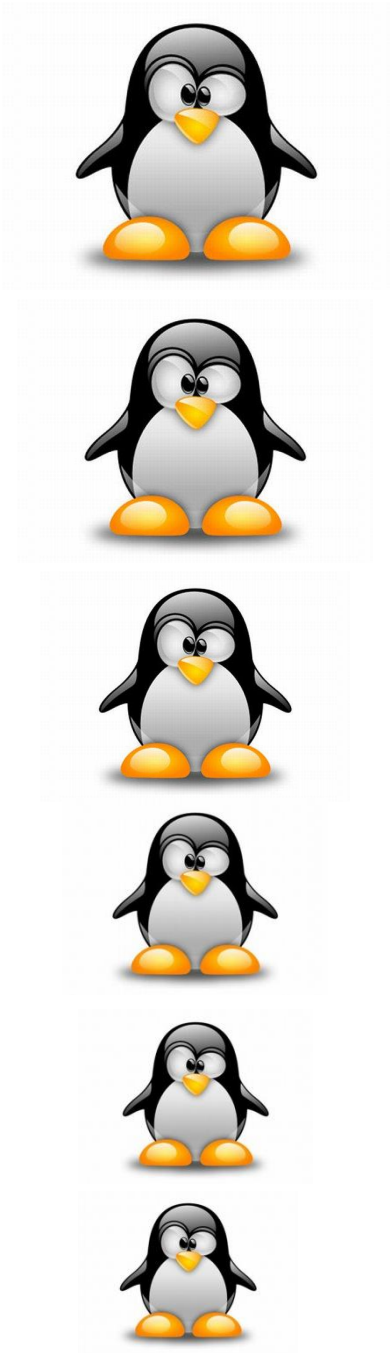
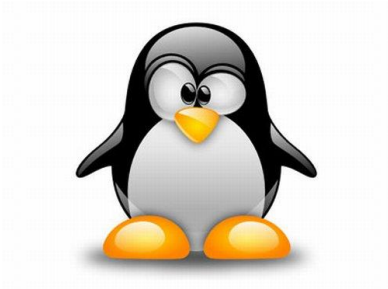


Introduzione allo shell scripting in LINUX

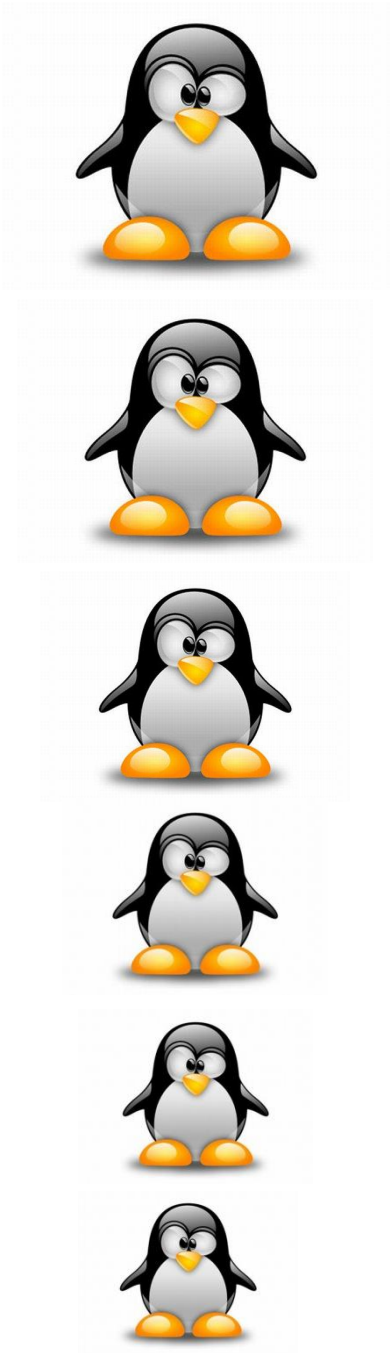
Linux™



Sommario

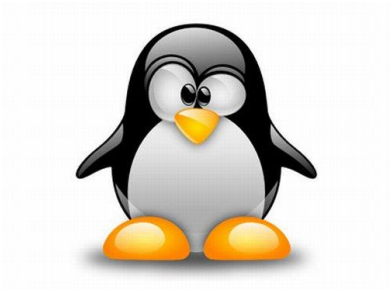
- Perché usare la shell
- Comandi base della shell
- La shell
- Shell scripting

Linux™



Perché usare la shell

Linux™



Che cos'è una "shell"

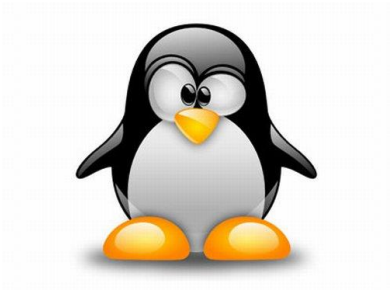
- Una "nicchia", cioè un ambiente comodo e protetto in cui lavorare
- Un programma che utilizza comandi in formato testo per rivolgersi al sistema operativo
- Un esempio: DOS
- In UNIX/Linux: **`sh`** (Bourne shell), **`bash`** (Bourne Again Shell), **`csh`** (C shell), **`ksh`** (Korn shell), **`zsh`** (Z shell), etc.
- L'elenco delle shell disponibili sul proprio sistema è contenuto nel file **`/etc/shells`**
- Nel seguito si farà (quasi) sempre riferimento alla **`bash`**

Linux

Perché usare la shell

- Perchè permette di comprendere meglio l'ambiente di lavoro, cosa stiamo facendo e come risponde il sistema operativo ai comandi che impartiamo
- Perchè vi sono casi in cui l'utilizzo del terminale è indispensabile per risolvere un problema (es.: non viene riconosciuta la scheda video)
- Perchè nella shell si lavora più velocemente
- Per realizzare operazioni complesse su insiemi di file
- Per gestire facilmente computer remoti

Linux



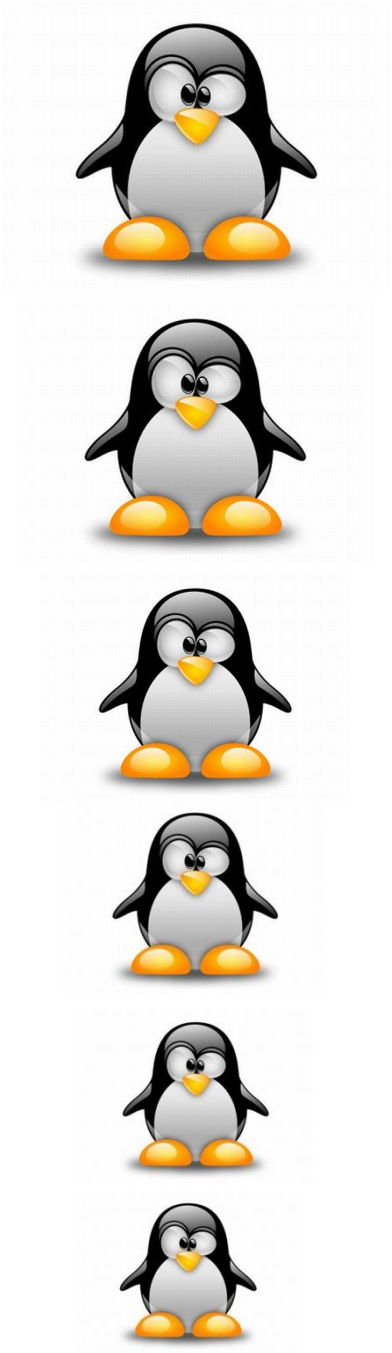
I comandi

- Rappresentano una richiesta di servizio al SO
- Generalmente, hanno il formato

  `comando opzioni argomenti`

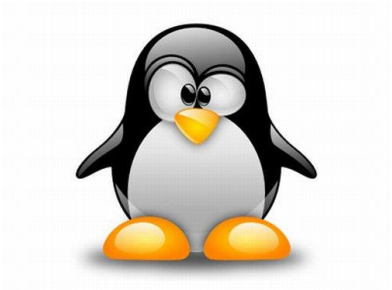
  - Gli argomenti indicano l'oggetto su cui eseguire il comando
  - Le opzioni (di solito) iniziano con un "–" e modificano (specializzano) l'azione del comando
- Esempi di comandi:
  - Comandi built–in della shell
  - Script (file di testo)
  - Codice eseguibile

Linux

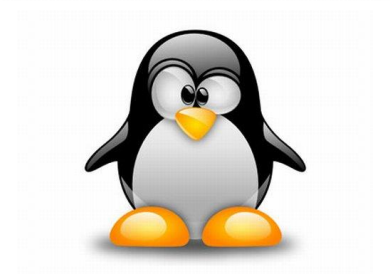


Comandi di shell

Linux™

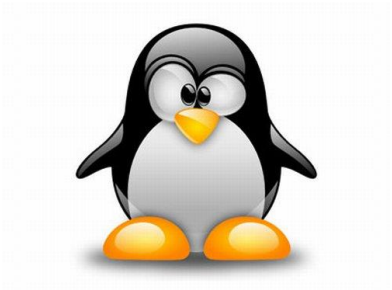


Comandi base della shell



Esempi

- File system: **`ls`**, **`cp`**, **`mv`**, **`rm`**, **`mkdir`**
- Accesso: **`chmod`**, **`chown`**
- Controllo del disco: **`df`**, **`mount`**
- Controllo dei processi: **`ps`**, **`kill`**, **`bg`**, **`fg`**
- Rete: **`ifconfig`**, **`ping`**, **`tracepath`**



File system – Navigazione

- **`pwd`**: Print Working Directory
- **`cd directory`**: Change working Directory ("vai alla directory")
- **`mkdir directory`**: MaKe a DIRectory
- **`rmdir directory`**: ReMove a DIRectory
- **`ls directory`**: LiSt directory content
  - Esempi di opzioni:
    - **`-a`** list all files (also hidden files)
    - **`-l`** long listing
    - **`-g`** group information (not user information)
    - **`-t`** ordered by date
    - **`-R`** recursive
- **`find`**: cerca i file con un certo nome (eventualmente definito tramite wildcard)

Linux






Attributi

- Per ottenere informazioni complete su un file

```
$ ls -lsF prova.txt
1 -rw-r--r- 1 monica staff 213 May 22 00:12 prova.txt
```

(con `l` per lista estesa, `s` per ottenere la dimensione in blocchi, `F` per accodare un carattere particolare a definire il tipo di certi file)

- Si ha…
  - `1` è il numero di blocchi utilizzati dal file
  - `-rw-r--r-` sono i bit di accesso
  - `1` è il conteggio degli hard link
  - `monica` e `staff` sono rispettivamente username e groupname del possessore del file
  - `213` è la dimensione in byte
  - `May 22 00:12` è la data di ultima modifica
  - `prova.txt` è il nome del file
- … gran parte dei dati contenuti in un i–node

Linux

File owner

- Ogni file è associato a:
  - un utente proprietario del file
  - un gruppo (i.e. un insieme di utenti) con diritti speciali sul file
- Come identificare utenti e gruppi
  - user_id (valore intero) ↔ username (stringa)
  - group_id (valore intero) ↔ groupname (stringa)
- Come associare un utente ad un file:
  - Quando un utente crea un file, al file viene automaticamente associato il suo user_id
  - Il proprietario può "cedere il possesso" del file tramite

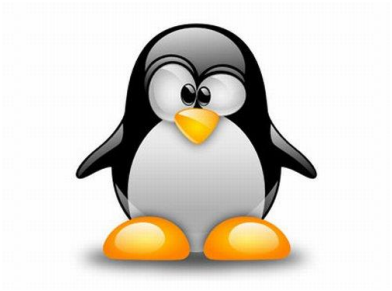
```
$ chown new_user_id file(s)
```


Linux

Gestione dei gruppi

- Come ottenere la lista di appartenenza di un utente ai gruppi:
  `$ groups [username]`
  invocata senza argomenti, elenca i gruppi a cui appartiene l'utente che l'ha invocata, indicando **`username`** ritorna i gruppi ad esso associati
- Come associare un gruppo ad un file:
  - Quando un utente crea un file, al file viene automaticamente associato il suo gruppo corrente; il gruppo corrente iniziale di ogni utente è impostato dall'amministratore
  - L'utente può cambiare il proprio gruppo corrente tramite il comando
    `$ newgrp groupname`
  - Si può invece modificare il gruppo di un file con
    `$ chgrp groupname file(s)`

Linux



Permessi di accesso – 1

- Modalità relativa

  `$ chmod [ugoa] [+-] [rwxX] file(s)`

- Esempi:

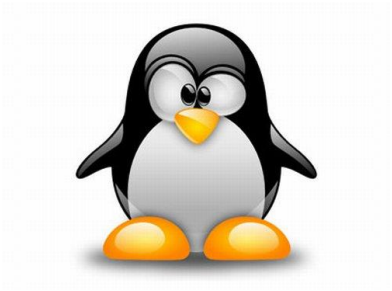
  `$ chmod u+x script.sh`

  aggiunge il diritto di esecuzione per il proprietario del file **`script.sh`**

  `$ chmod -R ug+rwX src/*`

  aggiunge (ricorsivamente) i diritti di lettura e scrittura per il proprietario ed il gruppo relativamente ai file contenuti nella directory **`src`**; aggiunge inoltre il diritto di esecuzione per le directory

Linux



Permessi di accesso – 2

- Modalità assoluta

User			Group			Others		
R 4	W 2	X 1	R 4	W 2	X 1	R 4	W 2	X 1

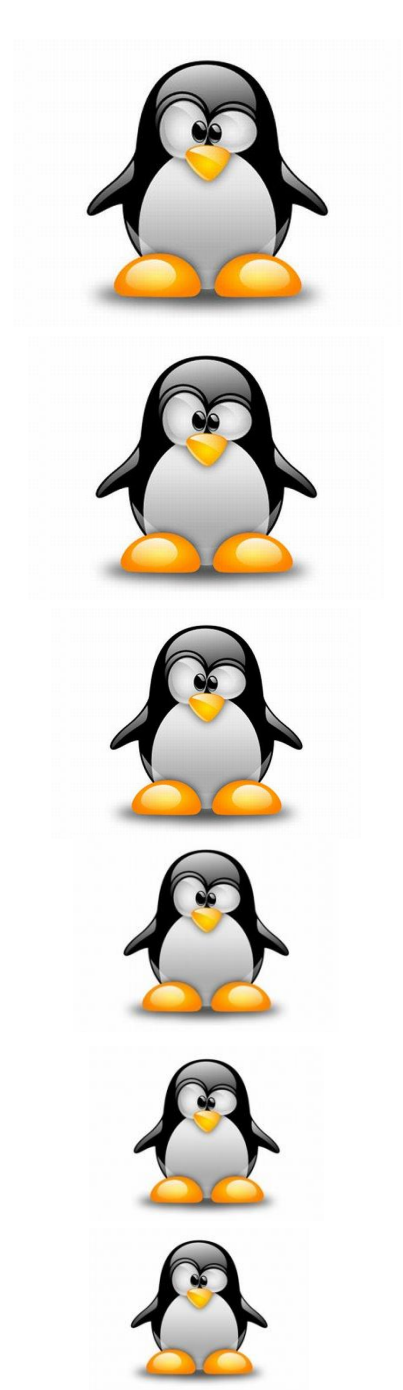
- Esempi:

  `$ chmod 755 pippo.txt`

  assegna diritto di lettura, scrittura ed esecuzione all'utente proprietario, diritto di lettura ed esecuzione al gruppo ed agli altri utenti

  `$ chmod 644`

  assegna diritto di lettura e scrittura all'utente proprietario ed il solo diritto di lettura al gruppo ed agli altri utenti



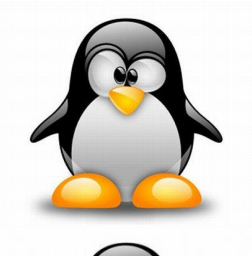
Gestione dei file

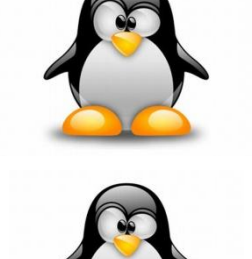
- **`rm`**: ReMove (delete) files
  - Esempi di opzioni:
    - **`-r`** remove recursively beforehand
    - **`-i`** remove after confirmation
    - **`-f`** "forced" removing
    - **`$ rm -rf`** eseguito dal superutente sulla root, "spiana" il sistema
- **`cp`**: CoPy files
- **`mv`**: MoVe (or rename) files
- **`ln`**: LiNk creation (symbolic or not)
- **`more, less`**: page through a text file
- **`df`**: Disk Free (mostra lo spazio libero su disco)
- **`du`**: Disk Usage (mostra lo spazio utilizzato per i file contenuti all'interno di una directory)
- **`mount`**: mostra/cambia l'aspetto del grafo delle directory, montando nuovi file system (remoti)

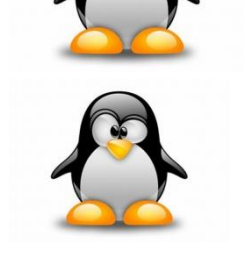
Linux

Gestione dei processi – 1

- Attributi associati ai processi
  - pid: identificatore del processo
  - ppid: identificatore del processo padre
  - nice number: priorità statica del processo, che può essere modificata invocando il comando **`nice`**
  - TTY: terminale associato al processo
  - Effective user_id/group_id: identificatore del proprietario e del gruppo cui appartiene il proprietario
  - Memoria e cpu utilizzate
  - …







Linux



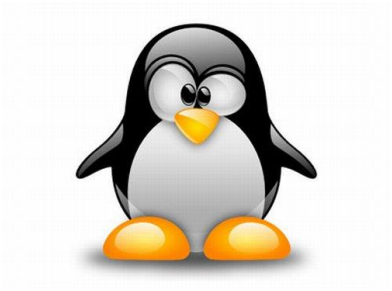
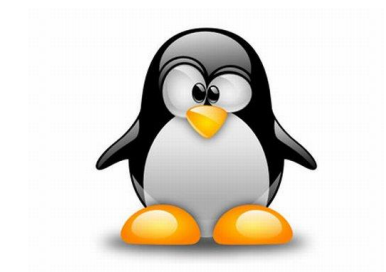
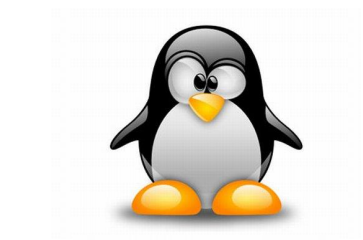



Gestione dei processi – 2

- **`ps`**: Process Status (riporta lo stato dei processi attivi nel sistema)
  - Esempio:

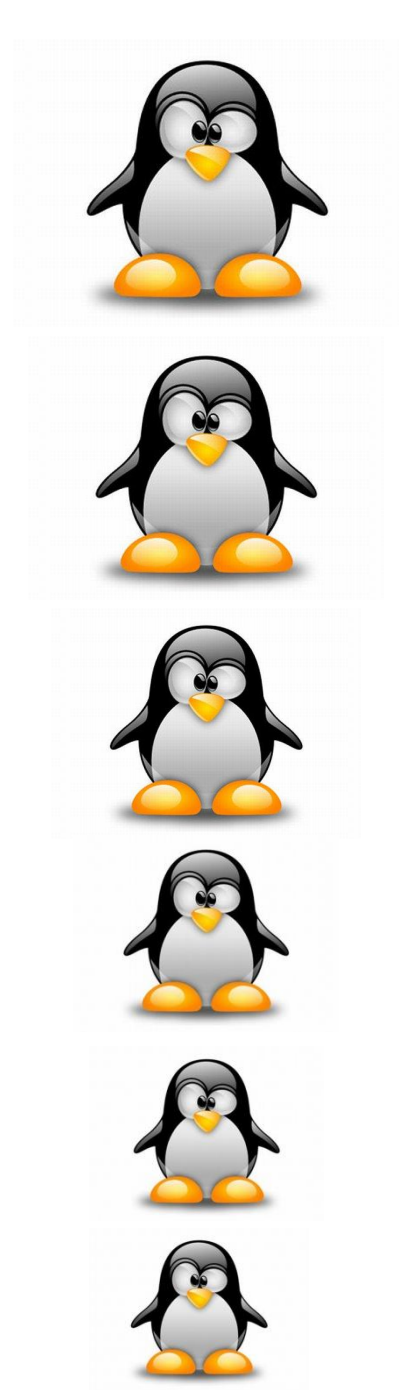
```
$ ps
PID   TTY      TIME       CMD
7431  pts/0    00:00:00   su
7434  pts/0    00:00:00   bash
18585 pts/0    00:00:00   ps
```

- **`nice`**: esegue un comando con una priorità statica diversa
  - Una "**`niceness`**" pari a –20 fornisce priorità massima, mentre il valore 20 corrisponde alla priorità più bassa
  - Il valore di default, per ogni processo, viene ereditato dal padre, ed è solitamente pari a 0
- **`kill`**: termina un processo

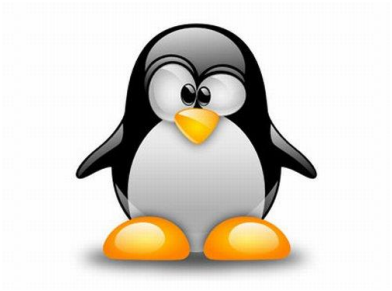

Gestione dei processi – 3

- Processi in foreground
  - Processi che "controllano" il terminale dal quale sono stati lanciati
  - In ogni istante, un solo processo è in foreground
- Processi in background
  - Vengono eseguiti senza che abbiano il controllo del terminale a cui sono "attaccati"
- Job control
  - Permette di portare i processi da background a foreground e viceversa



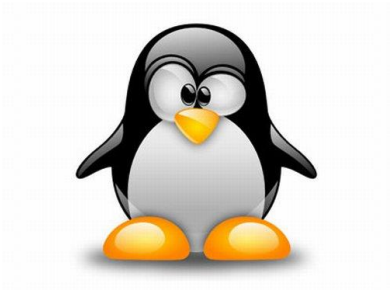
Gestione dei processi – 4

- **`&`**: lancia un processo direttamente in background
  - Esempio:
    `$ prova &`
- **`^Z`**: sospende un processo in foreground
- **`^C`**: termina un processo in foreground
- **`fg`**: porta in foreground un processo sospeso
- **`bg`**: riprende l'esecuzione in background di un processo sospeso
- **`jobs`**: produce una lista dei processi in background
- **`n`**: identifica il numero di un processo
  - Esempio:
    `$ kill 6152`
- Nota: **`jobs`** si applica ai processi attualmente in esecuzione nella shell, mentre la lista completa dei processi in esecuzione nel sistema può essere ottenuta con i comandi **`top`** o **`ps –ax`**

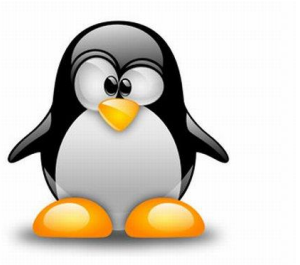


Accesso alla rete

- **`ifconfig`**: configura e controlla un'interfaccia di rete TCP/IP; permette di isolare un computer, disconnettendolo funzionalmente dalla rete mediante disattivazione delle sue interfacce e la modifica dell'indirizzo fisico delle stesse
- **`ping`**: utilizzato per verificare la presenza e la raggiungibilità di un altro computer (identificato mediante il suo indirizzo IP) connesso in rete e per misurare le latenze di trasmissione di rete
- **`tracepath`**: traccia il percorso seguito dai pacchetti ICMP (Internet Control Message Protocol) sulla rete, ovvero l'indirizzo IP di ogni router attraversato per raggiungere il destinatario



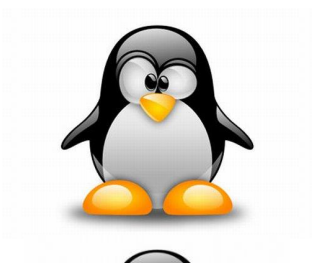
Comandi per "mettere tutto insieme"



- **`cat`**: concatena file e ne mostra il contenuto
  - Esempio

    ```
    $ cat file1 file2 > file3
    ```
- **`grep`**: cerca, in un insieme di file, quelli che contengono una stringa data e riporta la linea contenente la stringa
  - Esempio

    ```
    $ grep -r "zucchero" /home/listespesa/
    ```
- **`sort`**: ordina le linee di un file di testo
- **`head`**, **`tail`**: mostra le prime/ultime n linee di un file



Linux

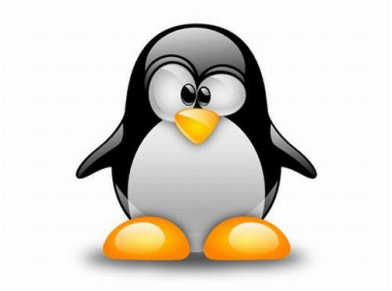
Editor di linea

- **`vi`**: è un potente editor di testo, che si invoca da linea di comando; usa come argomento il nome del file da editare (o apre un nuovo file)
- Dopo l'apertura del file…
  - **`i`** – permette di inserire un testo (ESC per terminare l'inserimento)
  - **`a`** – analogo ad **`i`**, ma inserisce iniziando dalla posizione successiva a quella corrente
  - **`x`** – elimina un carattere
  - **`dd`** – elimina una riga
  - **`:q!`** – chiusura del file senza salvataggio
  - **`ZZ`** – salvataggio ed uscita dall'editor (in alternativa, in successione, **`:w`** e **`:q`**, o ancora **`:wq`**)

Quindi con la shell…

- …possiamo
  - Gestire file
  - Monitorare l'uso del disco
  - Monitorare/gestire i processi
  - Controllare gli accessi utente
  - Gestire/controllare l'accesso alla rete
- Però… si può anche, *piuttosto facilmente*, distruggere completamente il proprio ambiente di calcolo, arrogandosi diritti da (ovvero lavorando come) superutente

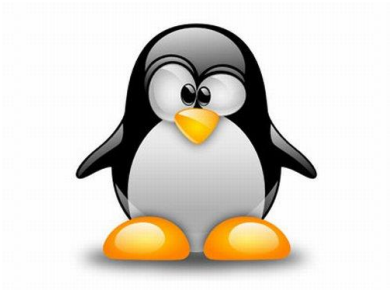
Linux



Come sapere chi fa che cosa?

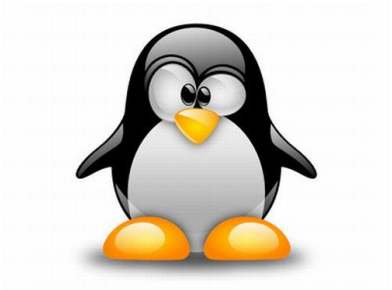
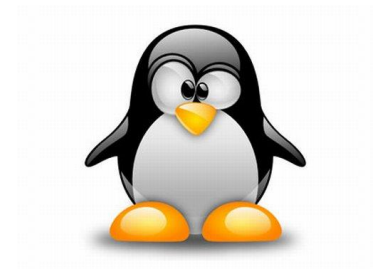
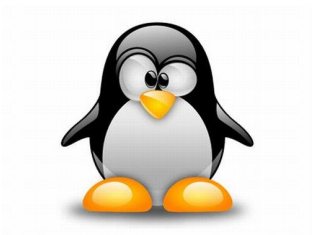
- **`man`**: seguito dal nome del comando, invoca il manuale in linea
- **`apropos`**: aiuta a reperire il comando "giusto" per eseguire una data operazione (mostrando tutti i comandi correlati ad una certa operazione)
- tldp.org: The Linux Documentation Project
- Google

Linux

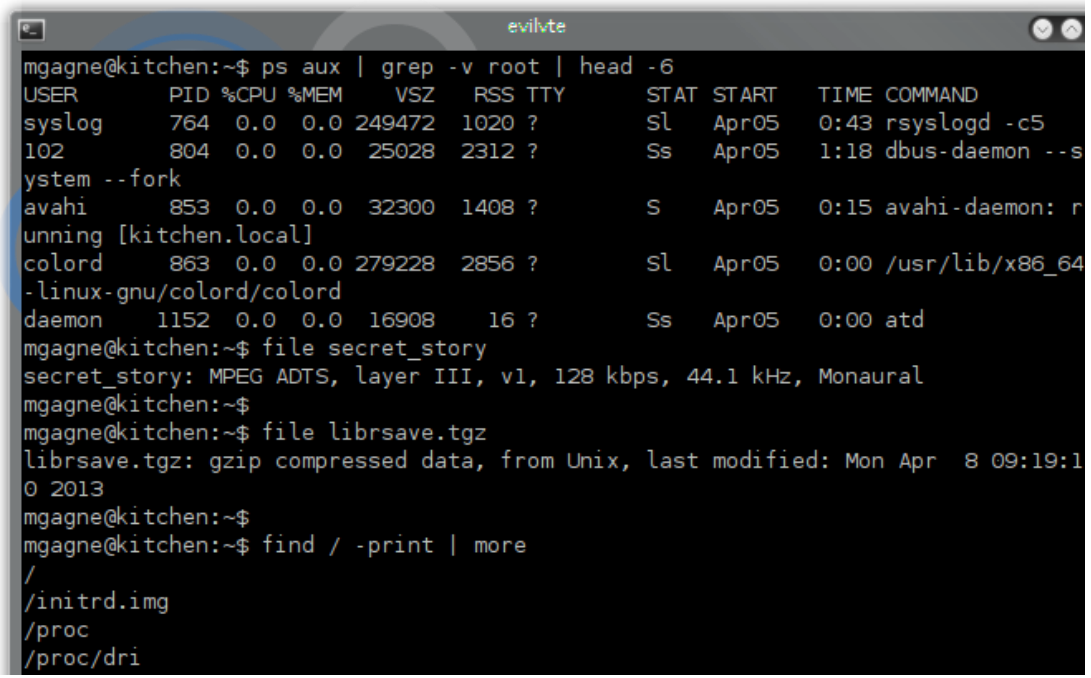


Esercizi

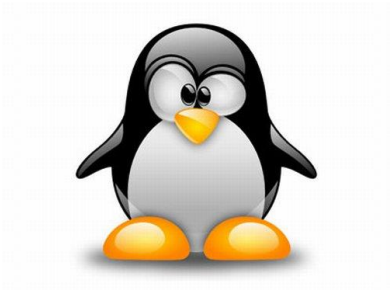
- Creare la directory **`/tmp`**
- Spostarsi nella directory **`/tmp`**
- Nella propria home directory creare il file **`prova.txt`** editandolo con **`vi`**
- Stampare a video il contenuto del file **`prova.txt`**
- Copiare il file **`prova.txt`** in **`riprova.txt`**
- Cancellare il file **`prova.txt`**
- Rinominare il file **`riprova.txt`** in **`rp.txt`**
- Verificare in quale directory ci si trovi


La shell



Linux™



Selezionare una shell

- Quando viene fornito un account su un sistema Linux viene anche selezionata una shell di default
- Per vedere che shell si sta utilizzando:
  `$ echo $SHELL`
  mostra il contenuto della variabile **`SHELL`**
- Per cambiare shell:

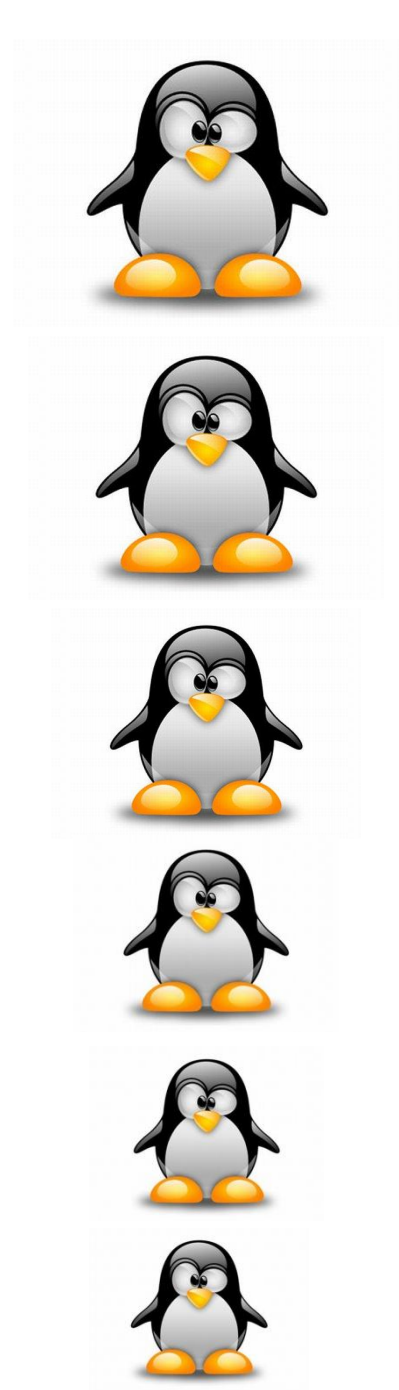
  `$ cat /etc/shells` → visualizza le shell disponibili

  `$ chsh` → Change Shell, visualizza fra parentesi la shell in uso; occorre specificare il percorso completo della nuova shell (es. **`/bin/csh`**)

- Esempio

```
$ echo $SHELL
  /bin/bash
$ chsh
  New shell [/bin/bash]: /bin/csh
```


Linux



Comandi interni ed esterni – 1

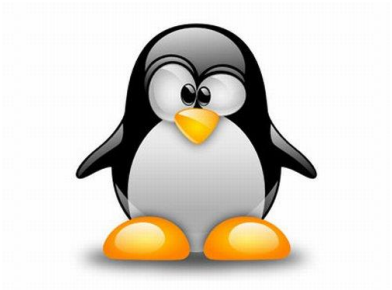
- La shell riesce ad interpretare un insieme di comandi interni (internal command o built–in command), implementati al suo interno
- L'elenco di tali comandi è ottenibile tramite il comando interno **`help`**
- I comandi interni, assieme ai nomi dei file eseguibili ed ai simboli che rappresentano gli operatori aritmetico–logici, formano lo scripting language interpretato dalla shell, con cui è possibile scrivere file di testo, detti script, contenenti comandi che possono essere interpretati ed eseguiti dalla shell stessa
- Viceversa, quando si richiede l'esecuzione di un comando esterno, viene prima ricercato il corrispondente file eseguibile, che viene successivamente caricato in memoria ed eseguito

Comandi interni ed esterni – 2

- Più in dettaglio… quando viene impartito un comando, la shell "prova" ad eseguirlo effettuando, nell'ordine, i seguenti passi:
  1) Tenta di eseguire il comando interno con il nome specificato
  2) Ricerca il file specificato; se per tale file viene descritto il path (sia esso assoluto o relativo), il file viene ricercato soltanto seguendo il path; se il path non è specificato, il file viene ricercato nell'elenco delle directory contenuto nella variabile d'ambiente **`PATH`**
  3) Tenta di eseguire il file specificato

Comandi interni ed esterni – 3

- Per poter essere eseguito, un file deve esistere ed avere il permesso di esecuzione per l'utente che ha impartito il comando: in tal caso la shell crea un processo figlio per l'esecuzione delle istruzioni contenute nel file
  - Se si tratta di un file eseguibile in formato binario (ELF – Executable and Linking Format), la shell crea un processo figlio che esegue i comandi contenuti nel file
  - Se si tratta di un file non binario, la shell lo considera uno script
- La shell può essere terminata per mezzo del comando interno **`exit`**



Metacaratteri – 1

- La shell riconosce alcuni caratteri speciali, chiamati metacaratteri, che possono comparire nei comandi
- Quando l'utente invia un comando, la shell lo scandisce alla ricerca di eventuali metacaratteri, che elabora in modo speciale
- Il comando viene eseguito solo dopo l'interpretazione dei metacaratteri

Metacaratteri – 2

- Esempio:

```
$ ls *.java

Albero.java          div.java          ProvaAlbero.java
AreaTriangolo.java   EasyIn.java       ProvaAlbero1.java
AreaTriangolo1.java  IntQueue.java
```

- Il metacarattere "**`*`**" all'interno di un pathname è un'abbreviazione per una stringa (in questo caso, un nome di file)
- **`*.java`** viene espanso dalla shell con tutti i nomi di file che terminano con l'estensione **`.java`**
- Il comando **`ls`** fornisce quindi la lista di tutti e soli i file con tale estensione

Abbreviazione del pathname – 1

- I seguenti metacaratteri, chiamati wildcard, sono usati per abbreviare il nome di un file in un pathname:
  - ***** stringa di 0 o più caratteri
  - **?** singolo carattere
  - **[ ]** singolo carattere tra quelli elencati
  - **{ }** stringa tra quelle elencate

Linux

Abbreviazione del pathname – 2

- Esempi

```
$ ls /dev/tty?
/dev/ttya /dev/ttyb
```

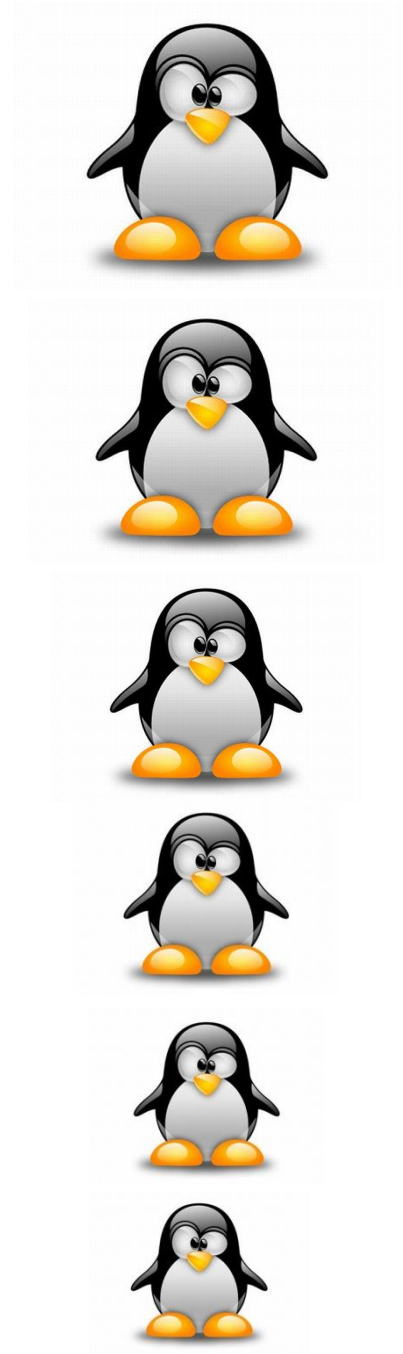
```
$ ls /dev/tty?[234]
/dev/ttyp2 /dev/ttyp3 /dev/ttyp4
/dev/ttyq2 /dev/ttyq3 /dev/ttyq4
/dev/ttyr2 /dev/ttyr3 /dev/ttyr4
```

```
$ ls /dev/tty?[2-4]
/dev/ttyp2 /dev/ttyp3 /dev/ttyp4
/dev/ttyq2 /dev/ttyq3 /dev/ttyq4
/dev/ttyr2 /dev/ttyr3 /dev/ttyr4
```

```
$ mkdir /user/studenti/rossi/{bin,doc,lib}
```

crea le directory **bin, doc, lib**

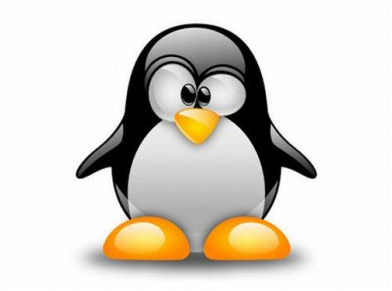
Linux



Il "quoting" – 1

- Il meccanismo del quoting è utilizzato per inibire l'effetto dei metacaratteri
- I metacaratteri a cui è applicato il quoting perdono il loro significato speciale e la shell li tratta come caratteri ordinari

Linux



Il "quoting" – 2

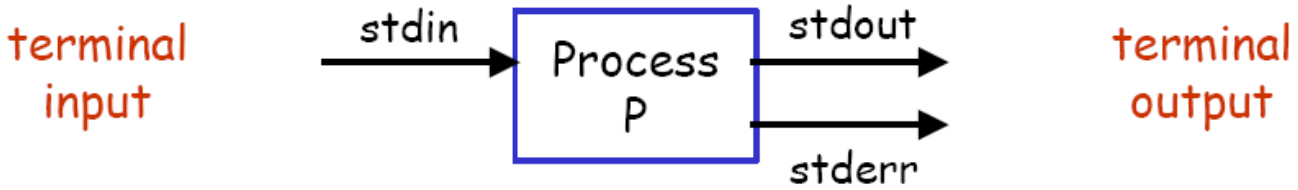
- Ci sono tre meccanismi di quoting:
  - il metacarattere di escape **** inibisce l'effetto speciale del metacarattere che lo segue
    - Esempio

```
$ cp file file\?
$ ls file*
file      file?
```

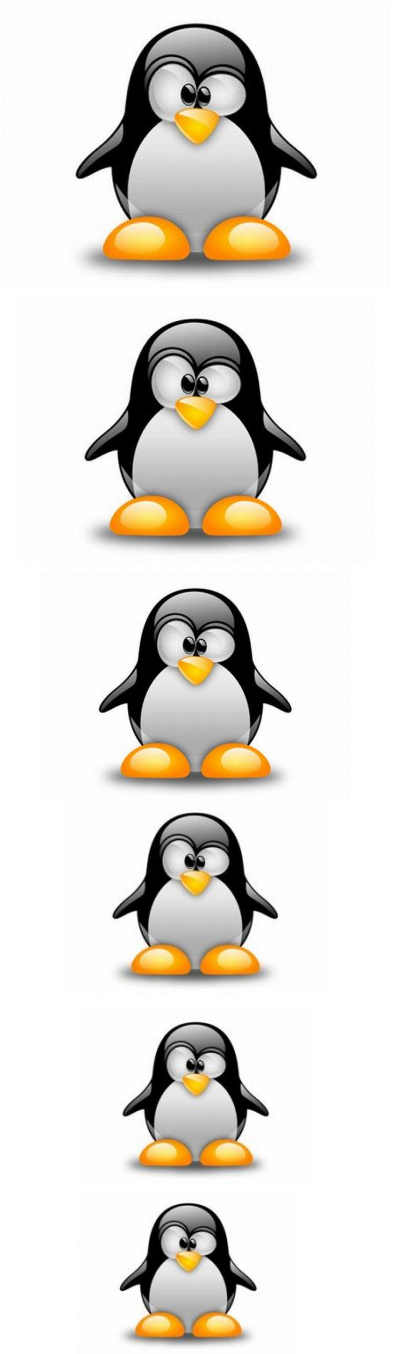
  - tutti i metacaratteri presenti in una stringa racchiusa tra singoli apici perdono l'effetto speciale
  - i metacaratteri per l'abbreviazione del pathname presenti in una stringa racchiusa tra doppi apici perdono l'effetto speciale

Redirezione dell'I/O e pipe

- Ogni processo è associato a tre stream
  - standard input (**`stdin`**)
  - standard output (**`stdout`**)
  - standard error (**`stderr`**)
- Redirezione dell'I/O e pipe permettono di:
  - "slegare" gli stream dalle loro destinazioni abituali
  - "legarli" ad altre sorgenti/destinazioni



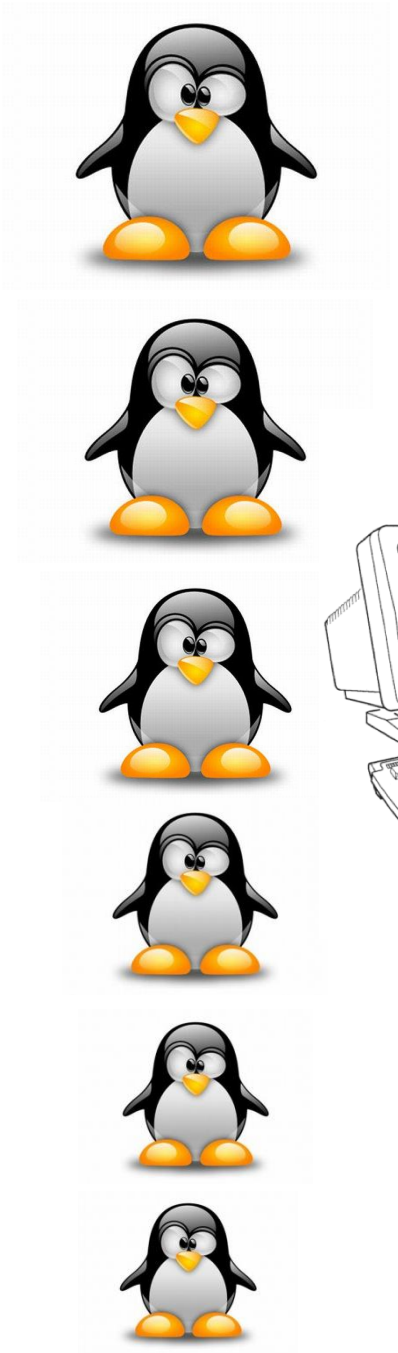
Linux



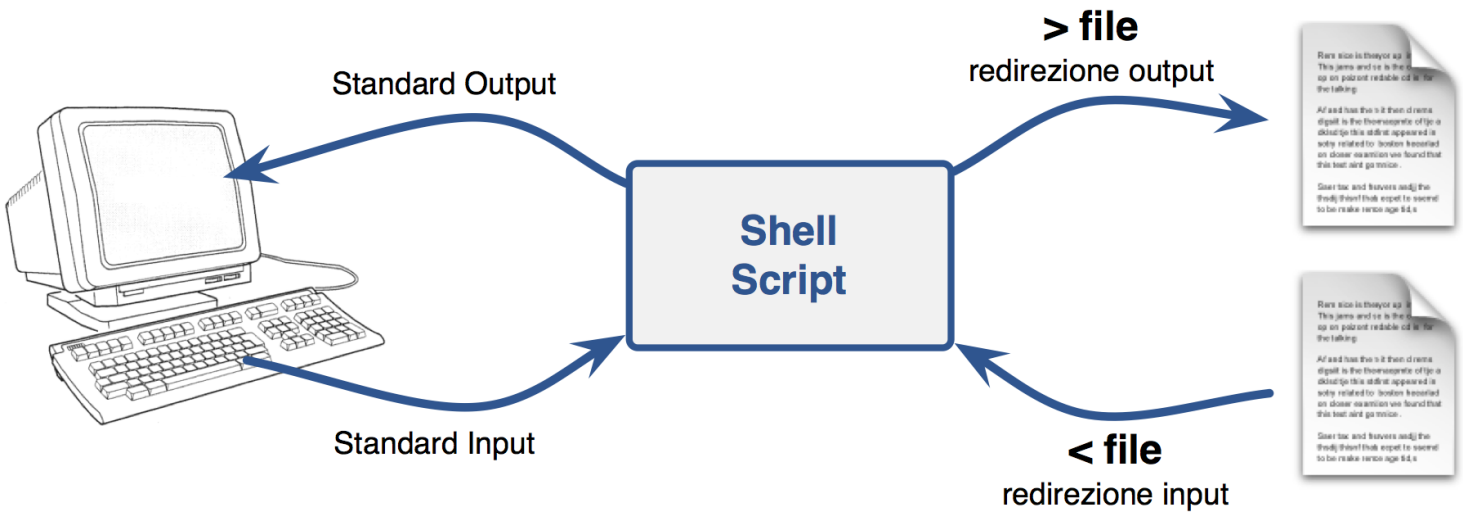
Redirezione dell'I/O – 1

- Redirezione
  - Salvare l'output di un processo su un file (output redirection)
  - Usare il contenuto di un file come input di un processo

Metacarattere	Significato
>	Redirezione dell'output
>>	Redirezione dell'output (append)
<	Redirezione dell'input
<<	Redirezione dell'input dalla linea di comando
2>	Redirezione dei messaggi di errore (bash Linux)



Redirezione dell'I/O – 2



Linux

Redirezione dell'I/O – 3

- Esempi
  - Salva l'output di **`ls`** in **`list.txt`**

    `$ ls > list.txt`
  - Aggiunge l'output di **`ls`** a **`list.txt`**

    `$ ls >> list.txt`
  - Spedisce il contenuto del file **`list.txt`** all'indirizzo monica@dii.unisi.it

    `$ mail monica@dii.unisi.it < list.txt`
  - Redireziona **`stdout`** del comando **`rm`** al file **`/dev/null`**

    `$ rm *.dat > /dev/null`

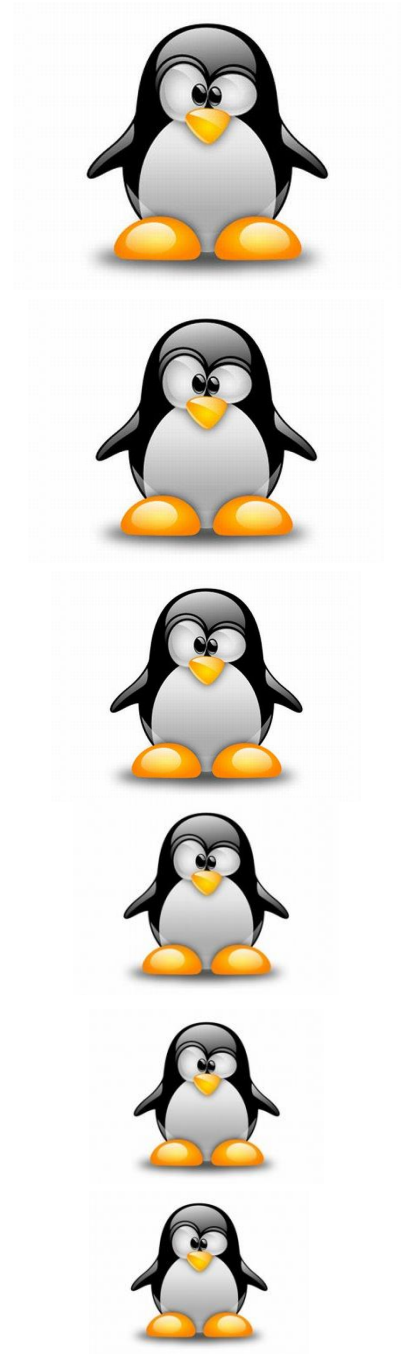
Linux

Redirezione dell'I/O – 4

- Altri esempi

```
$ echo ciao a tutti >file #redirezione dell'output
$ more file
ciao a tutti
$ echo ciao a tutti >>file #redirezione (append)
$ more file
ciao a tutti
ciao a tutti
$ wc <progetto.txt
21 42 77
```

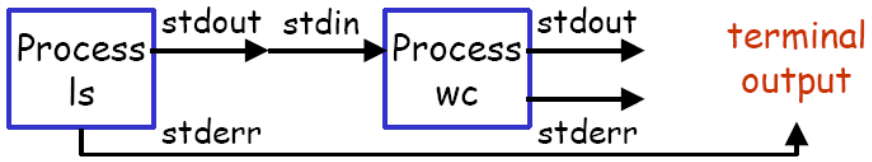

Linux



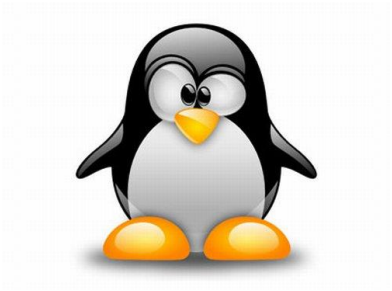
La pipe – 1

- La pipe, realizzata per mezzo del metacarattere `|`, è una catena di montaggio, serve cioè per comporre n comandi "in cascata", in modo che l'output di ciascuno sia fornito in input al successivo
- L'output dell'ultimo comando è l'output della pipeline
- Esempio

```
$ ls
a b c d f1 f2
$ ls | wc –w
6
```



Linux



La pipe – 2

- La sequenza di comandi

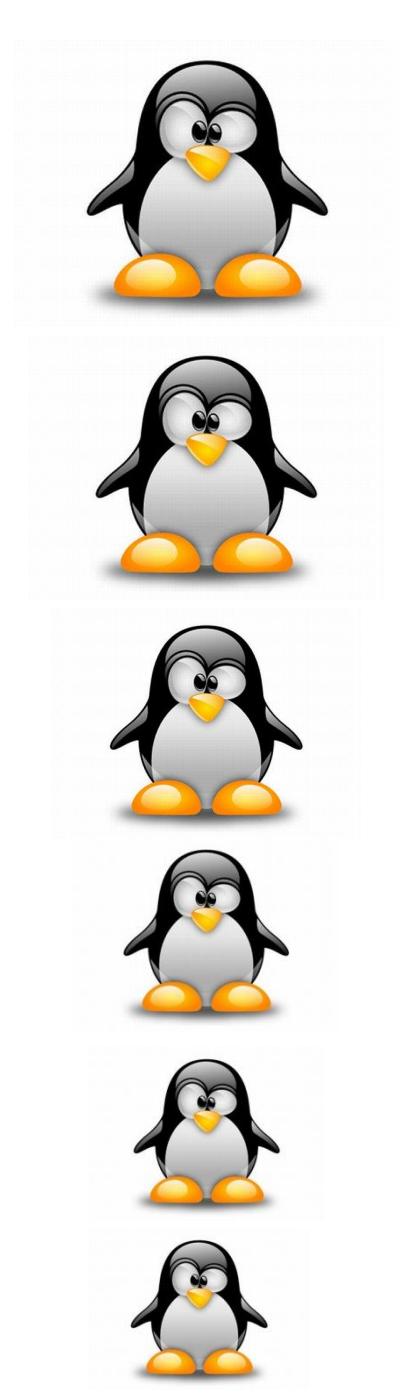
```
$ ls /usr/bin > temp
$ wc -w temp
459
```

  ha lo stesso effetto della pipeline:

```
$ ls /usr/bin | wc -w
459
```

- I comandi **`ls`** e **`wc`** sono eseguiti in parallelo: l'output di **`ls`** è letto da **`wc`** mentre viene prodotto

Linux



Sequenze condizionali (e non)

- Sequenze non condizionali
  - Il metacarattere "`;`" viene utilizzato per eseguire due o più comandi in sequenza
  - Esempio

```
$ date ; pwd ; ls
```

- Sequenze condizionali
  - "||" viene utilizzato per eseguire due comandi in sequenza, solo se il primo termina con un exit code uguale ad 1 (failure)
  - "`&&`" viene utilizzato per eseguire due comandi in sequenza, solo se il primo termina con un exit code uguale a 0 (success)
  - Esempi

```
$ gcc prog.c –o prog && prog
$ gcc prog.c || echo Compilazione fallita
```


Linux

Esecuzione in background

- Se un comando è seguito dal metacarattere **&**:
  - Viene creata una subshell
  - Il comando viene eseguito in background, in concorrenza con la shell attualmente in uso
  - Non prende il controllo della tastiera
  - Utile per eseguire attività lunghe che non necessitano di interazione con l'utente
  - Esempio

```
$ find / –name passwd –print > result.txt &
$ …
$ more result.txt
/etc/passwd
```

Linux

Subshell

- Quando si apre un terminale viene eseguita una shell
- Viceversa, viene creata una subshell:
  - Nel caso di comandi raggruppati
  - Quando viene eseguito uno script
  - Quando si esegue un processo in background
- Caratteristiche delle subshell
  - Hanno la propria directory corrente
  - Esempio

    ```
    $ cd / ; pwd
    ```
  - Nella shell e nella subshell vengono gestite indipendentemente due aree di variabili (parzialmente) distinte

Linux






Variabili – 1

- Ogni shell supporta due tipi di variabili
  - Variabili locali, non "trasmesse" da una shell alle subshell da essa create
    - Utilizzate per computazioni locali all'interno di uno script
  - Variabili di ambiente, "trasmesse" dalla shell alle subshell
    - Utilizzate per la comunicazione fra parent e child shell
- Entrambi i tipi di variabile contengono stringhe
- Ogni shell ha alcune variabili di ambiente inizia-lizzate da file di startup o dalla shell stessa
  - **`$HOME`**, **`$PATH`**, **`$USER`**, **`$SHELL`**, **`$TERM`**, etc.
  - Per visualizzare l'elenco completo, utilizzare il comando **`env`** (ENVironment)

Variabili – 2

- Per accedere al contenuto di una variabile
  - Utilizzare il metacarattere **$**
  - **$name** è la versione abbreviata di **$(name)**
- Per assegnare un valore ad una variabile
  - Sintassi diversa in base alla shell
  - Nel caso di **bash**

    $ nome=valore

    $ nome="valore"
  - Nel caso di **csh**

    $ setenv nome valore
  - Variabili così dichiarate sono locali
  - Per trasformare una variabile locale in una variabile di ambiente, usare il comando **export**

Variabili – 3

- Uso di variabili locali e d'ambiente
- Esempi

```
$ firstname="monica"
$ lastname="bianchini"
$ echo $firstname $lastname
$ export firstname
$ bash
$ echo $firstname $lastname
$ exit
$ echo $firstname $lastname
```


Linux

Esempi

- Scrivere un comando che stampa sul terminale l'elenco di tutti i file nella directory locale che hanno permessi **`rwxrwxrwx`** (o che hanno "rwxrwxrwx" nel loro nome)

```
$ ls -l | grep rwxrwxrwx
```

- Scrivere un comando che conta quanti file sono presenti nella directory corrente

```
$ ls | wc -w
```


Linux

Esercizi – 1

- Scrivere un unico comando per:
  - Copiare il contenuto della directory **`dir1`** nella directory **`dir2`**

```
$ cp -r dir1 dir2
```

  - Fornire la lista dei file contenuti nella home directory il cui nome è una stringa di tre caratteri seguita da una cifra

```
$ ls ~/???[0-9]
```

Esercizi – 2

- Qual è la differenza tra i seguenti comandi?

```
$ ls
$ ls|cat
$ ls|more
```

  - **`ls`** fornisce la lista dei file contenuti nella directory corrente (ordinati alfabeticamente)
  - **`ls|cat`** fornisce la lista dei file contenuti nella directory corrente (ordinati alfabeticamente) in modo che ogni linea contenga il nome di un solo file
  - **`ls|more`** fornisce la lista dei file contenuti nella directory corrente (ordinati alfabeticamente) in modo che ogni linea contenga il nome di un solo file e organizzando l'output su schermo in pagine

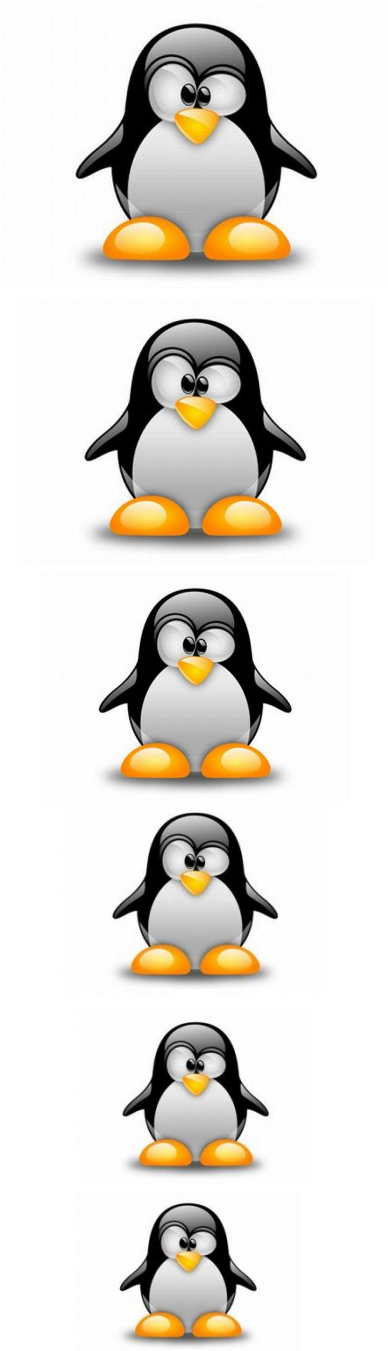
Esercizi – 3

- Che effetto produce la seguente sequenza di comandi?

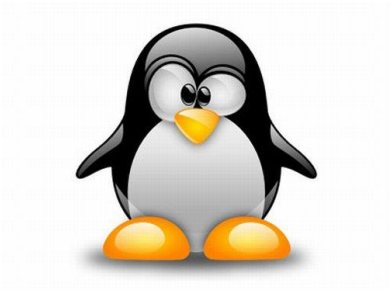
```
$ cd
$ mkdir d1
$ chmod 444 d1
$ cd d1
```

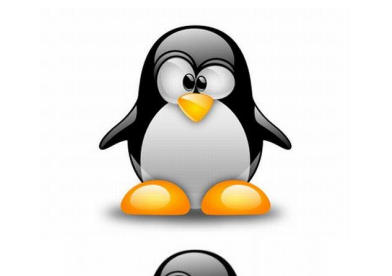
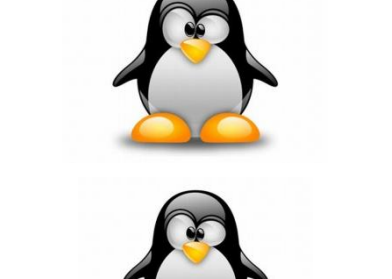
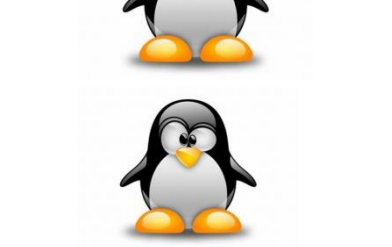
  - **cd** cambia la directory corrente nella home dell'utente
  - **mkdir d1** crea la directory **d1** nella directory corrente
  - **chmod 444** imposta i permessi di **d1** a **r--r--r--**
  - **cd d1** fallisce, provocando la visualizzazione del messaggio di errore **d1: Permission denied** poiché non è "settato" il bit relativo ad **x**

Linux



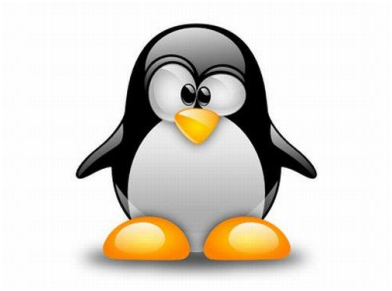
Shell scripting



Script

- Qualunque sequenza di comandi può essere registrata in un file di testo per poi essere eseguita
  ⇨ script
- Utili per sequenze di comandi ripetitive/automatiche
- Per eseguire uno script
  - Editare un file di testo
  - Rendere eseguibile il file tramite il comando **`chmod`**
  - Digitare il nome del file (previa raggiungibilità via path)
- Azioni della shell per eseguire uno script
  - Determina quale shell deve essere utilizzata (in base al contenuto dello script)
  - Crea una subshell che esegue lo script
  - Il file contenente lo script viene passato come argomento alla subshell, che lo esegue



Selezione della shell

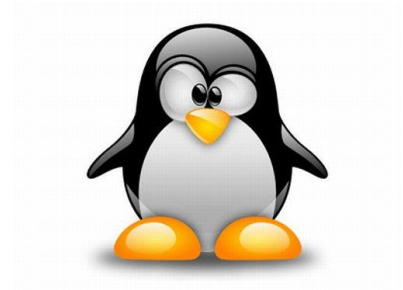
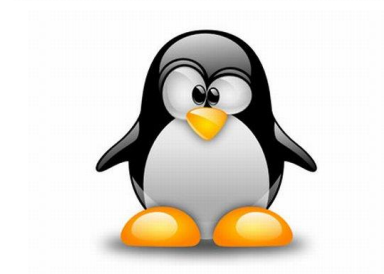
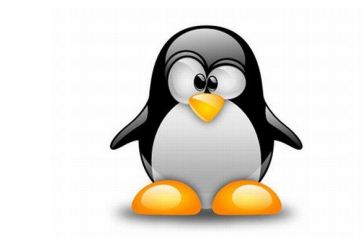
- La selezione della shell che eseguirà lo script è basata sulla prima riga dello script stesso
  - Se contiene solo un simbolo **#**, viene utilizzata la shell da cui lo script è stato lanciato (ovvero una subshell identica)
  - Se contiene **#!pathname**, viene utilizzata la shell identificata dal pathname
    - Esempio
      #!/bin/ksh
  - Altrimenti viene interpretato da una Bourne shell (**sh**)

Un semplice script

```
#!/bin/bash
a=23          # Simple case
echo $a
b=$a
echo $b
```

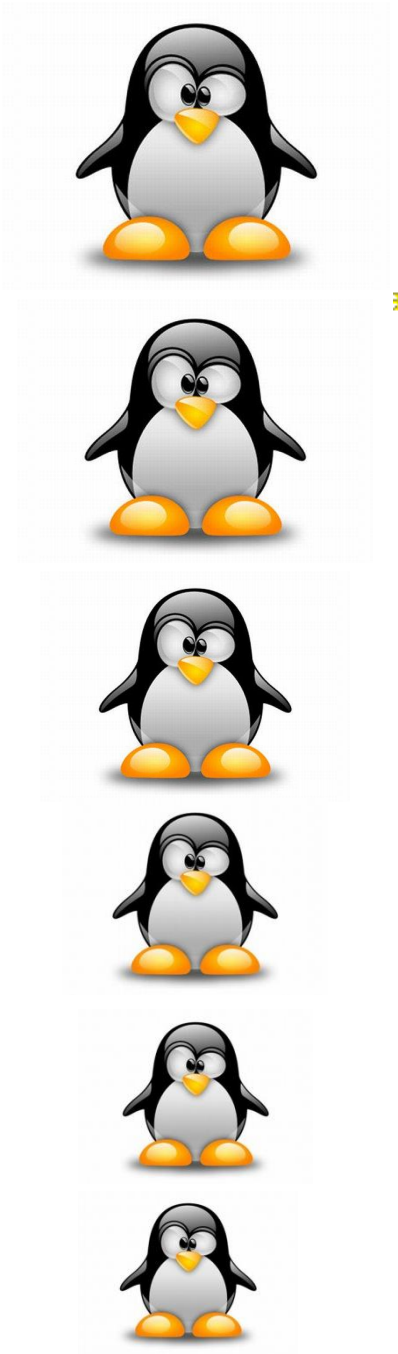
```
# Now, getting a little bit fancier…

a=`echo Hello!`
# Assign result of `echo` command to `a`
echo $a

a=`ls -l`
# Assign result of  `ls -l` command to `a`
echo $a

exit 0
```

Linux


Espressioni – 1

- Valutazione delle espressioni
  - La shell non supporta direttamente la valutazione di espressioni
  - Esiste la utility **`expr expression`**
    - Valuta l'espressione e spedisce il risultato allo standard output
    - Tutti i componenti dell'espressione devono essere separati da spazi
    - A tutti i metacaratteri deve essere prefisso un backslash (ad es., **`\*`**)
    - Il risultato può essere un numero o una stringa



Espressioni – 2

- Operatori
  - Aritmetici: `+ – * / %`
  - Confronto: `= != > < >= <=`
  - Logici: `& | !`
  - Parentesi: `()`
    - Anche alle parentesi deve essere prefisso il back-slash
  - Stringhe
    - **`match string regularExpression`**
    - **`substr string start length`**
    - **`length string`**
    - **`index string charList`**

Espressioni – Esempi

```
$ x=1
$ x=`expr $x + 1`
$ echo $x
2
$ echo `expr $x \* $x`
4
$ echo `expr length "cat"`
3
$ echo `expr index "donkey" "key"`
4
$ echo `expr substr "donkey" 4 3`
key
```


Linux






Exit status

- Quando un processo termina, ritorna un exit status
  - 0 *success* (TRUE)
  - non–zero *failure* (FALSE)
- Comando **`exit nn`**
  - Lo script termina con exit status **`nn`**

Condizioni – 1

- Condizione=exit status
  - Nelle strutture di controllo della shell, gli exit status di un comando vengono utilizzati come condizioni nei test
- Esempio (exit status =0 per input uguali)

```
if cmp a b >/dev/null  # Suppress output
   then echo "Files a and b are identical."
   else echo "Files a and b differ."
fi
```


Linux






Condizioni – 2

- Utility **`test <expression>`**
  - Ritorna un exit code 0 se l'espressione viene valutata vera
  - Ritorna un exit code 1 se l'espressione viene valutata falsa
- Espressioni valutate da **`test`**

`int1 -eq int2`	true if integer **`int1`** == integer **`int2`**
`int1 -ne int2`	true if integer **`int1`** != integer **`int2`**
`int1 -ge int2`	true if integer **`int1`** >= integer **`int2`**
`int1 -gt int2`	true if integer **`int1`** > integer **`int2`**
`int1 -le int2`	true if integer **`int1`** <= integer **`int2`**
`int1 -lt int2`	true if integer **`int1`** < integer **`int2`**

Condizioni – 3

- Espressioni booleane

`!expr`	true if **`expr`** is false
`expr1 -a expr2`	true if **`expr1`** and **`expr2`**
`expr1 -o expr2`	true if **`expr1`** or **`expr2`**
`\( \)`	grouping expression

- Confronti fra stringhe

`str1=str2`	true if strings **`str1, str2`** are equal
`str1!=str2`	true if strings **`str1, str2`** are not equal
`-z string`	true if **`string`** is empty
`-n string`	true if **`string`** contains at least one character

Linux






Condizioni – 4

- Espressioni su file

`-d filename`	true if **`filename`** exists as a directory
`-f filename`	true if **`filename`** exists as a regular file
`-r filename`	true if **`filename`** exists and is readable
`-w filename`	true if **`filename`** exists and is writable
`-x filename`	true if **`filename`** exists and is executable
`-a filename`	true if **`filename`** exists and is non empty

`etc.`

- Sinonimi per test: **`[ ]`**, **`[[ ]]`**
  - Esegue la condizione contenuta tra parentesi quadre
  - **`[]`** è un comando built–in

Strutture di controllo
if–then–elif–else

- Struttura

```
if list1
then
   list2
elif list3
then
   list4
else
   list5
fi
```

- Significato
  - Si eseguono i comandi in **`list1`**
  - Se l'ultimo comando in **`list1`** ha successo, viene eseguito **`list2`**
  - Altrimenti, viene eseguito **`list3`**
  - Se l'ultimo comando in **`list3`** ha successo, viene eseguito **`list4`**
  - Altrimenti viene eseguito **`list5`**

- Note
  - Il costrutto **`if`** può contenere zero o più sezioni **`elif`**
  - La sezione **`else`** è opzionale

Linux

Esempio
if–then–elif–else

```
echo “Chi sei?”
read risposta
if [[ “$risposta” = “1” ]]; then
   echo “amministratore”
elif [[ “$risposta” = “2” ]]; then
   echo “docente”
elif [[ “$risposta” = “3” ]]; then
   echo “studente”
else
   echo “Errore”
fi
```


Linux

Strutture di controllo
case–in–esac

- **Struttura**

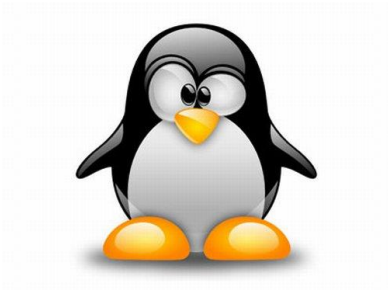
```
case expression in
pattern1 )
   list1
;;
pattern2 )
   list2
;;
esac
```

- **Significato**
  - **`expression`** viene valutata e confrontata con ognuno dei pattern (in ordine)
  - Quando si rileva il primo matching pattern, viene eseguita la lista dei comandi associata
  - Si salta ad **`esac`**
- **Note**
  - **`expression`** è un'espressione di tipo stringa
  - **`pattern`** può contenere wildcard

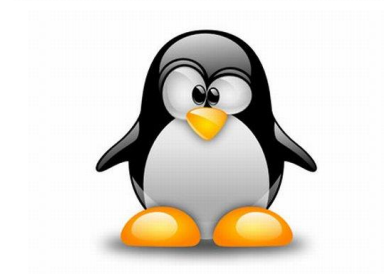
Esempio
case–in–esac

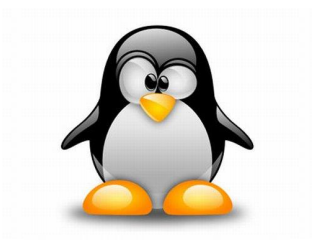
```
#!/bin/bash
   echo  "Inserire un numero da 1 a 9: "
   echo " "
   read numero
   case $numero in
     1) echo "UNO" ;;
     3) echo "TRE" ;;
     5) echo "CINQUE" ;;
     7) echo "SETTE" ;;
     9) echo "NOVE" ;;
     *) echo "Non hai inserito un dispari"
   esac
```

Linux



Strutture di controllo
while–do–done





- **Struttura**

```
while list1
do
   list2
done
```

- **Significato**
  - Il comando **`while`** esegue i comandi in **`list1`** e termina se l'ultimo comando in **`list1`** fallisce
  - Altrimenti, si eseguono i comandi in **`list2`** ed il processo viene ripetuto
  - Un comando **`break`** forza l'uscita istantanea dal loop
  - Un comando **`continue`** forza il loop a riprendere dalla prossima iterazione

Esempio – 1
while–do–done

```
#!/bin/bash
   #
   varA=2;
   varB=10;

   while [ $varA –le $varB ]; do
        echo “$varA”
        varA=‘expr $varA + 1‘
        sleep 1 #ritarda di 1 secondo
   done
```

Linux

Esempio – 2
while–do–done

```
#!/bin/bash
   # tabelline
   if [ "$1" -eq " " ]; then
      echo "Usage: $0 max"
      exit
   fi
   x=1
   while [ $x -le $1 ]; do
      y=1
      while [ $y -le $1 ]; do
          echo `expr $x \* $y` " "
          y=`expr $y + 1`
      done
      echo
      x=`expr $x + 1`
   done
```

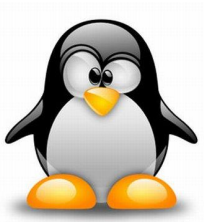
Linux

Strutture di controllo
until–do–done

- Struttura

```
until list1
do
   list2
done
```

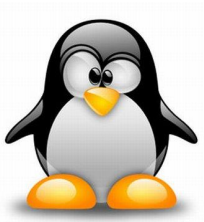
- Significato
  - Il comando **`until`** esegue i comandi in **`list1`** e termina se l'ultimo comando in **`list1`** ha successo
  - Altrimenti, si eseguono i comandi in **`list2`** ed il processo viene ripetuto
  - Un comando **`break`** forza l'uscita istantanea dal loop
  - Un comando **`continue`** forza il loop a riprendere dalla prossima iterazione



Linux

Esempio
until–do–done

```
#!/bin/sh
   #
   a=0;
   b=0;
     until [ $a –gt $b ]; do
     echo “$a”
     a=‘expr $a + 1‘
     sleep 1
   done
```



Linux

Strutture di controllo
for–in–do–done

Struttura

```
for name [in words]
do
   list
done
```

Significato

- Il comando **`for`** esegue un ciclo variando il valore della variabile **`name`** per tutte le parole nella lista **`words`**
- I comandi in **`list`** vengono valutati ad ogni iterazione
- Se la clausola **`in`** è omessa, **`$@`** (insieme di tutti i parametri posizionali passati) viene utilizzato al suo posto
- È possibile utilizzare **`break`** e **`continue`**

Linux

Esempio
for–in–do–done

```
#!/bin/sh
for color in red yellow green blue
do
   echo "one color is $color"
done
```

Output

```
one color is red
one color is yellow
one color is green
one color is blue
```

Linux






Comandi built–in di I/O

- **`echo`**
  - Stampa gli argomenti su **`stdout`**
  - **`echo –n`** evita che sia aggiunto un newline
- **`printf`**
  - Versione migliorata di **`echo`**, con sintassi simile a quella della **`printf`** in C
  - Esempio

```
PI=3.14159265358979
PI=$(printf "%1.5f" $PI)
```

- **`read`**
  - **`read var`** legge l'input da tastiera e lo copia nella variabile **`var`**

Comandi built–in per le variabili – 1

- **`declare`** o **`typeset`**
  - Permettono di restringere le proprietà di una variabile
  - Rappresentano una forma debole di gestione dei tipi
  - Esempi di opzioni
    - **`–r`** dichiara una variabile come read–only (equivalente a **`readonly`**)
    - **`–i`** dichiara una variabile come intera
    - **`–a`** dichiara una variabile come array
    - **`–x`** esporta una variabile (equivalente a **`export`**)






Comandi built–in per le variabili – 2

- **`let`**
  - Il comando **`let`** esegue operazioni aritmetiche sulle variabili
  - Esempi

```
let a=11        #Assegna 11 ad a
let "a <<= 3"   #Shift sin. di 3 posizioni
let "a/=4"      #Divisione di a per 4
let "a-=5"      #Sottrazione di 5 da a
```

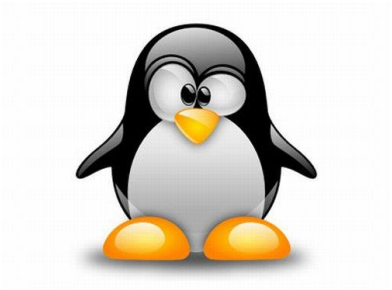
- **`unset`**
  - Cancella il contenuto di una variabile

Comandi built–in vari – 1

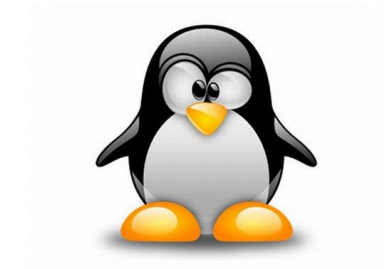
- **`true`**
  - Ritorna sempre un exit status pari a 0 (successo)
- **`false`**
  - Ritorna sempre un exit status pari a 1 (fallimento)
- **`type [cmd]`**
  - Stampa un messaggio che descrive se il comando **`cmd`** è una keyword, un comando built–in o un comando esterno
- **`shopt [options]`**
  - Imposta alcune opzioni della shell
  - Esempio

```
shopt –s cdspell
#permette mispelling in cd
```


Linux

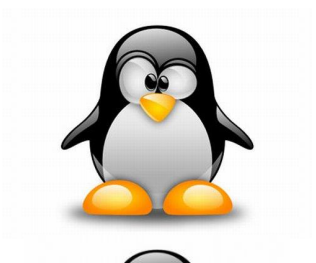


Comandi built–in vari – 2



- **alias, unalias**
  - Un alias bash non è altro che una scorciatoia per abbreviare lunghe sequenze di comandi
  - Esempio

```
alias dir="ls –l"
alias rd="rm –r"
unalias dir
alias h="history –r 10"
```



- **history**
  - Permette di visualizzare l'elenco degli ultimi comandi eseguiti






Comando `find` – 1

- **`find pathlist expression`**
  - L'utility **`find`** analizza ricorsivamente i path contenuti in **`pathlist`** e applica **`espression`** ad ogni file
- Sintassi di **`expression`**
  - **`–name pattern`**
    true se il nome dei file fa matching con pattern, che può includere metacaratteri di shell: **`* [ ] ?`**
  - **`–perm permission`**
    true se **`permission`** corrisponde ai permessi del file
  - **`–print`**
    stampa il pathname del file e ritorna true

Comando `find` – 2

- Sintassi di **`expression`**
  - **`–ls`**
    stampa gli attributi del file e ritorna true
  - **`–atime | –mtime | –ctime –count`**
    true se il file è stato acceduto/modificato/ha cambiato attributi negli ultimi **`count`** giorni
  - **`–exec command`**
    true se l'exit status di **`command`** è 0; **`command`** deve essere terminato da uno **`\;`**
  - …

Linux

Comando `find` – 3

- Esempi

`find –name "*.c" –print`
trova e stampa i nomi dei file sorgente C nella directory corrente ed in tutte lo sottodirectory

`find / –mtime 14 –ls`
elenca tutti i file che sono stati modificati negli ultimi 14 giorni

`find . –name "*.bak" –ls –exec rm {} \;`
cancella tutti i file che hanno estensione **`*.bak`**

Linux

Comandi per la gestione del testo – 1

- **`head [–n] file`**
  - Produce la lista delle prime **`n`** linee di un file (10 default)
- **`tail [–n] file`**
  - Produce la lista delle ultime **`n`** linee di un file (10 default)
- **`cut`**
  - Un strumento per estrarre campi dai file; nonostante esistano metodi più sofisticati, **`cut`** è utile per la sua semplicità
  - Due opzioni particolarmente importanti
    - **`–d delimiter`**: specifica il carattere di delimitazione (tab di default)
    - **`–f fields`**: specifica quali campi stampare

Comandi per la gestione del testo – 2

- **`expand`, `unexpand`**
  - L'utility **`expand`** converte i tab in spazi, viceversa per **`unexpand`**
- **`uniq`**
  - Rimuove linee duplicate consecutive dallo standard input
  - Viene usato spesso nelle pipe con **`sort`**
- **`sort`**
  - Ordina lo standard input linea per linea
  - È in grado di eseguire l'ordinamento lessicografico sulle linee o di gestire l'ordinamento dei vari campi
  - Ad esempio, l'opzione **`–g`** ordina numericamente il primo campo dell'input

Comandi per la gestione del testo
Esempio – 1

```
$ du –s /home/*
10000 /home/monica
500 /home/franco
2345 /home/maggini
26758 /home/studenti
```

```
$ du –s /home/* | sort –gr
26758 /home/studenti
10000 /home/monica
2345 /home/maggini
500 /home/franco
```

Linux

Comandi per la gestione del testo
Esempio – 2

```
$ du –s /home/* | sort –gr | head –2
26758 /home/studenti
10000 /home/monica
```

```
$ du –s /home/* | sort –gr | head –2 \
| cut –f2
/home/studenti
/home/monica
```

Linux






Comandi per la gestione del testo – 3

- **`wc`**
  - Conta linee, parole, caratteri
  - **`–l`**, **`–w`**, **`–c`** conta solo le linee, le parole, i caratteri
  - Certi comandi includono le funzionalità di **`wc`** come opzioni

    … **`| grep foo | wc –l`** è equivalente a

    … **`| grep ––count foo`**
- **`tr`**
  - Utility per la conversione di caratteri, seguendo un insieme di regole definite dall'utente
  - Esempi

    `tr A–Z a–z < filename`

    Stampa **`filename`** trasformando tutti i caratteri in minuscole

    `tr –d 0–9 < filename`

    stampa **`filename`** eliminando tutti i caratteri numerici

Comandi per la gestione del testo Esempio – 3

```
#!/bin/bash
# Changes every filename in the working
  directory to lowercase
for filename in * ; do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`
       #Change name to lowercase
   if [ "$fname" != "$n" ] ; then
       #Rename only files not lowercase
       mv "$fname" "$n"
   fi
done
exit 0
```

Linux

Comandi per il confronto di file

- **`cmp file1 file2`**
  - Ritorna true (exit status 0) se i due file sono uguali, ritorna false (exit status diverso da 0) altrimenti
  - Stampa la prima linea con differenze
  - Con l'opzione **`-s`** non produce stampe (utile negli script)
- **`diff file1 file2`**
  - Ritorna true (exit status 0) se i due file sono uguali, ritorna false (exit status diverso da 0) altrimenti
  - Stampa un elenco di differenze tra i due file (linee aggiunte, modificate, cancellate)
- **`diff dir1 dir2`**
  - Confronta due directory e mostra le differenze (file presenti in una sola delle due)

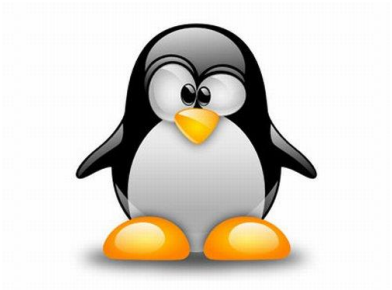
Comandi per la gestione di file – 1

- **`file {file}*`**
  - Identifica il tipo di file a partire dal suo magic number (quando disponibile)
  - L'elenco dei magic number si trova generalmente in **`/usr/share/magic`** (o consultare **`info file`**)
  - Esempio

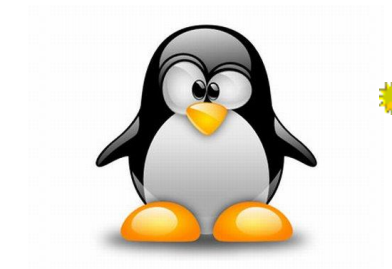
```
$ file prova.sh
prova.sh: Bourne-Again shell script text executable
```

- **`basename file`**
  - Rimuove l'informazione del path da un pathname
- **`dirname file`**
  - Rimuove l'informazione del nome di file da un pathname
- Nota
  - **`basename`** e **`dirname`** lavorano sulle stringhe, non agiscono direttamente sul file

Linux

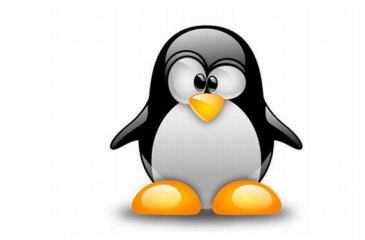


Comandi per la gestione di file – 2



- Esempio

```
$ basename /home/monica/index.html
index.html
$ dirname /home/monica/index.html
/home/monica/
```



Linux

Archiviazione e compressione

- **gzip, gunzip**
- **tar**
- Esempi

Archiviazione

tar zcvf archive-name {file}*

z=comprimi, **c**=crea, **v**=verboso, **f**=su file

Estrazione

tar zxvf archive-name

z=comprimi, **x**=estrai, **v**=verboso, **f**=su file

Linux

Comandi per la gestione del tempo

- **`touch {file}*`**
  - Utility per modificare il tempo di accesso/modifica di un file, ponendolo ad un valore specificato (touch –d)
  - Può essere utilizzata anche per creare un nuovo file
- **`date`**
  - Stampa informazioni sulla data corrente in vari formati
- **`time command`**
  - Esegue il comando **`command`** e stampa una serie di statistiche sul tempo impiegato
  - Esempio

```
$ time find / –name "*.bak" –print
```


Linux

Esercizio – 1

- Scrivere uno script che riceve due numeri e restituisce a schermo il valore massimo

```
#!/bin/bash
if [ $1 -gt $2 ] #Spazi prima e dopo!
then
  echo $1
else
  echo $2
fi
```

Linux

Esercizio – 2

Scrivere uno script che riceve tre numeri e ne restituisce la somma

```
#!/bin/bash
# Calcola la somma di tre numeri
echo "La somma: `expr $1 + $2 + $3`."
```


Linux

Esercizio – 3

- Scrivere uno script che compie le seguenti operazioni:
  - riceve due parametri, **`$1`** e **`$2`** con **`$2`** numero qualsiasi minore di **`$1`**
  - lo script esce con errore se **`$1`** è minore o uguale di **`$2`**
  - calcola il numero **`num`** compreso tra **`0`** e **`$1`** la cui seguente operazione in modulo (**`num % $1`**) è uguale a **`$2`** e lo stampa a schermo

Linux

Soluzione

```
#!/bin/bash
if [ $1 -le $2 ]; then
   printf  "Il secondo parametro deve essere minore del primo\n"
   exit 1
fi
num=0
while [ `expr $num % "$1"` -ne "$2" ]
do
   let num++
   if [ "$num" -gt "$1" ]; then
        echo "Numero non trovato"
        exit 2
   fi
done
echo "Il numero che cercavi e' $num"
exit 0
```