suo stato lo stato di uscita.

- **0**: Tutto OK.
- **>0**: Si è verificato un errore (la scelta del valore è del programmatore).

La segnalazione avviene tramite il comando **`exit`**, che accetta come argomento lo stato di uscita.

```
exit 0
exit 1
```

Un esempio concreto

(Segnalazione di un errore tramite stato di uscita)

Si crei uno script **`uscita.sh`** con i comandi seguenti.

```
echo "Operazione fallita"
exit 1
```

Si esegua lo script da terminale:

```
bash uscita.sh
```

Si ottiene l'output seguente:

```
Operazione fallita
```

Un esempio concreto

(Segnalazione di un errore tramite stato di uscita)

Si stampi lo stato di uscita:

```
echo $?
```

Si ottiene l'output seguente:

```
1
```

Esercizio 13 (4 min.)

Scrivete uno script shell di nome **`ciclo.sh`** che svolge le operazioni seguenti.

Riceve in ingresso un parametro e lo memorizza nella variabile con attributo intero **`i`**. Se il parametro non è definito, associa il valore di default **`10`**.

Esegue un ciclo da **`1`** a **`i`**.

Ad ogni iterazione del ciclo, stampa il valore del numero.

Iterazione 1 → Stampa "1"
Iterazione 2 → Stampa "2"
…

Esce con lo stato **`0`**.

Debugging

(You know how to debug, right?)

BASH mette a disposizione diversi strumenti per il debugging di una applicazione.
I più immediati sono esposti nel seguito.

- Stampa degli statement eseguiti.
- Stampa dell'evoluzione dei costrutti di controllo di flusso.
- Stampa delle variabili e dei parametri.

Uno script sbagliato

(Per esercitare le attività di debugging)

Viene richiesta la creazione di uno script che stampa i numeri da 1 a 3 inclusi.

Viene prodotto lo script **`conta.sh`**, che contiene un banale errore.

```
declare -i a=1
while [ $a -lt 3 ]; do
   echo $a
   let a=++a
done
```

Stampa statement da eseguire

(Si usa l'opzione `-v` di BASH)

L'opzione `-v` di BASH abilita la stampa di ogni statement prima della sua esecuzione.

- Consente di capire se uno statement è eseguito.
- Consente di capire quando uno statement è eseguito.

Se usata contestualmente all'esecuzione di uno script, ha effetto per l'intera durata dello script.

L'opzione `-v` in azione

(Gli statement sembrano essere eseguiti tutti)

```
bash -v conta.sh
declare -i a=1
while [ $a -lt 3 ]; do
   echo $a
   let a=++a
done
1
2
```

L'opzione `-v` in azione

(Gli statement sembrano essere eseguiti tutti)

```
bash -v conta.sh
declare -i a=1
while [ $a -lt 3 ]; do
   echo $a
   let a=++a
done
1
2
```

La dichiarazione di variabile è stata eseguita.

L'opzione `-v` in azione

(Gli statement sembrano essere eseguiti tutti)

```
bash -v conta.sh
declare -i a=1
while [ $a -lt 3 ]; do
   echo $a
   let a=++a
done
1
2
```

Il ciclo while è stato eseguito.

L'opzione `-v` in azione

(Gli statement sembrano essere eseguiti tutti)

```
bash -v conta.sh
declare -i a=1
while [ $a -lt 3 ]; do
   echo $a
   let a=++a
done
1
2
```

È stata eseguita una iterazione in meno del previsto. Strano.

Esplicitazione statement da eseguire

(Si usa l'opzione `-x` di BASH)

L'opzione `-x` di BASH abilita l'esplicitazione di ogni statement prima della sua esecuzione.

- Consente di capire se variabili e parametri sono espansi correttamente.

Se usata contestualmente all'esecuzione di uno script, ha effetto per l'intera durata dello script.

Esplicitazione in dettaglio

(Una mano santa)

Viene stampata ogni iterazione di un qualunque costrutto di controllo del flusso.
Vengono stampati i valori di parametri e variabili.

Queste stampe sono precedute dal carattere **+**.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Tutte le righe precedute da `+` sono esplicitazioni dovute a `-x`.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Le altre righe sono gli output dello script.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Esplicitazione della assegnazione.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Esplicitazione della prima iterazione del ciclo while.

L'opzione **-x** in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Esplicitazione del parametro di **echo**.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Output di `echo 1`.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Non vengono esplicitate le espressioni aritmetiche (per non confonderle con le assegnazioni).

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Esplicitazione della seconda iterazione del ciclo while.

L'opzione **-x** in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Esplicitazione del parametro di **echo**.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Output di `echo 2`.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Non vengono esplicitate le espressioni aritmetiche (per non confonderle con le assegnazioni).

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Esplicitazione della terza iterazione del ciclo while.

L'opzione `-x` in azione

(C'è un errore nel test condizionale)

```
bash -x conta.sh
+ declare -i a=1
+ '[' 1 -lt 3 ']'
+ echo 1
1
+ let a=++a
+ '[' 2 -lt 3 ']'
+ echo 2
2
+ let a=++a
+ '[' 3 -lt 3 ']'
```

Non viene eseguita la terza iterazione del ciclo. L'operatore condizionale usato (`-lt`) è sbagliato. Bisogna usare l'operatore `-le`.

ALIAS E FUNZIONI

Definizione

(Insieme di comandi memorizzati in un file)

Ripetere una sequenza di statement ogni volta che la si necessita è una operazione tediosa, incline agli errori ed inefficiente.

Per tale motivo, BASH introduce strumenti per ridurre la complessità delle operazioni di inserimento e per favorire la programmazione modulare.

- Alias.
- Funzioni.

Alias

(È una abbreviazione statica di un comando)

Un **alias** è una abbreviazione di un comando.
Lo si può pensare come una macro che sostituisce un intero statement complesso (o una sua porzione iniziale, consistente in un comando, opzioni, parametri) con una sequenza di caratteri più breve.

ATTENZIONE! L'alias abbrevia solamente la parte iniziale di uno statement; non si espande a metà o alla fine.

Il comando **`alias`** è responsabile della gestione degli alias.

Stampa elenco alias

(Si usa **`alias`** senza opzioni)

Lanciando il comando **`alias`** senza opzioni, si ottiene l'elenco degli alias disponibili.
Per la distribuzione Debian GNU/Linux si ottiene l'output seguente.

```
alias ls='ls --color=auto'
```

Uso corretto dell'alias

(All'inizio dello statement, seguito da **<SPAZIO>** o **<INVIO>**)

Se si esegue uno statement che inizia con l'alias **`ls`**, esso viene prima sostituito con il comando seguente:

```
ls --color=auto
```

Esempi:

```
ls           → ls --color=auto
echo hi; ls  → echo hi; ls --color=auto
ls -l        → ls --color=auto -l
ls f         → ls --color=auto f
```

Uso sbagliato dell'alias

(Altrove)

Se l'alias viene inserito altrove, non si sortisce l'effetto sperato.

Esempi:

```
echo ls
echo ls a
lscpu
```

Creazione nuovo alias

(Si usa **`alias`** con un argomento specifico)

Per creare un nuovo alias, si esegue il comando **`alias`** nel modo seguente:

```
alias NOME_ALIAS='COMANDO'
```

Ad esempio, si può attivare una rappresentazione più compatta di file e directory con l'alias seguente.

```
alias l='ls -hog'
```

Cancellazione alias

(Si usa **`unalias`** con il nome dell'alias)

Per cancellare un alias esistente, si esegue il comando **`unalias`** nel modo seguente:

```
unalias NOME_ALIAS
```

Ad esempio, per cancellare l'alias appena creato:

```
unalias l
```

Esercizio 14 (2 min.)

Girovagando sul Web avete trovato il comando seguente, che vi piace moltissimo e decidete di usare:

```
strings /dev/urandom | grep -o '[[:alnum:]]' | head -n 30 | tr -d '\n'; echo
```

Create un alias di nome **gp** per questo comando.

Eseguite l'alias **gp**. Che cosa ottenete?

Funzione

(Permette di praticare la divina arte della ricorsione)

BASH mette a disposizione un costrutto **funzione**.
Funzione: è una sequenza di statement con le proprietà seguenti.

- È raggruppata all'interno di un blocco di codice.
- È dotata di un nome che consente di invocarla.
- Può fare uso di parametri posizionali.
- Può ritornare un valore.

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

La parola chiave **`function`** introduce il costrutto. Può essere omessa.

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

Questo è il nome della funzione. Valgono le stesse regole di denominazione delle variabili.

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

La coppia di parentesi è ornamentale. I parametri formali NON vengono scritti lì dentro.

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

Le parentesi graffe delimitano il blocco di codice contenente il corpo della funzione.

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

Il corpo è fatto di una sequenza di statement **`STATEMENTS`**.

Costrutto funzione

(Equivalente agli altri linguaggi)

Il costrutto funzione ha la seguente sintassi generale.

```
function FUNC_NAME() {
   STATEMENTS
}
```

Il corpo di una funzione non può essere vuoto.
Se proprio lo si vuole vuoto, si può usare lo statement nullo, ovvero i due punti `:`.

Invocazione di una funzione

(Rozza, ma efficace)

Una funzione si invoca nel modo seguente.

```
FUNC_NAME arg1 arg2 arg3 … argN
```

dove **`arg1`**, **`arg2`**, … **`argN`** sono stringhe associate ai parametri posizionali.

Ad esempio:

```
my_function 1 2 3 4
```

Uso dei parametri posizionali

(Sempre tramite le variabili speciali `$1`, `$2`, …)

All'interno di una funzione, gli argomenti sono accessibili tramite le variabili speciali **`$1`**, **`$2`**, … (esattamente come per i parametri di uno script).

ATTENZIONE! Dentro una funzione, **`$0`** continua ad avere il nome dello script che la contiene (e non il nome della funzione, come si potrebbe essere portati a credere).

Ritorno di un valore

(Si usa lo statement **`return`**)

BASH non ha il concetto di "ritorno di un valore ad una funzione invocante".
BASH offre lo statement **`return`** che permette di impostare la variabile speciale **`$?`** con un valore che rappresenta lo stato di uscita della funzione.

Ad esempio:

```
return 1
return "a"
```

Uso del valore di ritorno

(Si usa **`$?`**, magari assegnandola ad un'altra variabile)

Subito dopo l'invocazione della funzione, il valore di ritorno è disponibile in **`$?`** per essere consumato e/o assegnato ad una variabile.

Ad esempio:

```
my_function 1 2 3 4
var=$?
```

Esercizio 15 (2 min.)

Scrivete una funzione **`square()`** che accetta un parametro, lo interpreta come un intero e ne ritorna il quadrato.
Usate **`square()`** per stampare il quadrato di **`4`**.

Definizione

(Campo di visibilità)

Campo di visibilità (scope). È la porzione di uno script in cui è valida l'associazione tra un nome simbolico e la corrispondente cella di memoria.

Variabile globale. È una variabile il cui campo di visibilità è l'intero programma.
Variabile locale. È una variabile il cui campo di visibilità non ricopre l'intero programma, ma solo una sua parte.

Determinazione del campo di visibilità

(Statica o dinamica)

Il campo di visibilità di una variabile contenuta in un blocco di codice B è determinato in due modi diversi.

Scope statico (static scoping). Il campo di visibilità di una variabile locale dipende esclusivamente dal codice sorgente del programma.

Si coinvolgono blocchi di codice di livello superiore a B.

Scope dinamico (dynamic scoping). Il campo di visibilità di una variabile locale è calcolato a tempo di esecuzione (run time).

Si coinvolgono i blocchi di codice invocanti a cascata B.

Un esempio generale

(Programma in pseudocodice, con una variabile **x** definita più volte)

```
MAIN
dichiarazione di x
    SUB1
    dichiarazione di x
        ...
        call SUB2
        ...

    SUB2
        ...
        riferimento a x
        ...
... (corpo di MAIN)
```

In questo programma, la variabile **x** è definita in diverse posizioni:

- nella funzione **MAIN**;
- nella funzione **SUB1**.

Un esempio generale

(Programma in pseudocodice, con una variabile **x** riferita una volte)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```

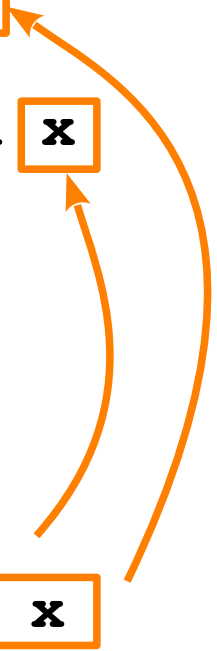
In questo programma, la variabile **x** è riferita in una posizione, ovvero la funzione **SUB2**.

Un esempio generale

(Bella domanda!)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```



Domanda: all'interno di **SUB2**, a quale definizione di **x** punta lo statement evidenziato?

Quella in **MAIN**?

Quella in **SUB1**?

Un esempio generale

(Scope statico – **x** punta alla definizione nel blocco superiore più vicino)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```

Nello scope statico, il campo di visibilità è dedotto esclusivamente dal codice sorgente (“su carta”).

Un esempio generale

(Scope statico – **x** punta alla definizione nel blocco superiore più vicino)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```

Tipicamente, si cerca la definizione di **x** nei blocchi di livello via via superiore.
Qui, il primo (e unico) blocco di livello superiore di **SUB2** contenente **x** è **MAIN**.

Un esempio generale

(Scope statico – **x** punta alla definizione nel blocco superiore più vicino)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```

Pertanto, **x** punta alla definizione di **MAIN**.

Un esempio generale

(Scope dinamico – **x** punta al blocco più vicino in una cascata di invocazioni)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```

Nello scope dinamico, il campo di visibilità è calcolato a tempo di esecuzione.

Un esempio generale

(Scope dinamico – **x** punta al blocco più vicino in una cascata di invocazioni)

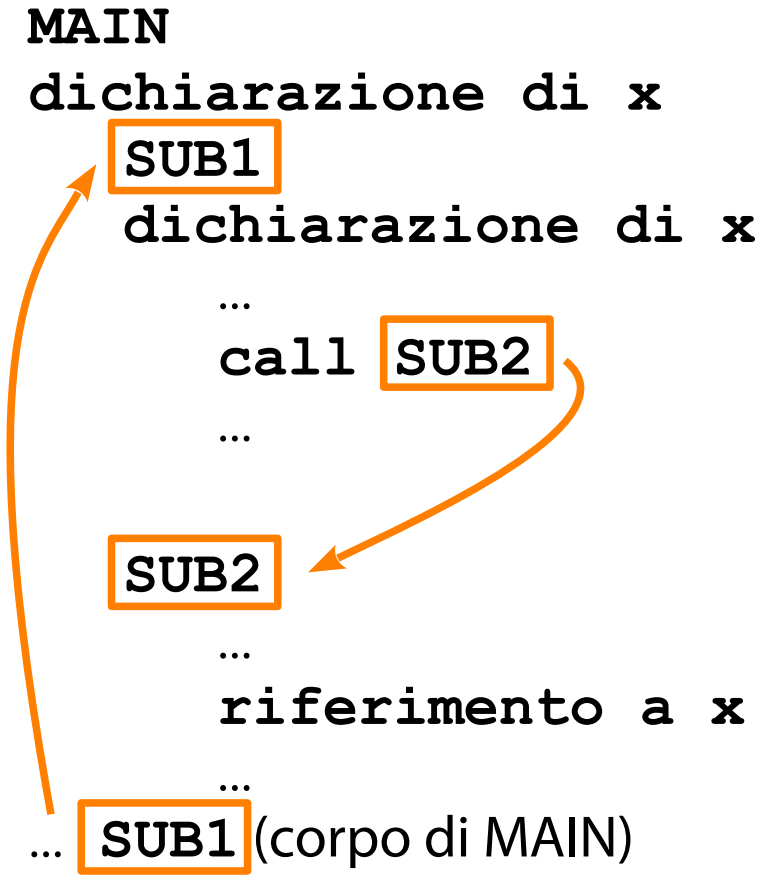
```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… (corpo di MAIN)
```

Tipicamente, si cerca la definizione di **x** nei blocchi coinvolti in una cascata di invocazioni.
Si usa la definizione nel blocco "più vicino" a quello contenente il riferimento.

Un esempio generale

(Scope dinamico – **x** punta al blocco più vicino in una cascata di invocazioni)



Ad esempio, si consideri la sequenza di chiamate:

MAIN → SUB1 → SUB2

Si procede a ritroso cercando **x** prima in **SUB1** e poi in **MAIN**.

Un esempio generale

(Scope dinamico – **x** punta al blocco più vicino in una cascata di invocazioni)

```
MAIN
dichiarazione di x
    SUB1
    dichiarazione di x
        …
        call SUB2
        …

    SUB2
        …
        riferimento a x
        …
… SUB1 (corpo di MAIN)
```

Ad esempio, si consideri la sequenza di chiamate:

MAIN → SUB1 → SUB2

Il blocco più vicino a **SUB2** e contenente una definizione di **x** è **SUB1**.

Un esempio generale

(Scope dinamico – **x** punta al blocco più vicino in una cascata di invocazioni)

```
MAIN
dichiarazione di x
   SUB1
   dichiarazione di x
      …
      call SUB2
      …

   SUB2
      …
      riferimento a x
      …
… SUB1 (corpo di MAIN)
```

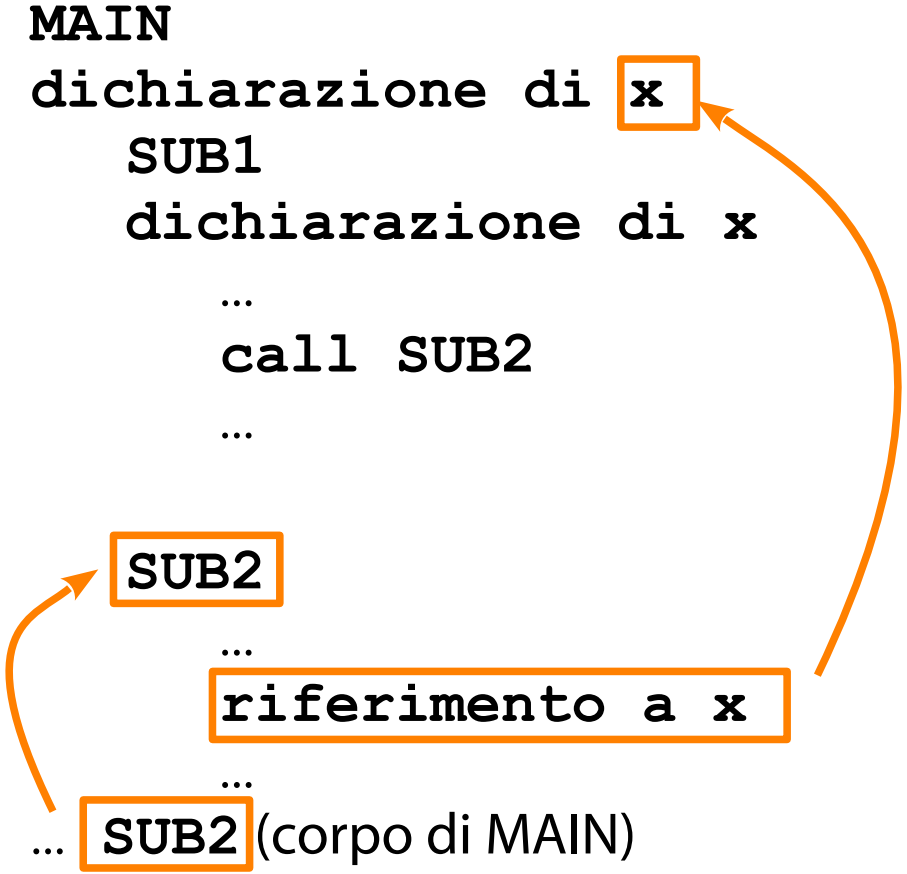
Ad esempio, si consideri la sequenza di chiamate:

MAIN → SUB1 → SUB2

Pertanto, per questa cascata di invocazioni **x** punta alla definizione di **SUB1**.

Un esempio generale

(Scope dinamico – **x** punta al blocco più vicino in una cascata di invocazioni)



Si consideri invece la sequenza di chiamate:

MAIN → SUB2

Si procede a ritroso cercando **x** in **MAIN**, che contiene una definizione di **x**.

Pertanto, per questa cascata di invocazioni **x** punta alla definizione di **MAIN**.

Campo visibilità variabili in BASH

(È globale, di default)

In BASH, il campo di visibilità di default è **globale**.

Anche per le variabili definite all'interno di una funzione!

Di conseguenza, una variabile è visibile in tutto lo script in cui è presente.

L'importante è che, al momento del suo uso, lo statement che definisce la variabile sia già stato interpretato.

Un esempio concreto

(Variabili globali)

```
my_function() {
   a=1
}
my_function
echo $a
```

Un esempio concreto

(Variabili globali)

```
my_function() {
   a=1
}
my_function
echo $a
```

Lo statement evidenziato stampa il valore della variabile **a**.

Un esempio concreto

(Variabili globali)

```
my_function() {
   a=1
}
my_function
echo $a
```

La variabile **a** è definita in **my_function**.

Un esempio concreto

(Variabili globali)

```
my_function() {
   a=1
}
my_function
echo $a
```

Le variabili sono globali di default.
È sufficiente eseguire **`my_function`**, e l'assegnazione **`a=1`** è visibile in tutto lo script.

Un esempio concreto

(Variabili globali)

```
my_function() {
   a=1
}
my_function
echo $a
```

Quando viene eseguita l'intera sequenza di comandi, lo statement **`my_function`** imposta il valore di **`a`**.

Un esempio concreto

(Variabili globali)

```
my_function() {
   a=1
}
my_function
echo $a
```

Qui viene stampata **a**. Essa è già stata definita con uno scope globale in **my_function**. Pertanto, si ha **a=1**.

Creazione di variabili locali in BASH

(Si usa la parola chiave **`local`**)

In BASH, è possibile forzare una variabile ad avere un campo di visibilità **locale** alla funzione che la definisce.
A tal scopo, si prepende la parola chiave **`local`** allo statement di definizione e/o assegnazione della variabile.

```
local a=1
```

La definizione locale è possibile solo all'interno di una funzione.

Un esempio concreto

(Variabili locali)

```
my_function() {
   local a=1
}
my_function
echo $a
```

Un esempio concreto

(Variabili locali)

```
my_function() {
   local a=1
}
my_function
echo $a
```

Lo statement evidenziato stampa il valore della variabile **a**.

Un esempio concreto

(Variabili locali)

```
my_function() {
   local a=1
}
my_function
echo $a
```

La variabile **a** è definita in **my_function**.

Un esempio concreto

(Variabili locali)

```
my_function() {
   local a=1
}
my_function
echo $a
```

La definizione è locale. Pertanto, è visibile solo in **`my_function`**.

Un esempio concreto

(Variabili locali)

```
my_function() {
   local a=1
}
my_function
echo $a
```

Quando viene eseguita l'intera sequenza di comandi, lo statement **`my_function`** imposta il valore di **`a`**.

L'assegnazione **`a=1`** vale solo per l'esecuzione di **`my_function`**.

Un esempio concreto

(Variabili locali)

```
my_function() {
   local a=1
}
my_function
echo $a
```

Qui viene stampata **a**. Essa è già stata definita con uno scope locale in **my_function**. Pertanto, risulta non ancora definita. L'output è nullo.

Calcolo campo visibilità in BASH

(Si usa lo scope dinamico)

In BASH, il campo di visibilità delle variabili è determinato in modo **dinamico**.

Di conseguenza, per risolvere l'espressione di una variabile è necessario studiare di volta in volta la cascata di funzioni che porta alla sua interpretazione.

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```

La variabile **x** è definita in due punti distinti:
localmente, in **func2**;
globalmente, al di fuori delle funzioni;

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```

Domanda: all'interno di **func1**, a quale definizione di **x** punta lo statement evidenziato?

- Quella locale?
- Quella globale?

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```

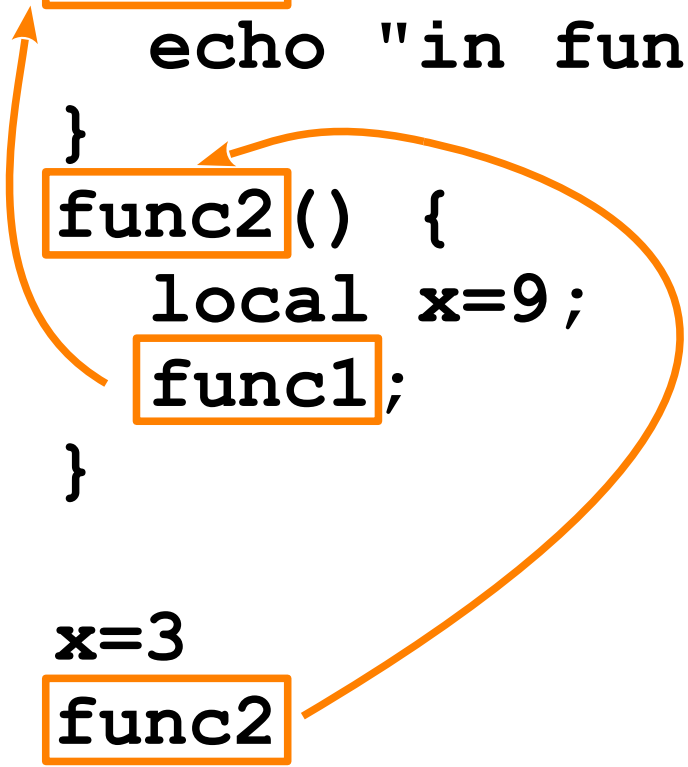
Lo scope è dinamico; pertanto, il campo di visibilità di una variabile dipende dalla cascata di funzioni che porta all'esecuzione dello statement evidenziato che la riferisce.

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```



func2 attiva una invocazione a **func1**.

SCRIPT → func2 → func1

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```

Si esaminano a ritroso gli invocatori di **`func1`**.
Il più vicino è **`func2`**.

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```

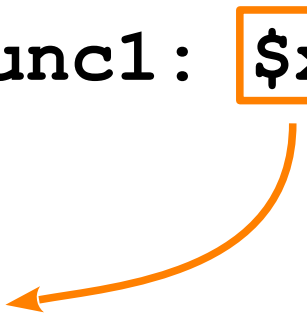
In **func2** è definita una variabile **x** con scope locale.
→ L'assegnazione **x=9** è visibile solo durante l'esecuzione di **func2** (che include la chiamata a **func1**).

Un esempio concreto

(Scope dinamico)

```
func1() {
   echo "in func1: $x";
}
func2() {
   local x=9;
   func1;
}

x=3
func2
```



Lo scope delle variabili è dinamico.
→ **func1** usa la definizione di **x** in **func2**.
Pertanto, l'output è **9**.

Esercizio 16 (3 min.)

Provate ad indovinare l'output dello script seguente di nome **`scope.sh`**, senza eseguirlo.

```
function g() {
   echo $x; x=2;
}
function f() {
   local x=3; g;
}

x=1
f
echo $x
```

STORIA DEI COMANDI

Definizione

(Al posto dell'alias)

Ripetere statement complessi al terminale ogni volta che li si necessita è una operazione tediosa, incline agli errori ed inefficiente.
Per tale motivo, BASH introduce strumenti per la gestione della **storia dei comandi**.

Storia dei comandi (command history). È l'insieme dei comandi immessi all'interprete in tutte le sessioni interattive al terminale.

Architettura ad alto livello

(Buffer di righe sincronizzato al termine di una sessione di lavoro)

La storia dei comandi (un comando per riga) è concepita come un buffer di memoria contenente i comandi (un buffer per riga), gestito da BASH.

Per motivi di efficienza, BASH appende la porzione più recente del buffer ad un file di log (avente nome **`.bash_history`** di default) solo al termine della sessione di lavoro.

Configurazione della storia

(Si usano le variabili di ambiente **`HIST*`**)

La storia dei comandi può essere configurata tramite variabili di ambiente.

Quelle che iniziano con la stringa **`HIST`**.

`HISTFILE`: percorso del file.

`HISTFILESIZE`: numero massimo di righe in **`HISTFILE`**.

`HISTSIZE`: numero massimo di comandi salvabili in **`HISTFILE`** al termine di una sessione di lavoro.

`HISTIGNORE`: una lista di pattern (separati dal carattere **`:`**) da ignorare.

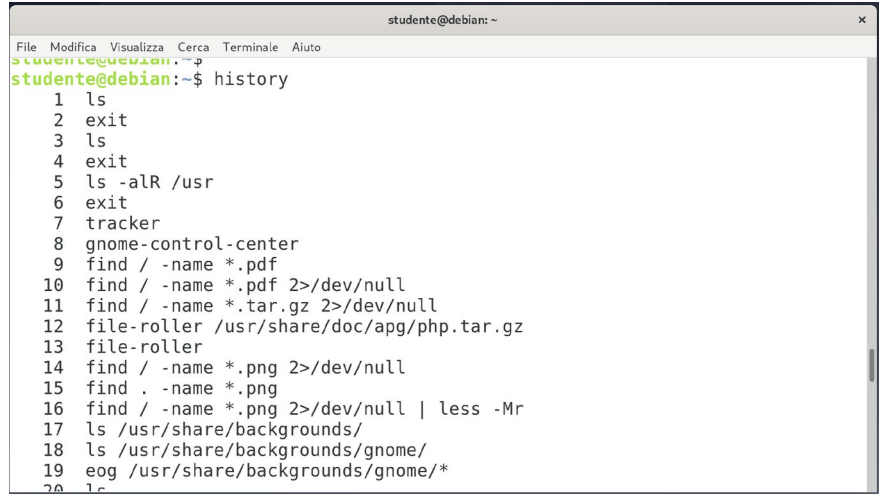
…

Visione buffer storia dei comandi

(Si usa il comando **`history`**)

Il comando **`history`** permette di gestire buffer e log dei comandi.
Lanciato senza argomenti, **`history`** elenca tutti i comandi presenti nella storia (uno per riga).

`history`



Visione buffer storia dei comandi

(Si usa il comando **`history`**)

Lanciato con un argomento **`N`** (numero intero positivo), **`history`** elenca gli ultimi **`N`** comandi nel buffer.

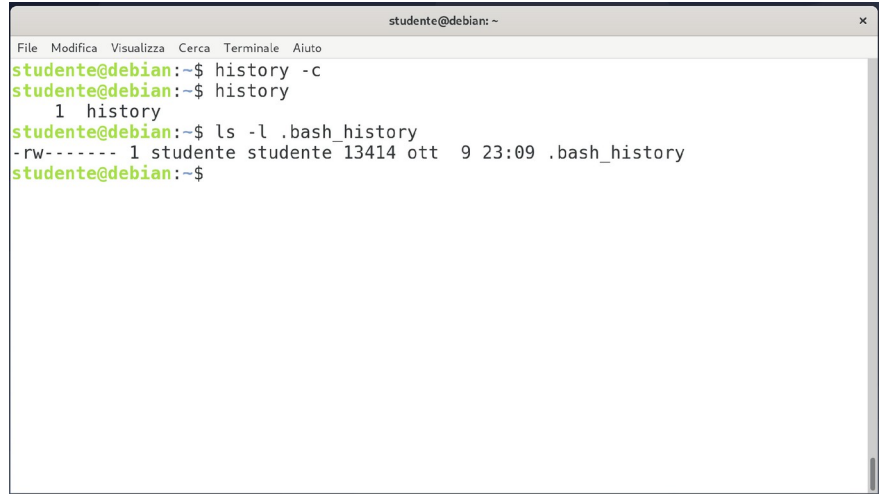
`history N`

Cancellazione storia dei comandi

(Si usa l'opzione **`-c`** del comando **`history`**)

Lanciato con l'opzione **`-c`**, **`history`** cancella il buffer (ma non il file di log).

```
history -c
```

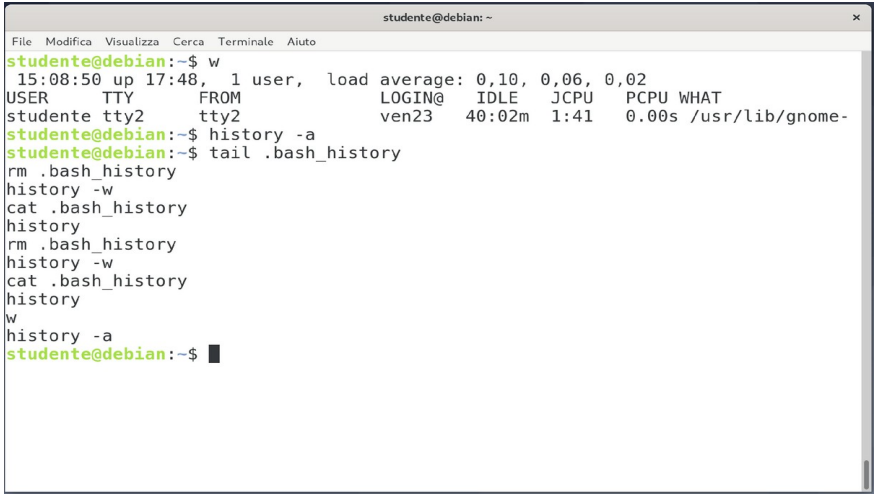


Salvataggio storia dei comandi

(Si usano le opzioni **`-w`** e **`-a`** del comando **`history`**)

È possibile forzare la sincronizzazione del buffer sul log prima del termine di una sessione.

- **`history -w`**: il buffer della storia è salvato su file.
- **`history -a`**: il buffer della storia è appeso al file.

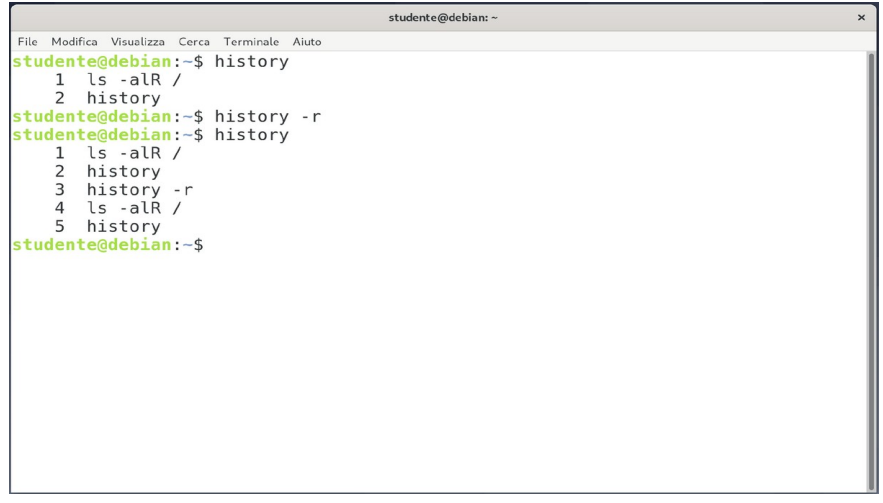


Lettura storia dei comandi

(Si usa l'opzione **`-r`** del comando **`history`**)

È possibile forzare la rilettura del log dei comandi nel buffer dopo l'inizio di una sessione.

`history -r`: i comandi nel log sono appesi alla fine del buffer.



Esercizio 17 (1 min.)

Individuate il comando più recente presente nella vostra storia dei comandi.

Manipolazione cronologia dei comandi

(Si usa il comando **`fc`**)

Il comando **`fc`** permette di gestire la **cronologia** dei comandi, ovvero gli ultimi **`HISTSIZE`** comandi del buffer.

PS: **`fc`** sta per **fix command**.

Per chi se lo stesse chiedendo…

Edit ed esecuzione ultimo comando

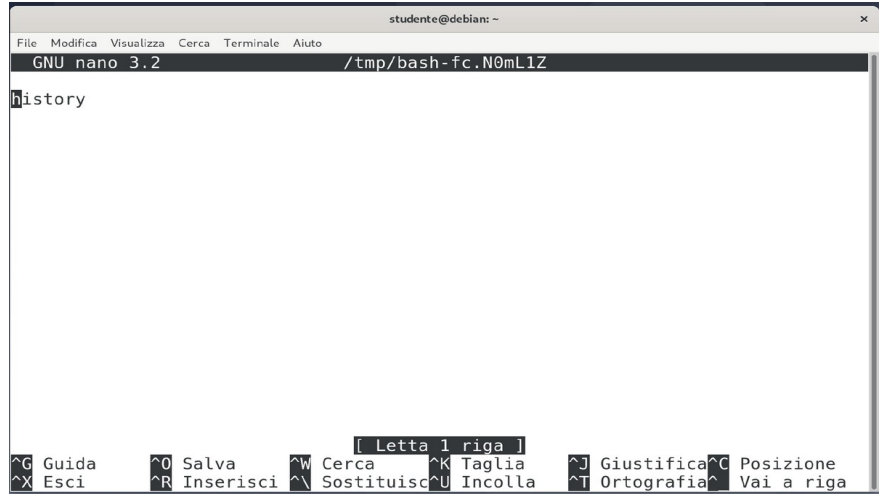
(Si usa il comando **`fc`** senza argomenti)

Si provi a lanciare **`fc`** senza argomenti.

`fc`

Viene avviato l'editor di default di Debian (**GNU nano**).

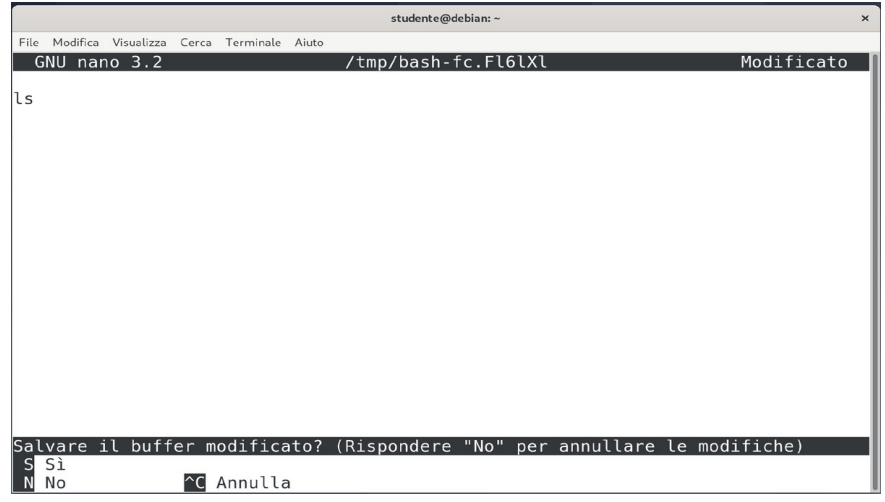
Il buffer dell'editor contiene l'ultimo comando ed è associato ad un file temporaneo.



Edit ed esecuzione ultimo comando

(Si usa il comando **`fc`** senza argomenti)

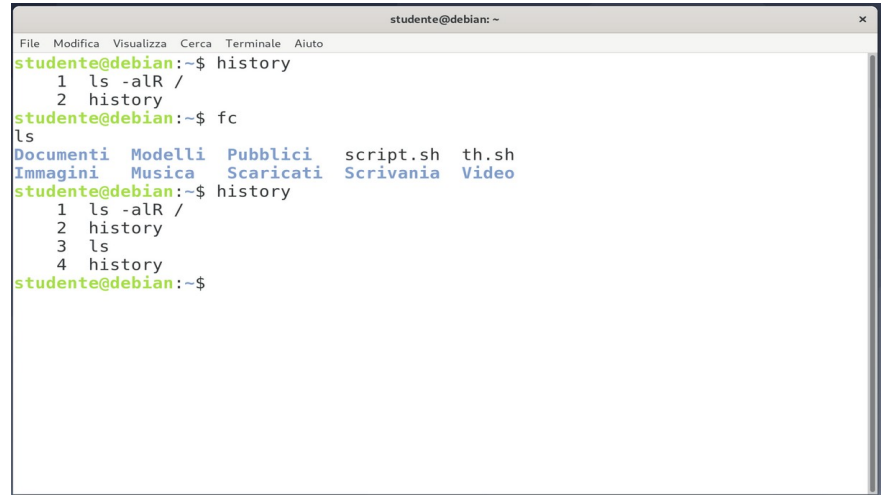
È possibile editare il comando. Lo si faccia!
Si cambi **`history`** in **`ls`**.
Si salvi il buffer premendo la combinazione **`<CTRL>-x`**.



Edit ed esecuzione ultimo comando

(Si usa il comando **`fc`** senza argomenti)

Il comando **`fc`** ha eseguito il comando salvato nel file temporaneo (**`ls`**).



ATTENZIONE! Per non far eseguire il comando si può:

salvare un file vuoto;

OPPURE

far uscire l'editor con un codice di errore (ad esempio, in VIM **`:cq!`**).

Edit ed esecuzione ultimo comando

(Si usa il comando **`fc`** senza argomenti)

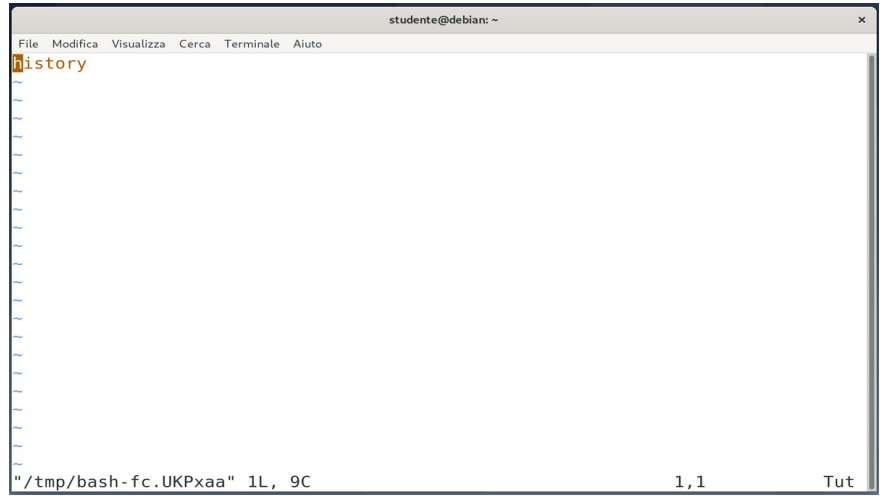
È possibile usare un altro editor (senza modificare l'editor di default) in diversi modi.

Si usa l'opzione **`-e`** di **`fc`**.

```
fc -e vim
```

Si imposta nella variabile di ambiente **`EDITOR`** il nome del programma editor.

```
EDITOR=vim fc
```



Edit ed esecuzione N-mo comando

(Si usa il comando **`fc`** con argomento l'indice del comando)

È possibile editare ed eseguire il comando n-mo presente nel buffer di storia dei comandi.

Si passa a **`fc`** un argomento, ovvero l'indice **`N`** del comando (presente nell'elenco della storia dei comandi).

`fc N`

Edit ed esecuzione N-ultimo comando

(Si usa il comando **`fc`** con argomento l'indice del comando)

È possibile editare ed eseguire il comando n-mo presente nel buffer di storia dei comandi.
Passando un argomento intero negativo **`-N`**, è possibile selezionare l'N-ultimo comando.

`fc -N`

È un numero negativo (non una opzione).

Edit ed esecuzione comando con match

(Si usa il comando **`fc`** con argomento una sottostringa)

È possibile editare ed eseguire un comando presente nel buffer di storia dei comandi, tramite match.

Si passa a **`fc`** un argomento, ovvero una sottostringa **`S`** di un comando. Viene ritornato l'ultimo comando che contiene S.

```
fc S
```

Edit ed esecuzione gruppo comandi

(Si usa il comando **`fc`** con due argomenti indicanti primo e ultimo comando)

È possibile editare ed eseguire un gruppo di comandi, identificabile nei modi ora visti.

Due indici **`N1`** e **`N2`**.

`fc N1 N2`

ATTENZIONE! I due indici devono riferirsi agli ultimi **`HISTSIZE`** comandi del buffer.

Edit ed esecuzione gruppo comandi

(Si usa il comando **`fc`** con due argomenti indicanti primo e ultimo comando)

È possibile editare ed eseguire un gruppo di comandi, identificabile nei modi ora visti.

Due sottostringhe **`S1`** e **`S2`**.

`fc S1 S2`

ATTENZIONE! Le due stringhe devono riferirsi agli ultimi **`HISTSIZE`** comandi del buffer.

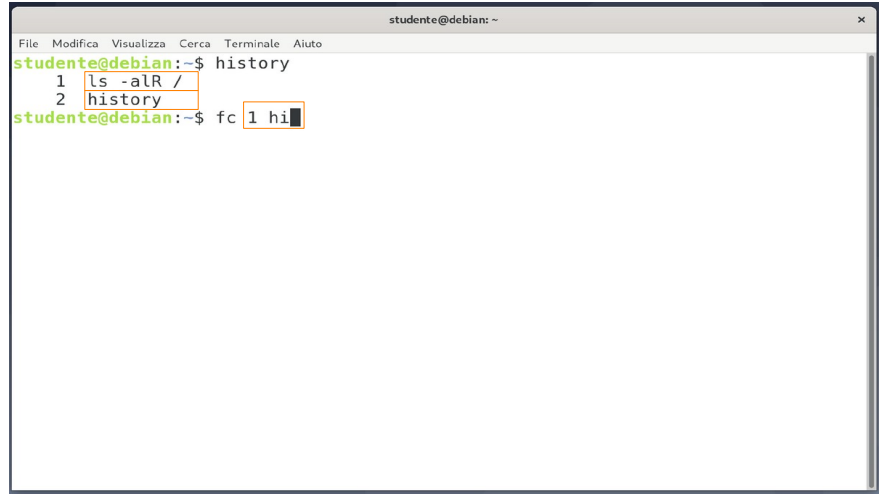
Edit ed esecuzione gruppo comandi

(Si usa il comando **`fc`** con due argomenti indicanti primo e ultimo comando)

È possibile mischiare numeri e stringhe.

```
fc N1 S2
fc S1 N2
```

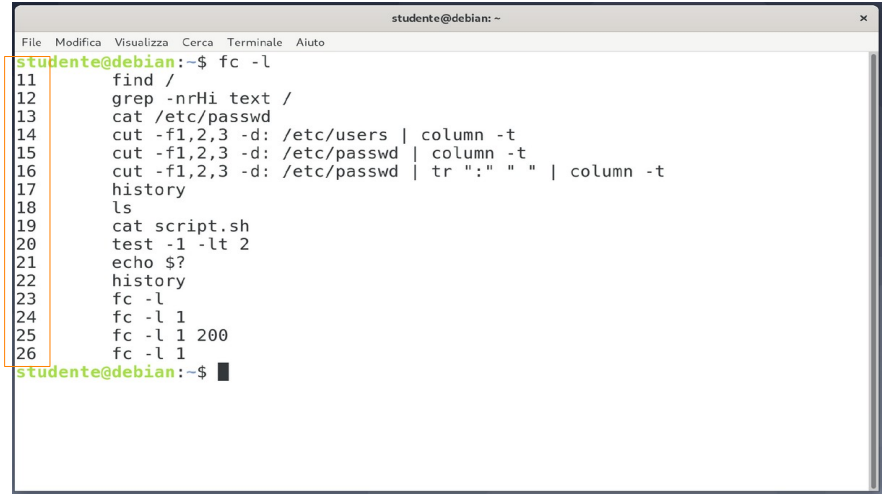
ATTENZIONE! Numeri e stringhe devono riferirsi agli ultimi **`HISTSIZE`** comandi del buffer.



Visione buffer storia dei comandi

(Si usa l'opzione **`-l`** del comando **`fc`**)

È possibile elencare i comandi nella cronologia.
Si usa l'opzione **`-l`** di **`fc`**. Se non si usano argomenti, vengono stampati gli ultimi 16 comandi.



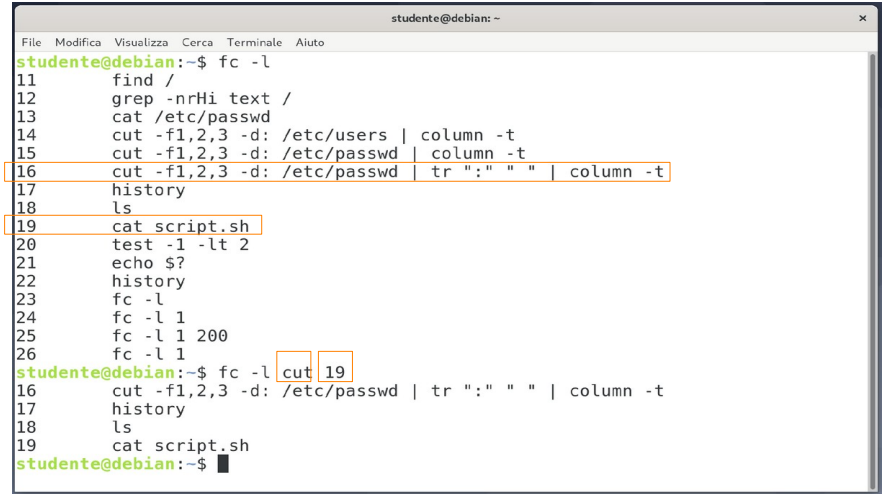
Visione buffer storia dei comandi

(Si usa l'opzione **`-l`** del comando **`fc`**)

L'opzione **`-l`** accetta tutti gli argomenti visti finora.

- Uno/due numeri interi (positivi o negativi).
- Una/due stringhe.

ATTENZIONE! Numeri e stringhe devono riferirsi agli ultimi **`HISTSIZE`** comandi del buffer.



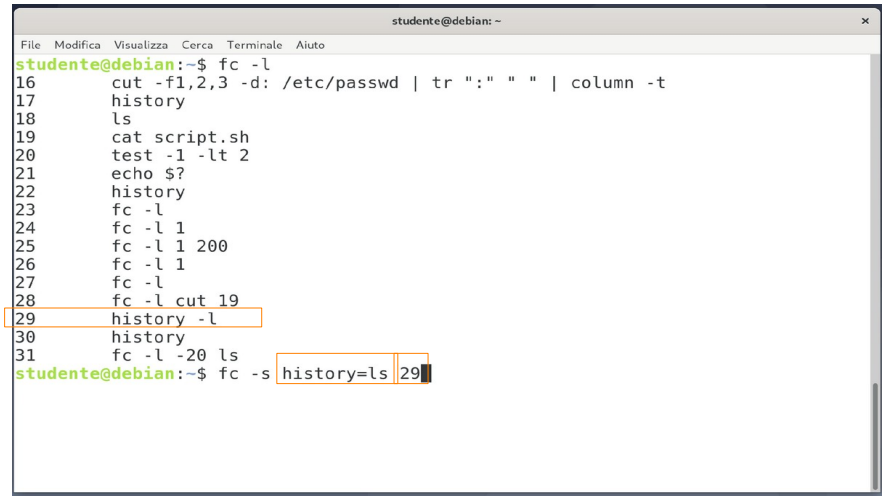
Esecuzione comandi a partire da altri

(Si usa l'opzione **`-s`** del comando **`fc`**)

L'opzione **`-s`** accetta due argomenti non obbligatori.

Una stringa della forma **`pat=rep`**.

Un indice/stringa che si riferisce ad un comando.



`fc [pat=rep] [S|N]`

Gli argomenti sono espressi in forma di Backus-Naur.

Esecuzione comandi a partire da altri

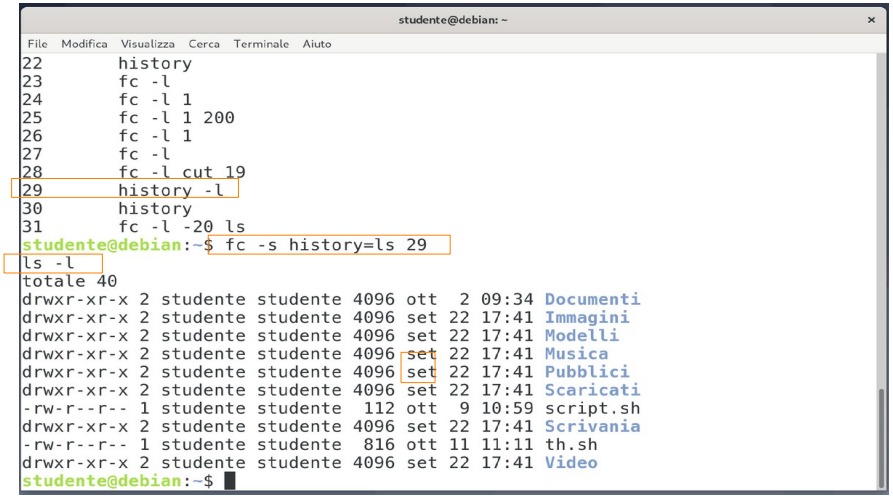
(Si usa l'opzione **`-s`** del comando **`fc`**)

Ad esempio, data la storia dei comandi nell'immagine a destra, il comando seguente:

`fc -s history=ls 29`

svolge le operazioni seguenti:

- individua il comando di indice 29 (**`history -l`**);
- sostituisce **`history`** con **`ls`**;
- esegue il comando risultante (**`ls -l`**).



Esercizio 18 (1 min.)

Elencate tutti i comandi a partire dall'ultimo **`ls`**.

Operatori di espansione della storia

(Permettono di identificare ed eseguire comandi, anche modificandoli)

BASH mette a disposizione anche degli **operatori di espansione** della storia dei comandi.

Questi operatori, piazzabili ovunque all'interno di un comando si espandono in comandi presenti nella storia dei comandi.

Se immessi da soli, le funzionalità offerte sono identiche a quelle di **`history`** e **`fc`**.

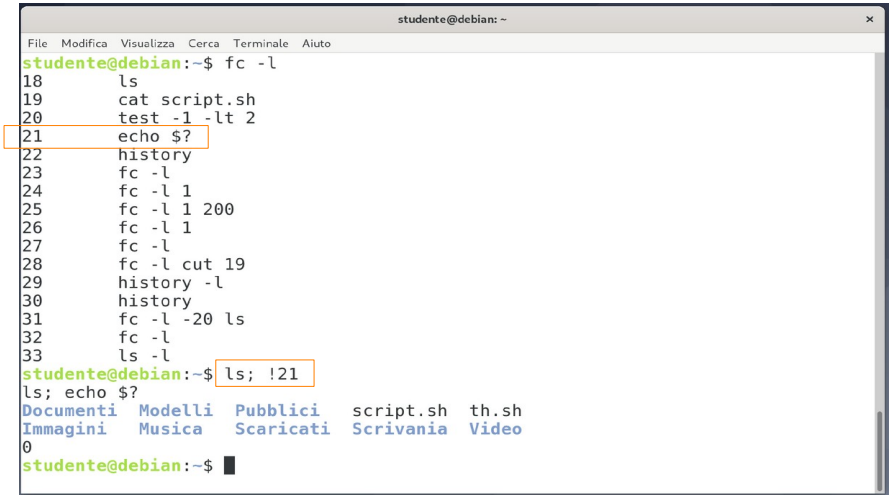
A differenza di **`history`** e **`fc`**, gli operatori di espansione si possono integrare in una porzione di comando preesistente.

Espansione nel comando N-mo

(Si usa l'operatore **!N**, dove **N** è un numero intero)

L'operatore **!N** (numero intero) si espande nell'N-mo comando presente nel buffer della storia dei comandi.

!N

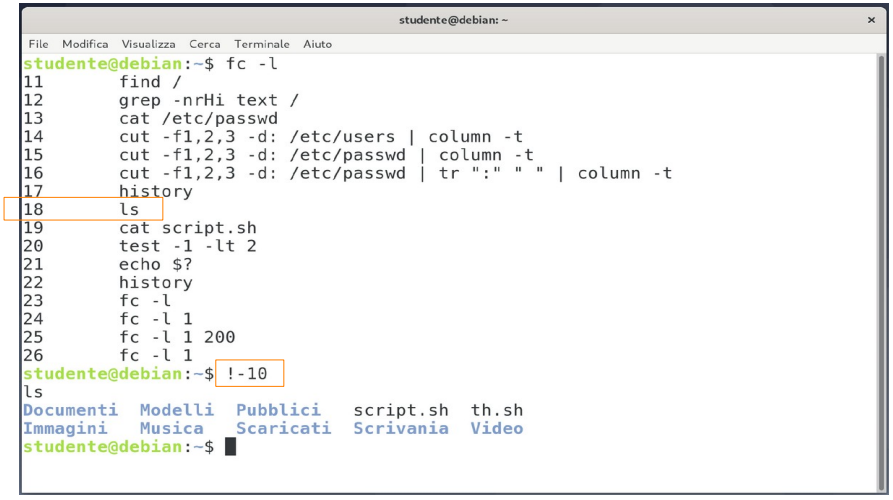


Espansione nel comando N-ultimo

(Si usa l'operatore **!-N**, dove **N** è un numero intero)

L'operatore **!-N** (numero intero negativo) si espande nell'N-ultimo comando presente nel buffer della storia dei comandi.

!-N



ATTENZIONE! Il conteggio all'indietro parte dall'indice del comando attuale (27).

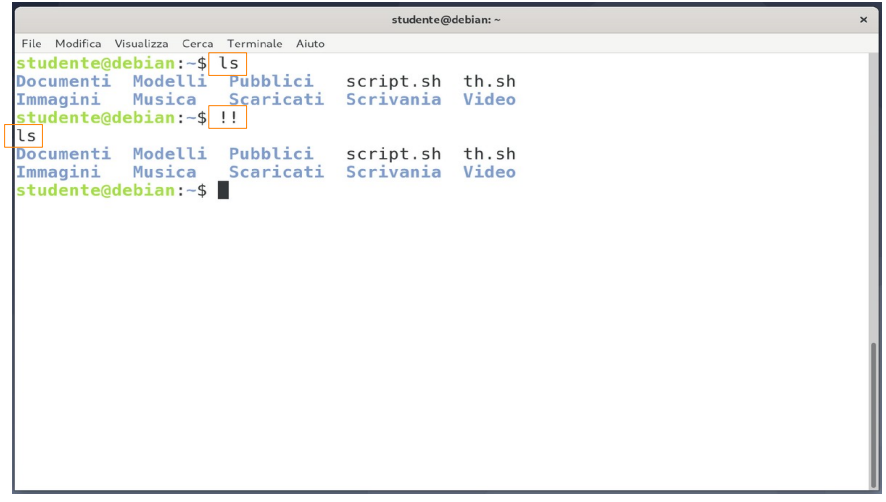
27-26-25-…(10 volte) → 18

Espansione dell'ultimo comando

(Si usa l'operatore `!!`)

L'operatore `!!` si espande nell'ultimo comando immesso. È un alias di `!-1`.

```
!!
```

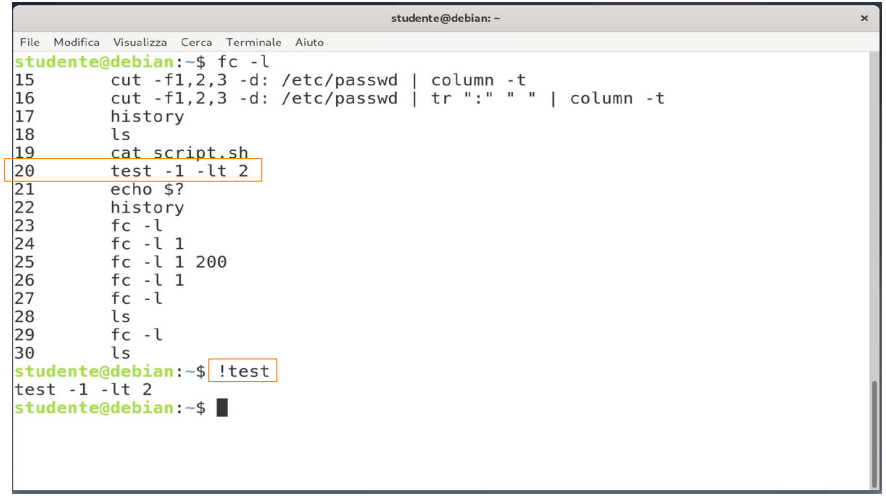


Espansione ultimo comando con match

(Si usa l'operatore **!s**, dove **s** è una stringa)

L'operatore **!s** (dove **s** è una stringa) si espande nell'ultimo comando immesso che contiene la stringa **s**.

!s



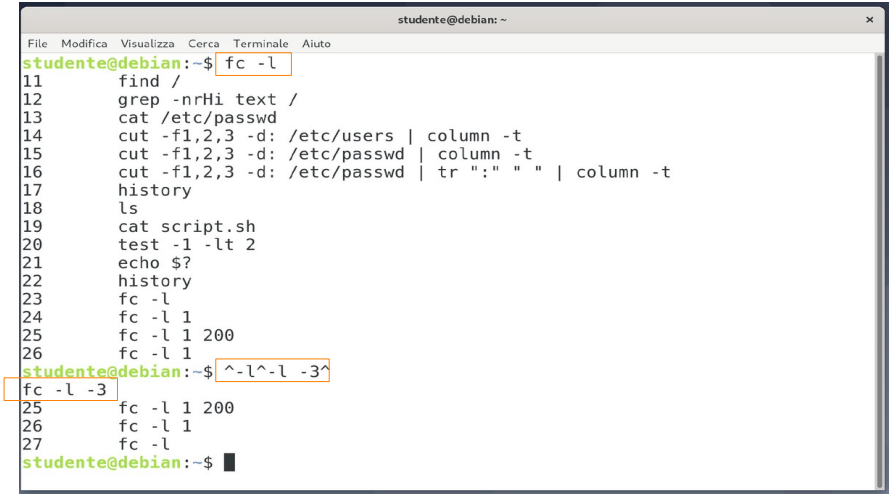
Espansione con sostituzione

(Si usa l'operatore ^ con una o due stringhe)

L'operatore **^** introduce la sostituzione di stringhe nei comandi precedenti.

Se usato con due stringhe **S1** e **S2**, sostituisce **S1** con **S2** nell'ultimo comando.

^S1^S2^

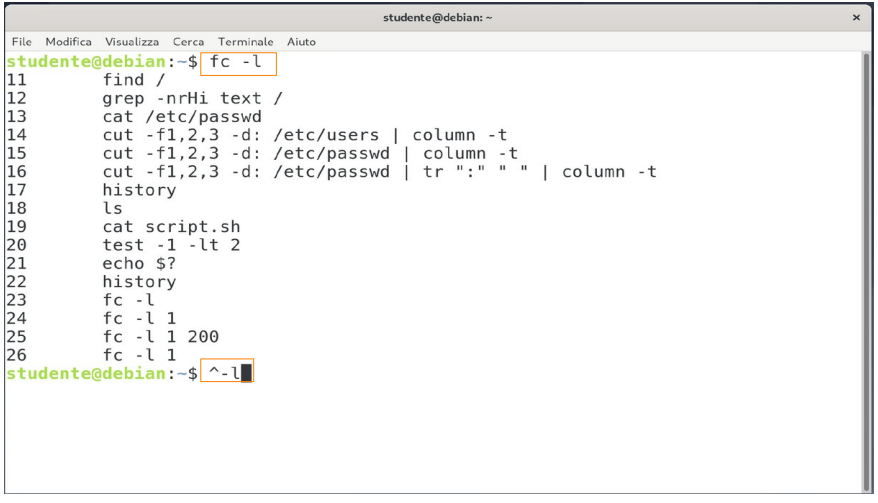


Espansione con sostituzione

(Si usa l'operatore ^ con una o due stringhe)

L'operatore **^** introduce la sostituzione di stringhe nei comandi precedenti.
Se usato con una sola stringa **S1**, cancella **S1** nell'ultimo comando.

^S1^



Espansione con sostituzione

(Si usa l'operatore ^ con una o due stringhe)

L'operatore **^** introduce la sostituzione di stringhe nei comandi precedenti.
Se usato con una sola stringa **S1**, cancella **S1** nell'ultimo comando.

^S1^

Nell'esempio in questione, dopo aver rimosso **-l** da **fc -l** si esegue **fc**.

Espansione con sostituzione

(La forma generica, un pelino più complicata)

La forma più generale di espansione con sostituzione è più complicata, ma molto versatile.
Essa ha la forma seguente:

`E:s/S1/S2`

dove:

`E` è un operatore di espansione della storia dei comandi;

`S1` è la stringa da sostituire;

`S2` è la stringa sostituta.

Espansione con sostituzione

(La forma generica, un pelino più complicata)

Ad esempio, si voglia sostituire il comando:

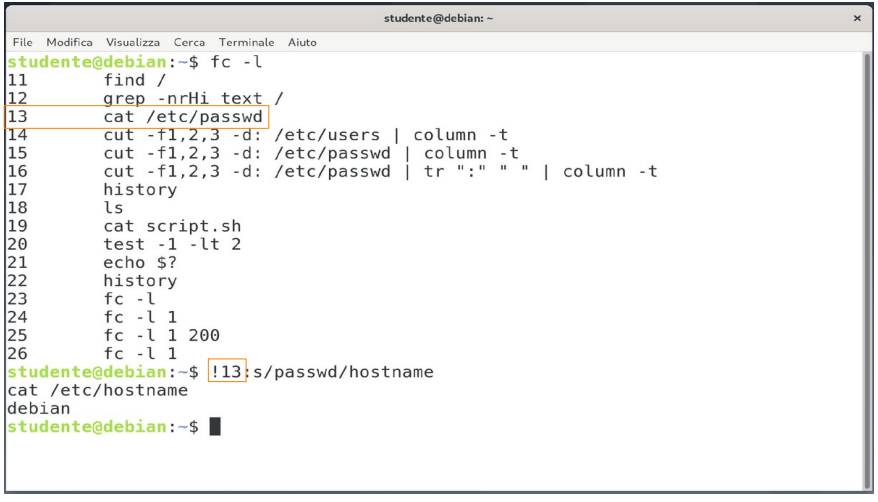
`cat /etc/passwd`

con il comando:

`cat /etc/hostname`

Si identifica il comando **`/etc/passwd`** con una espansione.

`!13`



Espansione con sostituzione

(La forma generica, un pelino più complicata)

Ad esempio, si voglia sostituire il comando:

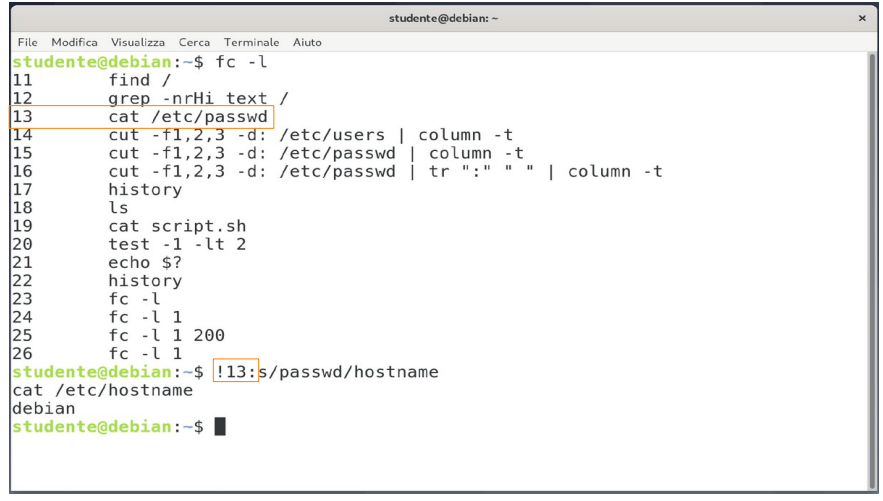
```
cat /etc/passwd
```

con il comando:

```
cat /etc/hostname
```

Si aggiunge il carattere `:`.

```
!13:
```



Espansione con sostituzione

(La forma generica, un pelino più complicata)

Ad esempio, si voglia sostituire il comando:

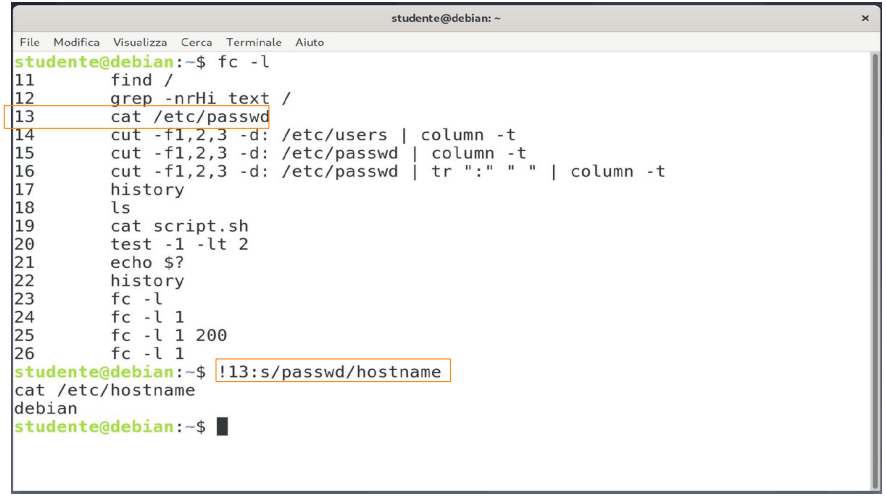
`cat /etc/passwd`

con il comando:

`cat /etc/hostname`

Si aggiunge la sostituzione di **`passwd`** con **`hostname`**.

`!13:s/passwd/hostname`



Espansione con sostituzione

(La forma generica, un pelino più complicata)

Ad esempio, si voglia sostituire il comando:

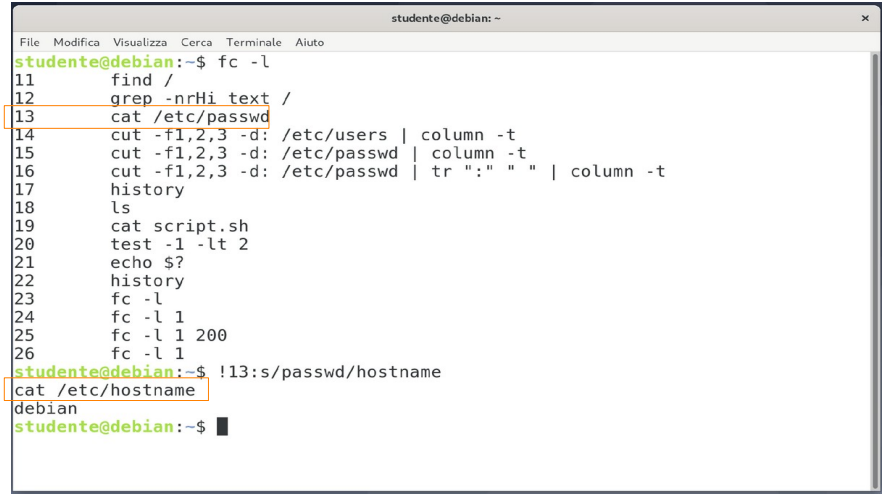
`cat /etc/passwd`

con il comando:

`cat /etc/hostname`

Il risultato finale è l'esecuzione del comando seguente:

`cat /etc/hostname`



Scorciatoie di tastiera – storia

(Sono quelle di EMACS! È possibile usare anche quelle di VIM.)

BASH mette a disposizione una serie di scorciatoie di tastiera per la navigazione della storia dei comandi.

`Up`, **`<CTRL>-p`**: comando precedente.

`Down`, **`<CTRL>-n`**: comando successivo.

`<ALT>-<`: inizio della storia.

`<ALT>->`: fine della storia.

`<ALT>-.`: ultimo argomento.

`<CTRL>-N<ALT>-.`: N-mo argomento (N=0, 1, …, 9).

`<CTRL>-r`: ricerca incrementale all'indietro.

Scorciatoie di tastiera – storia

(Sono quelle di EMACS! È possibile usare anche quelle di VIM.)

Osservazione importante.

La sequenza **<CTRL>-N** è già usata da qualche emulatore di terminale (GNOME Terminal), e non è inviata a BASH.

In tal caso, si può usare la scorciatoia alternativa seguente:

<ESC>-N<ALT>-. : N-mo argomento (N=0, 1, …, 9).

Scorciatoie di tastiera – comando

(Sono quelle di EMACS! È possibile usare anche quelle di VIM.)

BASH mette a disposizione una serie di scorciatoie di tastiera per la navigazione del singolo comando.

- **`<CTRL>-a`**: inizio riga.
- **`<CTRL>-e`**: fine riga.
- **`<Right>`**, **`<CTRL>-f`**: un carattere avanti.
- **`<Left>`**, **`<CTRL>-b`**: un carattere indietro.
- **`<ALT>-f`**: una parola avanti.
- **`<ALT>-b`**: una parola indietro.

Scorciatoie di tastiera – comando

(Sono quelle di EMACS! È possibile usare anche quelle di VIM.)

BASH mette a disposizione una serie di scorciatoie di tastiera per la manipolazione del singolo comando.

<DEL>, **<CTRL>-d**: cancella il carattere sotto il cursore.

<BACKSPACE>: cancella il carattere precedente.

<ALT>-d: cancella fino a fine parola.

<ALT>-<BACKSPACE>, **<CTRL>-w**: cancella fino a inizio parola.

<CTRL>-k: cancella fino a fine riga.

<CTRL>-u: cancella fino a inizio riga.

<CTRL>-_: undo.

Esercizio 19 (1 min.)

Eseguite l'ultima istanza di **`ls`** nella storia dei comandi, aggiungendo l'opzione **`-a`**.

TIPOLOGIA DI COMANDI

Interrogativo

(Ma sì, cambiamo, va. Niente introduzione stavolta.)

In questa introduzione a BASH sono stati eseguiti tanti comandi.
Alcuni di questi comandi sono forniti dall'interprete.
Altri comandi sono esterni all'interprete.
Certi comandi sono alias di altri.
I comandi possono, in realtà, essere anche funzioni.

Domanda: come fare a riconoscere le diverse tipologie di comandi?

Tipologie di comandi

(**Esterno**, **interno**, alias, funzione)

Comando esterno (external command).

Fornito da un eseguibile diverso dalla shell (BASH).
Memorizzato in una delle directory in PATH.
Quando è eseguito, la shell esegue una nuova applicazione.

Comando interno (shell builtin).

Comando fornito dalla shell.
Quando è eseguito, la shell esegue la funzione associata senza caricare altro.

Tipologie di comandi

(Esterno, interno, **alias**, **funzione**)

Alias.

BASH permette la definizione di alias brevi di comandi tramite il builtin **`alias`**.

```
alias ls='ls --color'
```

Funzione.

BASH permette la definizione di funzioni.

```
fun() { a=1; echo $a; }
```

Identificazione tipologia comando

(Si usa il comando **`type`**)

Il comando **`type`** riceve in ingresso uno o più argomenti **C_1**, **C_2**, … **C_N** (ciascuno rappresentante un nome di comando) e ritorna la tipologia associata.

`type` C_1 C_2 ... C_n

Ad esempio, per conoscere la tipologia dei comandi **`help`**, **`ls`** e **`vmstat`**, si digiti:

```
type help ls vmstat
```

L'output del comando

(Semplice ed efficace)

Si dovrebbe ottenere l'output seguente:

```
help è un comando interno di shell
ls ha "ls --color=auto" come alias
vmstat è /usr/bin/vmstat
```

Esercizio 20 (1 min.)

Individuate la tipologia dei comandi seguenti:

- `cd`
- `stat`

Omonimia di comandi

(Un comando interno ed uno esterno condividono lo stesso nome)

È possibile che un comando interno ed uno esterno condividano il medesimo nome.

Perché le cose stanno così?

BASH mira a fornire un ambiente di comandi completo (pur se minimale), indipendentemente dal SO sotto-stante.

Il SO mira a fornire programmi di utilità più completi di quelli forniti da BASH.

Un esempio concreto

(Il comando interno **`printf`** vs il comando esterno **`printf`**)

Un esempio classico è il comando **`printf`** (presente in versione interna ed esterna).

Si provi a stampare l'help di **`printf`**:

```
printf --help
```

Si ottiene un output.

Si provi a stampare l'help di **`/usr/bin/printf`**:

```
/usr/bin/printf --help
```

Si ottiene un altro output.

`/bin/printf` e `/usr/bin/printf`? Eh?

(Sono lo stesso binario? Sono due binari distinti? Che confusione)

Individuazione delle tipologie

(Si usa il comando **`type`** con due argomenti)

Si individuano le categorie di comandi con **`type`**:

```
type printf /usr/bin/printf
```

Si ottiene l'output seguente:

```
printf è un comando interno di shell
/usr/bin/printf è /usr/bin/printf
```

Cosa si è scoperto?

(Captain Obvious to the rescue)

Il comando **`printf`** è un builtin di BASH.

Il comando **`/usr/bin/printf`** è esterno a BASH, e probabilmente fornisce ulteriori funzionalità rispetto al builtin corrispondente.

Individuazione delle tipologie

(Si usa il comando **`type -a`** con un argomento)

Un modo più comodo di scoprire omonimie è quello di eseguire **`type`** con l'opzione **`-a`**, che stampa tutte le tipologie di un comando (interno, esterno, alias, funzione). Tale opzione richiede un argomento, ovvero il nome del comando:

```
type -a printf
```

Si ottiene l'output seguente:

```
printf è un comando interno di shell
printf è /usr/bin/printf
printf è /bin/printf
```

Cosa si è scoperto?

(Captain Obvious to the rescue)

Il comando esterno **`printf`** è fornito da due percorsi:

`/bin`
`/usr/bin`

Perché?
Le distribuzioni GNU/Linux più recenti aderiscono alla unificazione dei binari nella directory **`/usr/bin`**.
Per motivi di retrocompatibilità, rimane anche una versione in **`/bin`**, che sparirà nelle future versioni.
I comandi **`/bin/printf`** e **`/usr/bin/printf`** sono identici.

Ordine di esecuzione

(È l'ordine fornito dal comando **`type -a`**)

L'ordine di esecuzione delle tipologie di un comando è quello fornito dal comando **`type -a`**:

```
type -a printf
```

1. **printf è un comando interno di shell**
2. **printf è /usr/bin/printf**
3. **printf è /bin/printf**

Un esempio più complesso

(Si aggiungono un alias **`printf`** ed una funzione **`printf`**)

Si aggiunga un alias di nome **`printf`**:

```
alias printf="echo alias di printf"
```

Si aggiunga una funzione di nome **`printf`**:

```
function printf() { echo "funzione printf"; }
```

Quante tipologie di comandi **`printf`** esistono ora?
In che ordine viene scelto il comando da eseguire?
È possibile forzare un ordine diverso di esecuzione? Se sì, come?

Individuazione delle tipologie

(Si usa il comando **`type -a`** con un argomento)

Si esegue il comando seguente per elencare le diverse tipologie di comando **`printf`**:

```
type -a printf
```

Si ottiene l'output seguente:

```
printf ha "echo alias di printf" come alias
printf è una funzione
printf() { echo "funzione printf"; }
printf è un comando interno di shell
printf è /usr/bin/printf
printf è /bin/printf
```

Ordine di esecuzione

(È l'ordine fornito dal comando **`type -a`**)

L'ordine di esecuzione delle tipologie di un comando è quello fornito dal comando **`type -a`**:

```
type -a printf
```

1. `printf ha "echo alias di printf" come alias`
2. `printf è una funzione`
   `printf() { echo "funzione printf"; }`
3. `printf è un comando interno di shell`
4. `printf è /usr/bin/printf`
5. `printf è /bin/printf`

Il comando eseguito di default

(È 'alias)

Il comando eseguito di default è l'alias:

```
printf
```

1. printf ha "echo alias di printf" come alias
2. printf è una funzione
   printf() { echo "funzione printf"; }
4. printf è un comando interno di shell
5. printf è /usr/bin/printf
6. printf è /bin/printf

Cambio dell'ordine di esecuzione

(Forzatura funzione)

Per forzare l'esecuzione della funzione si può quotare (fortemente o debolmente) **`printf`** (il quoting disattiva gli alias):

```
'printf'
```

```
1. printf ha "echo alias di printf" come alias
2. printf è una funzione
   printf() { echo "funzione printf"; }
3. printf è un comando interno di shell
4. printf è /usr/bin/printf
5. printf è /bin/printf
```

Cambio dell'ordine di esecuzione

(Forzatura builtin)

Per forzare l'esecuzione del builtin si può eseguire il comando **`builtin`**:

```
builtin printf
```

1. `printf ha "echo alias di printf" come alias`
2. `printf è una funzione`
   `printf() { echo "funzione printf"; }`
3. `printf è un comando interno di shell`
4. `printf è /usr/bin/printf`
5. `printf è /bin/printf`

Cambio dell'ordine di esecuzione

(Forzatura builtin)

Per forzare l'esecuzione del primo comando (builtin o esterno) disponibile si può eseguire il comando **command**:

```
command printf
```

```
1. printf ha "echo alias di printf" come alias
2. printf è una funzione
   printf() { echo "funzione printf"; }
3. printf è un comando interno di shell
4. printf è /usr/bin/printf
5. printf è /bin/printf
```

Cambio dell'ordine di esecuzione

(Forzatura builtin)

Per forzare l'esecuzione di uno specifico comando esterno si può usare il suo percorso (oppure **`command`**, se non sono presenti varianti interne):

```
/usr/bin/printf
/bin/printf
```

```
1. printf ha "echo alias di printf" come alias
2. printf è una funzione
   printf() { echo "funzione printf"; }
3. printf è un comando interno di shell
4. printf è /usr/bin/printf
5. printf è /bin/printf
```

Esercizio 21 (1 min.)

Individuate le diverse tipologie del comando seguente:

```
kill
```