PROGRAMMAZIONE

Capitoli

I. INSTALLAZIONE DI SIMPLY MEPIS 6.0

Le partizioni del disco fisso

L'unica cosa un po' difficile nell'installazione di un sistema Linux moderno è la creazione delle partizioni del disco fisso. Ciò significa che dobbiamo suddividere il disco fisso in parti separate, ognuna delle quali può essere formattata in modo diverso (tipicamente si usano i formati *fat32* e *ntfs* per Windows, *ext3* e *swap* per Linux).

Windows (spesso già presente) dovrà occupare la prima partizione che all'inizio dell'installazione di Linux può essere ridimensionata. La riduzione della partizione di Windows comporta talvolta (tipicamente sui portatili) dei problemi. I dischi fissi recenti hanno capacità enormi e quindi si può chiedere, al momento dell'acquisto del computer, che il rivenditore, quando installa Windows, gli assegni una partizione ad esempio di 20 - 30 GB, ampiamente sufficiente per far girare il sistema.

Definita la partizione di Windows, è molto utile creare una volta per tutte le partizioni per Linux (o chiedere di effettuare questa operazione, comunque reversibile, direttamente al rivenditore). Siccome sotto Linux è facile accedere alle altre partizioni, conviene creare partizioni relativamente piccole per poter così installare più versioni del sistema operativo e per avere spazi separati per diversi tipi di dati.

Tutto ciò rimane valido anche nel caso che si disponga di più dischi fissi. Windows comunque deve allora trovarsi sul primo. Diamo due esempi, uno per un disco fisso di 80 GB, il secondo per un disco fisso di 200 GB.

Portatile con HD da 80 GB

/dev/sda1	ntfs	29 GB	/windows	Windows
/dev/sda2	ext3	20 GB	/	Mepis 6.0
/dev/sda3	swap	1 GB	swap	
/dev/sda4	estesa		—	
/dev/sda5	ext3	10 GB	**/home**	**Dati principali**
/dev/sda6	ext3	14 GB	*	Altro

PC con HD da 200 GB

/dev/hda1	ntfs	20 GB	/windows	Windows
/dev/hda2	ext3	20 GB	/	Mepis 6.0
/dev/hda3	swap	1 GB	swap	
/dev/hda4	estesa		—	
/dev/hda5	ext3	40 GB	**/home**	**Dati principali**
/dev/hda6	ext3	20 GB	/dati	Altri dati
/dev/hda7	ext3	20 GB	*	Altro
/dev/hda8	ext3	20 GB	*	Altro
/dev/hda9	ext3	20 GB	*	Altro
/dev/hda10	ext3	20 GB	*	Altro
/dev/hda11	ext3	20 GB	*	Altro

Se ad esempio adesso sul disco da 200 GB volessimo installare un secondo sistema Linux, potremmo mettere il sistema (cioè la / del secondo sistema) su */dev/hda7*, utilizzando però la stessa */home*. Nell'installazione di questo secondo sistema non dobbiamo quindi creare nuove partizioni, ma solo indicare le cartelle di aggancio (in inglese *mount points*), ad esempio nel modo seguente:

/dev/hda1	ntfs	20 GB	/windows	Windows
/dev/hda2	ext3	20 GB	non usata	Mepis 6.0
/dev/hda3	swap	1 GB	swap	
/dev/hda4	estesa		—	
/dev/hda5	ext3	40 GB	**/home**	**Dati principali**
/dev/hda6	ext3	20 GB	/dati	Altri dati
/dev/hda7	ext3	20 GB	/	Linux 2
/dev/hda8	ext3	20 GB	*	Altro
/dev/hda9	ext3	20 GB	*	Altro
/dev/hda10	ext3	20 GB	*	Altro
/dev/hda11	ext3	20 GB	*	Altro

Va formattata soltanto la nuova partizione */dev/hda7*!

La */dev/hda2* normalmente non viene utilizzata dal secondo sistema; se dovessimo averne bisogno, ad esempio per vedere una vecchia configurazione o per trasferire un programma sul secondo sistema, la possiamo agganciare in ogni momento con il comando

```
mount -t ext3 /dev/hda2 /mnt/sistema-1
```

dopo aver creato, con

```
mkdir /mnt/sistema-1
mkdir /mnt/sistema-2
```

(da effettuare come utente *root*), le cartelle dove agganciare i due sistemi. Altrimenti potremmo già in fase di installazione del sistema 2 prevedere un aggancio ad esempio in */mepis* oppure, dopo aver creato una cartella */mepis*, inserire nella parte superiore (non dinamica) del file */etc/fstab* la riga

```
/dev/hda2 /mepis ext3 defaults,noatime 1 2
```

Un file system gerarchico

Sotto Unix con il termine *file* si comprendono non solo i tipici file di testo o di codice in linguaggio macchina, ma anche le cartelle (in inglese *directories*), i dispositivi periferici e ausiliari.

Il *file system* di Unix è gerarchico, dal punto di vista logico i file accessibili sono tutti contenuti nella cartella principale (in inglese *root*, perché rappresenta la radice dell'albero a cui corrisponde l'organizzazione del file system, da non confondere con l'utente *root*) che è sempre designata con /. I livelli gerarchici sono indicati anch'essi mediante il simbolo /, per esempio */alfa* è il file *alfa* nella cartella root, mentre */alfa/beta* è il nome completo di un file *beta* contenuto nella cartella */alfa*. In questo caso *beta* si chiama anche il *nome relativo* del file.

Se abbiamo installato due sistemi Linux diversi sullo stesso computer, naturalmente ciascuno avrà la sua cartella principale /, mentre la cartella principale ad esempio del primo sistema apparirà, se è stato agganciato, nel file system del secondo sotto un altro nome scelto da noi, ad esempio come */mepis*.

Le prime fasi dell'installazione

Prima dell'installazione preparare alcuni fogli di carta, su cui annotare i singoli passaggi. Si può lavorare in due, con una persona che esegue le operazioni richieste, mentre l'altra tiene il diario di installazione.

La tabella delle partizioni (nuova o della volta precedente) deve essere pronta e in vista!

Proviamo adesso di installare Simply Mepis 6.0 come primo sistema su un portatile con dischi fisso da 80 GB, su cui è già presente una partizione Windows da 20 GB.

(1) Colleghiamo il computer in rete e lo avviamo dal CD; poi scegliamo *BOOT NORMAL*.

(2) *Username: root, Password: root*.
Adesso potremmo usare Linux dal CD per allenarci o per riparare il sistema. Scegliamo però direttamente l'icona *MEPIS Install* per iniziare l'installazione.

(3) *Next*.

(4) Attenzione: Se dobbiamo ancora creare le partizioni o se non ricordiamo più la tabella delle partizioni, scegliamo *Run QTParted*. Questa fase è descritta a pagina 2. Si noti che su un portatile il disco fisso si chiama */dev/sda* invece di */dev/hda*.

(5) Attenzione: Assumiamo che le partizioni siano pronte. Lasciando l'opzione proposta *Custom install on existing partitions* premiamo *Next*. Secondo la nostra tabella, scegliamo *sda2* per la *root*, *sda3* per la *swap*, *sda5* per la */home*. Controllare attentamente. Poi *Next*.

(6) Rispondiamo *Yes* alle domande, se vogliamo veramente effettuare le formattazioni. Queste vengono effettuate, poi inizia l'installazione del nuovo sistema che durerà circa 15 minuti.

(7) Accettiamo *Install GRUB for MEPIS and Windows* su *MBR* con *Next* e rispondiamo *Yes* alla domanda di conferma.

La guida d'installazione continua a pagina 2.

Usare QTParted

Come sempre in KDE possiamo allargare le finestre premendo allo stesso tempo il tasto *Alt* e il tasto destro del mouse.

Si possono creare al massimo 4 partizioni *primarie*, oppure 3 partizioni primarie e una partizione *estesa* (un termine più naturale sarebbe *ausiliaria*) che può contenere altre partizioni, dette *logiche*.

Le prime tre partizioni saranno perciò primarie, la quarta estesa.

(1) Scegliamo il disco, sul portatile quindi */dev/sda*.

(2) Quando dobbiamo cancellare delle partizioni, iniziamo dal basso.
Le cancellazioni si effettuando scegliendo *Operations : Delete*.

(3) Assumiamo che siano rimasti solo la prima partizione, dedicata a Windows, e una partizione virtuale che corrisponde allo spazio vuoto ancora a disposizione.

(4) Dopo aver selezionato lo spazio vuoto (ultima riga), con *Operations : Create* creiamo la seconda partizione primaria.
Appare una finestra, in cui indichiamo la grandezza desiderata di 20 GB, lasciando intatte le impostazioni *Primary Partition* e *ext3*.

(5) Ripetiamo l'operazione, scegliendo però stavolta *linux-swap* invece di *ext3* e impostando la grandezza a 1 GB.
Anche questa partizione è primaria.

(6) Adesso dobbiamo creare la partizione estesa scegliendo *Extended Partition* invece di *Primary Partition*. Lo spazio occupato da questa partizione è tutto lo spazio rimanente disponibile.

(7) Le partizioni successive sono necessariamente logiche. Ne creiamo due in formato *ext3*, ad esempio di 10 e 6 GB.

(8) Per ciascuna delle partizioni modificate (non per la */dev/sda1* quindi!) impostiamo la formattazione appropriata (*ext3* o *linux-swap*) con *Operations : Format*.

(9) Affinché i cambiamenti vengano realmente effettuati, scegliamo *Device : Commit*.
Attenzione: In questo modo i dati sulle partizioni modificate vengono distrutti.

(10) A questo punto con *File : Quit* possiamo uscire da *QTParted* e tornare alla fase (5) dell'installazione a pagina 1.

Ultime fasi dell'installazione

Continuiamo l'installazione iniziata a pagina 1.

(8) In *Common Services to Enable* possiamo disattivare *ppp*, se disponiamo di un router o se siamo direttamente in rete, altrimenti confermiamo l'impostazione con *Next*.

(9) Possiamo scegliere *Computer name: casamia*, *Computer domain: rovigo.it*. Poi *Next*.

(10) Tastiera e lingua con *Keyboard: pt-old*, *Locale: it_IT*. La tastiera portoghese è particolarmente pratica, perché contiene tutti i tasti speciali (soprattutto le parentesi quadre e graffe) in posizioni comode.

(11) Con *Run KDE clock* possiamo impostare l'ora, poi procediamo con *Next*.

(12) Creiamo un utente normale e impostiamo la sua password e la password di *root*.
Attenzione: Se reinstalliamo il sistema con la cartella principale di quell'utente, ad esempio */home/rossi*, già esistente, è preferibile rinunciare per il momento alla creazione dell'utente *rossi* (per il nuovo sistema). Infatti altrimenti vengono effettuate modifiche in quella cartella. Verrà spiegato nell'articolo seguente come riavere la cartella così com'è per il nostro utente *rossi*. *Next*.

(13) Abbiamo finito e premiamo *Finish*. Invece di riavviare direttamente, sembra preferibile tornare al sistema. Rispondiamo quindi con *No* alla domanda se vogliamo riavviare. Chiudiamo la finestra di installazione con *Close*.

(14) Adesso possiamo scegliere *Log Out* dal bottone di *KMenu* (una ruota a cui è sovrapposta una K). Dopo il riavvio possiamo entrare con il nome di utente e la password scelti.

KDE

L'ambiente di lavoro scelto da Mepis è KDE. Lo possiamo configurare in molti modi come vedremo adesso. Assumiamo che il nostro nome utente sia *rossi*. La cartella principale di questo utente viene allora denotata con *~rossi*; nella configurazione tipica essa sarà uguale alla cartella */home/rossi*.

Creazione di un nuovo utente

In KDE arriviamo alla gestione degli utenti tramite la sequenza

```
KMenu : System : More Applications
User Manager (KUser)
```

Assumiamo adesso che vogliamo creare un utente *rossi* di cui però esiste già la cartella */home/rossi*. Possiamo allora procedere nel modo seguente:

(1) Apriamo un terminale e, da *root*, entriamo in */home* con `cd /home`.
Per diventare *root*, si usa il comando `su`.

(2) Rinominiamo la cartella con

```
mv rossi rossi-originale
```

(3) Creiamo il nuovo utente *rossi*, lasciando valide le opzioni impostate *Create home folder* e *Copy skeleton*.

(4) Copiamo i file della vecchia cartella con

```
mv rossi-originale/* rossi
```

(5) Diamo il diritto d'accesso per i nuovi file a *rossi* con

```
chown -R rossi.users rossi
```

(6) Usciamo da *root* con `exit`.

Lasciamo ancora in vita la vecchia cartella *rossi-originale* come backup temporaneo; la possiamo eliminare dopo 1-2 settimane quando siamo sicuri che la nuova configurazione ci soddisfa.

Si noti in particolare che con il comando al punto (4) non sono stati copiati i file nascosti. In genere è preferibile decidere a posteriori quali di questi utilizzare anche nel nuovo sistema.

Installazione della stampante

Assumiamo di voler installare una stampante collegata tramite USB. Allora seguiamo i seguenti passi:

```
KMenu : System Configuration (Settings)
Peripherals : Printers
Administrator Mode}
Add : Add Printer/Class
```

Ci viene proposta una procedura guidata che iniziamo con *Local printer*. Alla fine potremo dare un nome alla nostra stampante, ad esempio *laser* oppure *stamp1*.

Questo nome può essere utilizzato nel caso che abbiamo installato più stampanti.

Il programma enscript

Se la nostra stampante è una stampante PostScript, stamperà direttamente i file PostScript e, in modo piuttosto primitivo, anche i file di testo.

Con il programma *enscript* possiamo però trasformare file di testo in file PostScript oppure mandarli direttamente alla stampante in questo formato. Il programma ha molte opzioni che vengono elencate con `man enscript` dalla shell.

Per una stampa senza data del file *alfa* si può usare il comando

```
enscript -i 2c -f Times-Roman@8 -B alfa
```

Per una stampa con data creiamo Un piccolo programma in Python, da usare con la sintassi `stampa alfa`:

```
#! /usr/bin/python
# stampa
import os,sys

nomefile=sys.argv[1]
fonttesto='-f Times-Roman@8'
data=r'||%D{%y%m%d-%H.%M}'
sopra="-b '%s   $N         $%%/$='" %(data)
fontsopra='-F Times-Roman-Bold@7'
comando='enscript -i 2c %s %s %s %s' \
  %(fonttesto,sopra,fontsopra,nomefile)
os.system(comando)
```

La cartella principale

Apriamo un terminale ed eliminiamo tutti i file (visibili, cioè i cui nomi non iniziano con .), nella cartella principale con

```
rm *
```

Rimangono solo le cartelle. Elimiamo anche tutte le cartelle tranne *Desktop* con più comandi della forma

```
rm -rf nomecartella1 nomecartella2 ...
```

Il comando `rm` è spiegato a pagina 5. Si usi sempre la possibilità di completare un nome iniziato premendo il tasto tabulatore. Per vedere il catalogo si usa `ls`.

Dovrebbe essere rimasta in */home/rossi* soltanto la cartella *Desktop*.

I pacchetti linguistici

Dall'icona di sistema (due ruote) tramite il *Package Manager* installiamo i pacchetti di supporto per la lingua italiana. Poi dobbiamo uscire dal sistema per rendere effettivi i cambiamenti.

kde-i18n-it
language-pack-it
language-support-it

Adesso in molti menu vengono utilizzati termini italiani. Nonostante ciò per uniformità e sicurezza nel seguito usiamo le voci inglesi per la descrizione delle opzioni di configurazione.

Alcuni strumenti

Preleviamo dal sito del corso i file

```
p.bashrc
Menuprogramm
holz-hell.jpg
```

Creiamo una cartella per lo sfondo con

```
mkdir -p Immagini/Sfondo
```

L'opzione `-p` in `mkdir` fa in modo che le cartelle superiori necessarie (in questo caso la cartella *Immagini*) vengano create automaticamente.

Spostiamo *holz-hell.jpg* nella sua cartella con

```
mv holz-hell.jpg Immagini/Sfondo
```

usando il tasto tabulatore per completare i nomi.

Rendiamo eseguibile *Menuprogramm* con

```
chmod +x Menuprogramm
```

e lo collochiamo in una sua cartella con

```
mkdir Software
mv Menuprogramm Software
```

Le seguenti istruzioni salvano una copia di *p.bashrc* in *Software*, creano una copia di sicurezza del file già esistente *.bashrc* e lo sostituiscono con *p.bashrc*. Questo file contiene le nostre personali configurazioni della shell, soprattutto abbreviazioni e, nel *PATH*, l'indicazione delle cartelle dove la shell cerca i programmi da eseguire, e verrà descritto più in dettaglio a pagina 7.

```
cp p.bashrc Software
cp .bashrc .bashrc-originale
mv p.bashrc .bashrc
```

Usciamo dal terminale con *exit*. Quando lo riapriamo possiamo usare `c` per entrare in una cartella o per leggere il contenuto di un file.

Lo sfondo

Abbiamo già messo il file *holz-hell.jpg* in *Immagini/Sfondo*. Possiamo usare questa immagine come sfondo seguendo

```
System Configuration (Settings)
Appearance & Themes : Background
```

Data e ora

Data e ora possono essere impostate attraverso

```
System Configuration (Settings)
System Administration : Date & Time
```

Eliminazione delle icone

Le icone sul desktop vengono eliminate tramite

```
System Configuration (Settings)
Desktop : Behavior
- Show icons on desktop
```

Anche il cestino non serve più, se ci abituiamo a lavorare con il terminale.

La barra dei servizi

Per rendere trasparente la barra dei servizi e per sistemarla sul lato superiore dello schermo seguiamo i menu

```
System Configuration (Settings)
Desktop : Panels
Arrangement : Position : Appearance
+ Enable transparency
```

Eliminiamo a questo punto l'aquario e tutte le piccole icone che appaiono nella metà destra della barra, tranne l'orologio. Anche a sinistra togliamo *Kontact*, il *Desktop Access* e il *Desktop Previewer & Manager*. A questo scopo si clicca sulla barra con il tasto destro del mouse.

A destra dell'orologio rimane un angolo che eliminiamo con

```
System Configurations (Settings)
Desktop: Panels: Hiding
- Show left panel-hiding button
- Show right panel-hiding button
```

Possiamo decidere di lavorare con una pagina sola dello schermo mediante

```
System Configuration (Settings)
Desktop : Multiple Desktops : 1
```

Configurazione dell'orologio

Possiamo modificare l'orologio attraverso il tasto destro del mouse, scegliendo ad esempio altri colori e il formato italiano nella rappresentazione di data e ora.

Configurazione del terminale

Le possibilità non sono molte, ad esempio:

```
Settings : Schema
Black on Light Yellow
Save as Default
```

Installazione di pacchetti

Comodissima da Internet con *Synaptic*.

Disattivare il ripristino

Nell'impostazione predefinita KDE ricostruisce all'inizio di ogni sessione la situazione in cui siamo usciti dal sistema la volta precedente. Per disattivare questa opzione, eseguiamo i seguenti passaggi:

```
System Configuration (Settings)
KDE Components : Session Manager
* Restore previous session
    -> * Start with an empty session
```

KInfoCenter

Per esaminare le partizioni possiamo usare

```
System : KInfoCenter : Partitions
```

Similmente troviamo informazioni sulla memoria centrale con

```
System : KInfoCenter : Memory
```

kruler

Aggiungiamo alla barra dei servizi il programma *kruler*, uno strumento piuttosto utile per misurare distanze sullo schermo.

Per sapere dove il programma si trova, usiamo `type kruler`; troviamo allora che la collocazione del programma è `/usr/bin/kruler`. Cliccando con il tasto destro del mouse sulla barra dei servizi, aggiungiamo il programma, navigando tra le cartelle. L'icona può essere modificata, infatti nel catalogo delle icone si trova una apposita per *kruler*.

Uso del tasto sinistro del mouse

Il tasto sinistro del mouse sul desktop non è associato a un'azione predefinita. Possiamo modificare questa impostazione mediante

```
System Configuration (Settings)
Desktop : Behavior : General : Left Button
No Action -> Application Menu
```

Attivazione delle finestre

Per attivare le finestre semplicemente passando sopra con il mouse (senza dover necessariamente cliccare sulla finestra) usiamo la seguente configurazione:

```
System Configuration (Settings)
Desktop : Window Behavior : Focus
Policy: Focus Strictly Under Mouse
* Auto Raise
Delay: 0 msec
```

Combinazioni Alt e mouse

Molto utili sono le seguenti combinazioni di tasti quando vengono usate all'interno di una finestra:

`Alt mouse-sin`. Spostare la finestra.
`Alt mouse-med`. Posporre la finestra.
`Alt mouse-des`. Ridimensionare la finestra.

II. LAVORARE CON LINUX

La cartella /

Una seduta sotto Unix inizia con l'entrata nel sistema, il *login*. All'utente vengono chiesti il nome di login (l'*account*) e la *password*. Si esce con `exit` oppure, in modalità grafica, con apposite operazioni che dipendono dal *window manager* utilizzato.

Il file system di Unix è gerarchico, dal punto di vista logico i file accessibili sono tutti contenuti nella cartella *root* che è sempre designata con */*.

I livelli gerarchici sono indicati anch'essi con */*, per esempio */alfa* è il file (o la cartella) *alfa* nella cartella root, mentre */alfa/beta* è il nome completo di un file *beta* contenuto nella cartella */alfa*. In questo caso *beta* si chiama anche il *nome relativo* del file.

man

Per molti comandi Unix con `man` seguito dal nome di questo comando viene visualizzato un manuale conciso ma affidabile per il comando. Provare `man ls`, `man who`, `man cd` ecc., un po' ogni volta che viene introdotto un nuovo comando. Tra l'altro `man` è molto utile per vedere le opzioni di un comando. Nessuno le ricorda tutte, ma spesso con `man` si fa prima che guardando nei libri.

ls

`ls` (abbreviazione di *list*) è il comando per ottenere il contenuto di una cartella. Battuto da solo fornisce i nomi dei file contenuti nella cartella in cui ci si trova; seguito dal nome di una cartella, dà invece il contenuto di questa. Il comando ha molte *opzioni*, tipicamente riconoscibili dal prefisso `-`. Le più importanti sono `ls -l` per ottenere il catalogo in formato lungo (cfr. pagina 8) e `ls -a` per vedere anche i file nascosti, che sono quelli il cui nome inizia con un punto (`.`). Spesso le opzioni possono essere anche combinate tra di loro, ad esempio `ls -la`. Vedere `man ls` e fare degli esperimenti.

pwd e whoami

Il prompt è spesso impostato in modo tale che viene visualizzata la cartella in cui ci troviamo. Noi sostituiremo il prompt con un semplice *:)* (segno del sorriso). Per sapere dove siamo si può usare allora il comando `pwd`. Un utente può anche cambiare identità, ad esempio diventare amministratore di sistema per eseguire certe operazioni. Per sapere quale nome di login stiamo usando, utilizziamo `whoami`.

mkdir

Si può creare una nuova cartella *gamma* con `mkdir gamma`.

Per eliminare *alfa* si usa `rm alfa`, se *alfa* è un file normale e `rm -r alfa`, se *alfa* è una cartella. Attenzione: con `rm *` si eliminano tutti i file (ma non le cartelle) contenute nella cartella di lavoro; cfr. pagina 5.

cd

Per entrare in una cartella *alfa* si usa il comando `cd alfa`, dove `cd` è un'abbreviazione di *choose directory*. Ogni utente ha una sua cartella di login, che può essere raggiunta battendo `cd` da solo. La cartella di login dell'utente *rossi* (nome di login) viene anche indicata con *~rossi*. *rossi* stesso può ancora più brevemente denotarla con `~`.

Quindi per l'utente *rossi* i comandi `cd ~rossi`, `cd ~` e `cd` hanno tutti lo stesso effetto.

File il cui nome (relativo) inizia con . (detti *file nascosti*) non vengono visualizzati con un normale `ls` ma con `ls -a`. Eseguendo questo comando si vede che il catalogo inizia con due nomi, . e ... Il primo (.)indica la cartella in cui ci si trova, il secondo (..) la cartella immediatamente superiore, che quindi può essere raggiunta con `cd`.

La cartella di login contiene spesso altri file nascosti, soprattutto i file di *configurazione* di alcuni programmi, il loro nome tipicamente termina con *rc* (*run command* o *run control*).

Il comando file

`file alfa` dà informazioni sul tipo del file *alfa*, con `file *` si ottengono queste informazioni su tutti i file della cartella. In questi comandi di shell l'asterisco (`*`) indica una parola (successione di caratteri) qualsiasi (con un'eccezione). Quindi `*` sono tutte le parole possibili e `*aux` sono tutte le parole che terminano con *aux*. Similmente `*aux*` sono tutte le parole che contengono `aux`. L'eccezione è che se l'asterisco sta all'inizio, non sono comprese le parole che iniziano con .. Per ottenere queste bisogna scrivere `.*` (provare `ls .*rc`).

mv

Il comando `mv` (abbreviazione di *move*) ha due usi distinti. Può essere usato per spostare un file o una cartella in un'altra cartella, oppure per rinominare un file o una cartella. Se l'ultimo argomento è una cartella, viene eseguito uno spostamento. Esempi: `mv alfa beta gamma delta` significa che *delta* è una cartella in cui vengono trasferiti *alfa*, *beta* e *gamma*. Se *beta* è il nome di una cartella, `mv alfa beta` sposta *alfa* in *beta*, altrimenti (se *beta* non esiste o corrisponde al nome di un file) *alfa* da ora in avanti si chiamerà *beta*. Attenzione: se esisteva già un file *beta*, il contenuto di questo viene cancellato e sostituito da quello di *alfa*!

In particolare `mv alfa alfa` cancella il contenuto del file *alfa*.

cp

Con `cp alfa beta` si ottiene una copia *beta* del file *alfa*. Anche in questo caso, se *beta* esiste già, il suo contenuto viene cancellato prima dell'operazione. Quindi non usare `cp alfa alfa`.

Il numero di bytes di cui consiste il file *alfa* viene visualizzato come una delle componenti fornite da `ls -l`. Il semplice comando `wc` (da *word count*) è spesso utile, perché fornisce in una volta sola il numero delle righe, delle parole e dei bytes di uno o più file. Con `wc *` si ottengono queste informazioni per tutti i file non nascosti della cartella.

type

Per avere informazioni sulla locazione e sul tipo di programmi eseguibili e alias di comandi si usa `type`, così ad esempio il risultato di `type python` è */usr/bin/python* oppure */usr/local/bin/python*.

`cd` è un comando un po' particolare e infatti con `type cd` si ottiene *cd is a shell builtin*.

who, w, last e finger

I comandi `who` e `w` indicano gli utenti in questo momento collegati. `w` dà le informazioni può complete sulle attività di quegli utenti.

Con /verb/last -20/ si ottiene l'elenco degli ultimi 20 collegamenti effettuati sul PC; per vedere solo i collegamenti dell'utente *rossi* si usa `last -20 rossi`.

In modo simile si usa il comando `finger` che per ragioni di sicurezza è spesso disattivato perché permette di ottenere le informazioni dall'esterno senza la necessità di effettuare un collegamento al computer remoto.

Gli script di shell

L'interazione dell'utente con il kernel di Unix avviene mediante la *shell*, un *interprete di comandi*. A differenza da altri sistemi operativi (tipo DOS) la shell di Unix è da un certo punto di vista un programma come tutti gli altri, e infatti esistono varie shell, e quando, come noi abbiamo fatto, l'amministratore ha installato come shell di login per gli utenti la *bash* (*Bourne again shell*), ogni utente può facilmente chiamare una delle altre shell (ad esempio la *C-shell* con il comando `csh`). Allora appare in genere un altro prompt, si possono usare i comandi di quell'altra shell e uscire con `exit`. Ognuna delle shell standard è allo stesso tempo un'interfaccia utente con il sistema operativo e un linguaggio di programmazione. I programmi per la shell rientrano in una categoria molto più generale di programmi, detti *script*, che consistono di un file di testo (che a sua volta può chiamare altri file), la cui prima riga inizia con `#!` a cui segue un comando di script, che può chiamare una delle shell, ma anche il comando di un linguaggio di programmazione molto più evoluto come il Perl o il Python, comprese le opzioni, ad esempio `\#! Perl -w`. Solo per la shell di default (la *bash* nel nostro caso), questa riga può mancare, per la *C-shell* dovremmo scrivere `#! bin/csh`. Se il file che contiene lo script porta il nome *alfa*, a questo punto dobbiamo ancora dargli l'attributo di eseguibilità con `chmod +x alfa`. Uno script per la *bash*:


Linus Torvalds

```
echo -n "Come ti chiami?"
read nome
echo "Ciao, $nome!"
```

date

`date` è un comando complesso che fornisce la data, nell'impostazione di default nella forma *Thu Oct 5 00:07:52 CEST 2000*. Usando le moltissime opzioni, si può modificare il formato oppure estrarre ad esempio solo una delle componenti. Vedere `man date`.

cal

`cal` visualizza invece il calendario del mese corrente, con `cal 1000` si ottiene il calendario dell'anno 1000, con `cal 5 1000` il calendario del maggio dell'anno 1000. Con `cal 10 2000` otteniamo

```
    Ottobre 2000
do lu ma me gi ve sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

echo, cat, more e less

Il comando `echo` viene usato, soprattutto all'interno di script di shell, per visualizzare una stringa sullo schermo. Il comando `cat` viene talvolta utilizzato per vedere il contenuto di un file di testo (ad esempio `cat alfa`, ma non permette di andare avanti e indietro. Alla visualizzazione di file servono invece `more` e `less`, di cui il secondo è molto più confortevole.

Il nome di `cat` deriva da *concatenate* e infatti l'utilizzo naturale di questo comando è la concatenazione di più file.

Con `cat alfa beta gamma > delta` si ottiene un file *delta* che contiene l'unione dei contenuti di *alfa*, *beta* e *gamma*. In questi casi bisogna sempre stare attenti che il contenuto del file di destinazione viene cancellato prima dell'esecuzione delle operazioni di scrittura, e quindi ad esempio `cat alfa > alfa` cancella il contenuto di *alfa*. Quale sarà l'effetto di `cat alfa beta > alfa`?

Per aggiungere il contenuto di uno o più file a un altro, invece di `>` bisogna usare `>>`. Per esempio `cat alfa beta >> gamma` fa in modo che dopo l'operazione *gamma* contiene, nell'ordine, i contenuti di *gamma* (prima dell'operazione), *alfa* e *beta*.

head e tail

Meno usati sono `head` (*testa*) e `tail` (*coda*). Con `head -9 alfa` si ottengono le prime 9 righe di *alfa*, le ultime 6 righe invece con `tail -6 alfa`.

df, du e free

`df` visualizza l'elenco delle partizioni e lo spazio ancora disponibile su ciascuna di esse. `du` indica in KB lo spazio occupato dai file e dalle sottocartelle di una cartella, mentre `du *.ps` indica lo spazio occupato dai file il cui nome termina in *.ps*.

`free -t` mostra la RAM disponibile.

uname e hostname

Con `uname -a` si ottengono informazioni sul proprio sistema (nome del sistema operativo, nome del PC in rete, versione del kernel, data e ora, tipo del processore). Il nome del PC lo si ottiene anche con `hostname`.

gzip, bzip, compress e tar

`gzip alfa` trasforma il file *alfa* nel file *alfa.gz* che è una versione compressa di quello originale. Per ottenere di nuovo l'originale si usa `gunzip alfa.gz`.

Più recente di `gzip` è `bzip2`, che trasforma *alfa* in *alfa.bz2*. La decompressione si ottiene con `bunzip2 alfa.bz2`. Il comando più vecchio `compress` produce un file *alfa.Z* che viene decompresso con `uncompress alfa.Z`. Si usa poco, ma conviene conoscerlo per poter decomprimere file *.Z* che si trovano su Internet oppure per la compressione/decompressione su un sistema Unix su cui `gzip` e `bzip2` non sono installati.

Il comando `tar` serve a raccogliere più file o intere cartelle in unico file.

Con `tar -cf raccolta.tar alfa beta` si ottiene un file che raccoglie il contenuto di *alfa* e *beta*, che rimangono intatti, in un unico file *raccolta.tar* non compresso; spesso si eseguirà un `gzip raccolta.tar`, ottenendo il file *raccolta.tar.gz*. Con `tar -xf raccolta.tar` si ottengono gli originali, con `tar -tf raccolta.tar` si vede il contenuto.

find, locate e updatedb

Questi due comandi vengono utilizzati per cercare un file sul disco fisso. `locate` (talvolta `slocate`) è molto più veloce perché invece di cercare sul disco cerca il nome richiesto in un elenco che viene regolarmente aggiornato (alle 4 di notte). `locate` non può tener conto di cambiamenti avvenuti dopo l'ultimo aggiornamento.

Come utente *root* possiamo però eseguire il comando `updatedb` che aggiorna il database di ricerca utilizzato da `locate`.

Più complesso è il comando `find` che rispecchia la situazione attuale del disco. Il formato generale del comando è `find A B C`, dove *A* è la cartella in cui si vuole cercare (la cartella in cui ci si trova, quando *A* manca), *B* è il criterio di ricerca (tutti i file, se manca), e *C* è un comando da eseguire per ogni file trovato.

I criteri di ricerca più importanti sono `-name N`, dove *N* è il nome del file da cercare, che può contenere i caratteri speciali della shell e deve in tal caso essere racchiuso tra apostrofi, ad esempio `-name 'geol*'` per cercare tutti i file il cui nome inizia con *geol*, `-type f` per cercare tutti i file normali, `-type d` per trovare le cartelle.

Si possono anche elencare i file che superano una certa grandezza o usare combinazioni logiche di più criteri di ricerca. Il comando (parte *C*) più importante è `-print`, che fa semplicemente in modo che i nomi dei file trovati vengono scritti sullo schermo. Con `-exec` invece è possibile eseguire per ogni file un certo comando; si fa molto meglio però con appositi programmi in Python o Perl.

rm e un errore pericoloso

Con `rm alfa/*` si eliminano tutti i file della cartella *alfa*. Qui è facile incorrere nell'errore di battitura `rm alfa/ *`. Il sistema protesterà perché per la cartella ci vorrebbe l'opzione `-r`, ma nel frattempo avrà già cancellato tutti i file dalla cartella di lavoro.

Questa è una delle ragioni per cui è bene impostare (ad esempio mediante un *alias*) il comando `rm` in modo tale che applichi automaticamente l'opzione `-i` che impone che il sistema chieda conferma prima di eseguire il comando.

Quando si è veramente sicuri di non sbagliare si può *forzare* l'esecuzione immediata con `rm -f` (per i file) e `rm -rf` per le cartelle.

tee

Il comando `tee alfa` ha come output il proprio input, che però scrive allo stesso tempo nel file *alfa*. Poco utile da solo, viene usato in pipelines (cfr. pagina 7).

`ls | tee alfa` mostra il catalogo della cartella sullo schermo, scrivendolo allo stesso tempo nel file *alfa*.

```
ls -l | tee alfa | grep Aug > beta
```

scrive il catalogo in formato lungo nel file alfa, e lo invia (a causa della seconda pipe) a `grep` che estrae le righe che contengono `Aug`, le quali vengono scritte nel file *beta*.

`tee -a` non cancella il file di destinazione, a cui aggiunge l'output.

gqview

Un bellissimo programma per la visualizzazione di immagini, adatto soprattutto per vedere in fila tutte le immagini di una cartella. Cliccando con il tasto sinistro del mouse sull'immagine visualizzata, fa vedere la prossima; si torna all'immagine precedente invece con il tasto destro.

Aggiungerlo alla barra dei servizi.

import

Per catturare una parte dello schermo come immagine si può usare il programma `import` che, come `convert`, fa parte della collezione di strumenti *ImageMagick*. In pratica si può sempre usare la stessa istruzione:

```
import -quality 100 ~/immagine.png
```

Per aggiungerlo alla barra dei servizi dobbiamo conoscere il nome completo, */usr/bin/import*, che scopriamo tramite `type`. Le opzioni vanno indicate nell'apposita riga della finestra di configurazione.

convert

Conosciamo già dal corso di Algoritmi (pag. 2) il programma `convert` che permette una conversione di immagini in molti formati, ad esempio con

```
convert imm.ps imm.png
convert -transparent black imm.ps imm.png
```

La scelta della password

L'utente *root* ha tutti i diritti sulla macchina, quindi può creare e cancellare i file degli altri utenti, leggere la posta elettronica, installare programmi ecc. Anche sulla propria macchina si dovrebbe lavorare il meno possibile come root e quindi creare subito un account per il lavoro normale.

Per cambiare la propria password si usa il comando `passwd`.

La password scelta dall'utente non viene registrata come tale nel sistema (cioè sul disco fisso), ma viene prima crittata (diventando ad esempio `P7aoXut3rabuAA`) e conservata insieme al nome dell'account. Ad ogni login dell'utente la password che lui batte viene anch'essa crittata e la parola crittata ottenuta confrontata con `P7aoXut3rabuAA` e solo se coincide l'utente può entrare nel sistema.

Siccome la codifica avviene sempre nello stesso modo (in verità c'è un semplice parametro in più, il *sale*) password troppo semplici possono essere scoperte provando con un dizionario di parole comuni. Esistono programmi appositi che lo fanno in maniera sistematica e abbastanza efficiente. Non scegliere quindi `alpha` o `Claudia`; funzionano già meglio combinazioni di parole facili da ricordare, ma sufficientemente insolite come `6globidighiaccio` (purtroppo valgono solo le prime 8 lettere, quindi questa scelta equivale a `6globidi`), oppure le prime lettere delle parole di una frase che si ricorda bene (*sei giganti buoni su un globo di ghiaccio* → `sgbsugdg`). Non scrivere la password su un foglietto o nella propria agenda.

Tasti speciali della shell

Alcuni tasti speciali nell'uso della shell: `^c` per interrompere un programma (talvolta funziona anche `^d` oppure `^\`, `^u` per cancellare la riga di comando, `^k` per cancellare la parte della riga di comando alla destra del cursore,`^a` per tornare all'inizio e `^e` per andare alla fine della riga di comando. Il cappuccio `^` indica qui il tasto *Ctrl*, quindi `^a` significa che bisogna battere *a* tenendo premuto il tasto *Ctrl*. I tasti `^a`, `^e` e `^k` vengono utilizzati nello stesso modo in Emacs, nel nostro editor ECP e nell'input di molti altri programmi. Lo stesso vale per le frecce orizzontali ← e → che, equivalenti ai tasti `^b` e `^f`, permettono il movimento indietro e avanti nella riga di comando, per il tasto `^d` che cancella il carattere su cui si trova il cursore, e il tasto `^h` che cancella il carattere alla sinistra del cursore.

Le frecce ↑ e ↓ si muovono nell'elenco dei comandi recenti (*command history*); sono molto utili per ripetere comandi usati in precedenza. Alle frecce sono equivalenti, ancora come in Emacs, i tasti `^p` e `^n`.

Anche il tasto tabulatore (il quarto dal basso a sinistra della tastiera), che da ora in avanti denoteremo con *TAB*, è molto comodo, perché completa da solo i nomi di file o comandi, se ciò è univocamente possibile a partire dalla parte già battuta. Abituarsi a usarlo sistematicamente, si risparmia molto tempo e inoltre si evitano molti errori di battitura.

`comando &` fa in modo che il comando venga eseguito *in background*. Ciò significa che semplicemente la shell non aspetta che il processo chiamato dal comando termini ed è quindi immediatamente disponibile per nuovi comandi.

Come quasi sempre sotto Unix (e anche in C) bisogna distinguere tra minuscole e maiuscole, quindi i file *Alfa* e *alfa* sono diversi, così come i comandi *date* e *Date*.

Lavorare in background

Ogni programma sotto Unix viene eseguito nell'ambito di un *processo* e ciò vale anche per la shell con cui stiamo lavorando. Se da questa shell chiamiamo un altro programma, viene creato un processo (detto *figlio*) con il compito di eseguire quel programma, e normalmente il processo (detto *padre*) che sta eseguendo la shell aspetta che il figlio termini. In questo modo la shell non è disponibile fino a quando non si esce da quell'altro programma.

Espressioni regolari

Un'espressione regolare è una formula che descrive un insieme di parole. Usiamo qui la sintassi valida per il `grep`, molto simile comunque a quella più completa del Perl.

Una parola come espressione regolare corrisponde all'insieme di tutte le parole (nell'impostazione di default di `grep` queste parole sono le righe dei file considerati) che la contengono (ad esempio `alfa` è contenuta in `alfabeto` e `stalfano`, ma non in `stalfino`). `^alfa` indica invece che `alfa` si deve trovare all'inizio della riga, `alfa$` che si deve trovare alla fine. È come se `^` e `$` fossero due lettere invisibili che denotano inizio e fine della riga. Il carattere spazio viene trattato come gli altri, quindi con `a lfa` si trova `kappa lfa`, ma non `alfabeto`.

Il punto `.` denota un carattere qualsiasi, ma un asterisco `*` non può essere usato da solo, ma indica una ripetizione arbitraria (anche vuota) del carattere che lo precede. Quindi `a*` sta per le parole `a`, `aa`, `aaa`, ... , e anche per la parola vuota. Per quest'ultima ragione `alfa*ino` trova `alfino`. Per escludere la parola vuota si usa `+` al posto dell'asterisco. Ad esempio `+` indica almeno uno e possibilmente più spazi vuoti. Questa espressione regolare viene spesso usata per separare le parti di una riga. Per esempio `x *, +y` trova `alfax , ybeta`, ma non `alfax ,ybeta`. Il punto interrogativo `?` dopo un carattere indica che quel carattere può apparire una volta oppure mancare, quindi `alfa?ino` trova `alfino` e `alfaino`, ma non *alfaaino*.

Le parentesi quadre vengono utilizzate per indicare insiemi di caratteri oppure il complemento di un tale insieme. `[aeiou]` denota le vocali minuscole e `[^aeiou]` tutti i caratteri che non siano vocali minuscole. È il cappuccio `^` che qui indica il complemento. Quindi `r[aeio]ma` trova `rima` e `romano`, mentre `[Rr][aeio]ma` trova anche `Roma`. Si possono anche usare trattini per indicare insiemi di caratteri successivi naturali, ad esempio `[a-zP]` è l'insieme di tutte le lettere minuscole dell'alfabeto comune insieme alla `P` maiuscola, e `[A-Za-z0-9]` sono i cosiddetti caratteri alfanumerici, cioè le lettere dell'alfabeto insieme alle cifre. Per questo insieme si può usare l'abbreviazione `\w`, per il suo complemento `\W`.

La barra verticale `|` tra due espressioni regolari indica che almeno una delle due deve essere soddisfatta. Si possono usare le parentesi rotonde: `a|b|c` è la stessa cosa come `[abc]`, `r(oma|ume)no` trova `romano` e `rumeno`.

Per indicare i caratteri speciali `.`, `*`, `^` ecc. bisogna anteporgli `\`, ad esempio `\.` per indicare veramente un punto e non un carattere qualsiasi. Oltre a ciò nell'uso con `grep` bisogna anche impedire la confusione con i caratteri speciali della shell e quindi, se un'espressione regolare ne contiene, deve essere racchiusa tra apostrofi o virgolette.

Ciò vale in particolare quando l'espressione regolare contiene uno spazio, quindi bisogna usare

```
grep -i 'Clint Eastwood' cinema/*
```

La seconda parte di `man grep` tratta le espressioni regolari.

Molto di più si trova nel bel libro di Jeffrey Friedl.

grep

Il comando `grep`, il cui nome deriva da *get regular expression*, cerca nel proprio input o nei file i cui nomi gli vengono forniti come argomenti le righe che contengono un'espressione regolare. L'output avviene sullo schermo e può come al solito essere rediretto oppure impiegato in una pipeline.

Delle molte opzioni in pratica servono solo `grep -i` che fa in modo che la ricerca non distingua tra minuscole e maiuscole e `grep -s` che sopprime i talvolta fastidiosi messaggi d'errore causati dall'inesistenza di un file. Più importante è invece capire le espressioni regolari, anche se ancora più complete e sofisticate (e comode) sono quelle del Perl, presenti, con una sintassi un po' diversa, anche in Python. Esempio:

```
grep [QK]atar alfa
```

trova `Qatar` e `Katar` e

```
grep [Dd](ott|r)\. alfa
```

trova `Dott.`, `dott.`, `Dr.` e `dr.` nel file *alfa*.

telnet, ssh e ftp

L'uso di `telnet` o le sue varianti più sicure `ssh` ed `ssh2` è molto semplice. Dopo il comando

```
telnet altro.computer.com
```

appare una schermata di login come sul nostro PC (tipicamente in puro formato testo) e tutto si svolge come in un normale login.

Il comando `ftp` (talvolta `sftp` oppure `ncftp`) seguito dal nome di un computer remoto, serve per prelevare (con `get`) o deporre (con `put`) file su quell'altro computer. Per la navigazione sono disponibili ad esempio `cd` ed `ls`.

Bibliografia

J. Friedl: Mastering regular expressions. O'Reilly 1997.

S. Gundavaram e.a.: Linux Fedora - guida professionale. Apogeo 2004.

H. Hahn: Unix. McGraw-Hill 1995.

J. Peek/T. O'Reilly/M. Loukides: Unix power tools. O'Reilly 1993.

M. Stutz: Linux - guida pratica. Mondadori 2003. Costa solo 11 euro e contiene, in un formato molto leggibile, numerosi dettagli sui comandi di terminale sotto Linux.

Redirezione dell'output

Consideriamo prima il comando `cat`, il cui uso principale è quello di concatenare file. Vedremo adesso che ciò non richiede nessuna particolare "operazione di concatenazione", ma il semplice output di file che insieme alla possibilità di *redirezione* tipica dell'ambiente Unix produce la concatenazione.

Infatti l'output di `cat alfa` è il contenuto del file *alfa*, l'output di `cat alfa beta gamma` è il contenuto unito, nell'ordine, di *alfa*, *beta* e *gamma*. In questo caso l'output viene visualizzato sullo schermo. Ma nella filosofia Unix non si distingue (fino a un certo punto) tra file normali, schermo, stampante e altre periferiche. Talvolta sotto Unix tutti questi oggetti vengono chiamati *file* oppure *data streams*. Noi useremo il nome file quasi sempre solo per denotare file normali o cartella. Un modo intuitivo (e piuttosto corretto) è quello di vedere i data streams come *vie di comunicazione*. Un programma (o meglio un processo) riceve il suo input da una certa via e manda l'output su qualche altra (o la stessa) via, e in genere dal punto di vista del programma è indifferente quale sia il mittente o il destinatario dall'altro capo della via.

Nella shell (e in modo analogo nel C) sono definite tre vie standard: lo *standard input* (abbreviato *stdin*, tipicamente la tastiera), lo *standard output* (abbreviato *stdout*, tipicamente lo schermo) e lo *standard error* (abbreviato *stderr*, in genere lo schermo, ma in un certo senso indipendentemente dallo standard output). Nell'impostazione di default un programma riceve il suo input dallo standard input, manda il suo output allo standard output e i messaggi d'errore allo standard error. Ma con semplici variazioni dei comandi (uso di `<`, `>` e `|`) si possono *redirigere* input e output su altre vie - e per il programma non fa alcuna differenza, non sa nemmeno che dall'altra parte a mandare o a ricevere i dati si trova adesso un file normale o una stampante!

`comando` sia un comando semplice o composto (cioè eventualmente con argomenti e opzioni) e *delta* un file normale o un nome possibile per un file che ancora non esiste. Allora `comando > delta` fa in modo che l'output di questo comando viene inviato al file *delta* (cancellandone il contenuto se questo file esiste, creando invece un file con questo nome in caso contrario). Quindi `ls -l > delta` fa in modo che il catalogo (in formato lungo) della cartella in cui ci troviamo non viene visualizzato sullo schermo, ma scritto nel file *delta*. Esiste anche la possibilità di usare `>>` per l'aggiunta senza cancellazione del contenuto eventualmente preesistente.

In questo modo si spiega anche perché `cat alfa beta gamma > delta` scrive in *delta* l'unione dei primi tre file: normalmente l'output - nel caso di `cat` semplicemente il contenuto degli argomenti - andrebbe sullo schermo, ma la redirezione `> delta` fa in modo che venga invece scritto in *delta*. Ricordarsi che `cat alfa > alfa` cancella *alfa*.

In modo simile si può usare il comando `man ls > alfa` per scrivere il manuale di `ls` su un file *alfa*, il quale può essere conservato oppure successivamente stampato.

Similmente si rivela utile ad esempio

```
find Dati -name '[Gg]eol*' -print > alfa
```

Redirezione dell'input

Si può anche redirigere l'input: `comando < alfa` fa in modo che un comando che di default aspetta un input da tastiera lo riceva invece dal file *alfa*. I due tipi di redirezione possono essere combinati, ad esempio `comando < alfa > beta` (l'ordine in cui appaiono `< e >` non è importante qui) fa in modo che il comando riceva l'input dal file *alfa* e l'output dal file *beta*.

Se si batte `cat` da solo senza argomenti, il programma aspetta input dalla tastiera e lo riscrive sullo schermo (si termina con `^c`). Ciò mostra che `cat` tratta il suo input nello stesso modo come un argomento e questo spiega perché `cat alfa` e `cat < alfa` hanno lo stesso effetto - la visualizzazione di *alfa* sullo schermo. Ma il meccanismo è completamente diverso: nel primo caso *alfa* è argomento del comando `cat`, nel secondo caso *alfa* diventa l'input. Provare `cat > alfa` e spiegare cosa succede.

Direzione dei messaggi d'errore

Ogni volta che un file viene aperto da un programma riceve un numero che lo identifica univocamente tra i file aperti da quel programma (diciamo programma anche se in verità si dovrebbe parlare di *processi*). In inglese questo numero si chiama *file descriptor*. I file *standard input*, *standard output* e *standard error* sono sempre aperti e hanno sempre, nell'ordine indicato, i numeri 0, 1 e 2.

Si usano questi numeri nella redirezione dei messaggi d'errore: con il `comando 2> alfa` si fa in modo che i messaggi d'errore vengano scritti nel file *alfa*. In questo caso in `2>` non ci deve essere uno spazio (provare con e senza spazio per il comando `ls -j` che contiene l'opzione non prevista `-j` che provoca un errore).

Per inviare nello stesso file *alfa* sia l'output che il messaggio d'errore si usa `comando >& alfa`.

Pipelines della shell

Le *pipelines* (o *pipes*) della shell funzionano in modo molto simile alle redirezioni, anche se il meccanismo interno è un po' diverso. `comando1 | comando2` fa in modo che la via di output del primo comando viene unita alla via di input del secondo, e ciò implica che l'output del primo comando viene elaborato dal secondo. Spesso si usa ad esempio `ls -l | less` per poter leggere il catalogo di una cartella molto grande con `less`. Lo si può anche stampare con `ls -l | lpr -s`. Anche il manuale di un comando può essere stampato in questo modo con `man comando | lpr -s`.

Per mandare un messaggio di posta elettronica in genere useremo `pine`, ma in verità ci sono comandi più elementari per farlo, ad esempio con `mail rossi < alfa` il contenuto del file *alfa* viene inviato all'utente *rossi* (sul nostro stesso computer, altrimenti invece di *rossi* bisogna ad esempio scrivere *rossi@student.unife.it*). Si può anche indicare il soggetto, ad esempio `mail -s Prova rossi < alfa`. Ciò mostra che `mail` prende il corpo del messaggio come input. Possiamo quindi mandare come mail anche l'output di un comando, ad esempio il catalogo di una cartella con `ls -l | mail rossi`.

Le pipelines possono essere combinate, `ls | grep [Rr]ob | lpr -s` ad esempio fa in modo che venga stampato l'elenco di tutti i file (nella cartella .) il cui nome contiene *rob* o *Rob*.

Di nuovo un esempio con `find`:

```
find -name 'matem*' | less
```

permette una può comoda lettura del risultato della ricerca tramite `less`.

Il file .bashrc

Sotto Mepis, ad ogni partenza della shell di login (quindi anche ogni volta che apriamo un terminale) viene eseguito il file *.bashrc*, che può contenere ad esempio le seguenti istruzioni:

```
[ -f /etc/bashrc ] && . /etc/bashrc
[ -f /etc/profile ] && . /etc/profile

export KDEDIR=/usr
export QTDIR=/usr/share/qt3

alias c='. ~/Software/Menuprogramm'
alias ecp='python -u ~/Software/ecp'
alias rm='rm -i'

PATH="/bin:/usr/bin"
PATH="$PATH:/usr/local/bin:/sbin:"
PATH="$PATH:/usr/sbin:/usr/X11R6/bin:"
declare -x PATH="$PATH:~/Software:."

declare -x PAGER='less'
declare -x PS1=':) '
clear; who; echo; date; umask 022; c
```

.bashrc è uno shell script e in pratica i comandi in esso contenuti potrebbero essere anche battuti uno per uno dalla tastiera dopo il login.

Tra le dichiarazioni la più importante è quella della variabile *PATH*, che indica, in forma di una lista ordinata i cui componenti sono separati da *:*, le cartelle in cui la shell cerca i programmi da eseguire. Quindi quando si chiama il programma *alfa*, la shell guarda prima nella cartella */bin*, se questa contiene un file *alfa* e, se è eseguibile, lo esegue. Se non lo trova, cerca in */usr/local/bin* e così via. L'ultima cartella nel nostro PATH è . (verificare sopra), ciò significa che si possono eseguire programmi dalla cartella corrente, se non ci sono programmi con lo stesso nome in altre cartella del PATH. Per ragioni di sicurezza la . dovrebbe sempre essere aggiunta per ultima al PATH. La variabile *PAGER* indica il programma che viene usato per leggere le pagine di `man`, quindi nella nostra imppostazione viene utilizzato `less`. *PS1* è il prompt della shell; noi abbiamo scelto *:)* , ma si potrebbe anche fare in modo che ad esempio nel prompt venga sempre indicata la cartella in cui si trova.

L'impostazione `umask 022` significa che dall'impostazione 777 risp. 666 dei diritti d'accesso per le cartelle risp. per i file viene sottratto 022 e che quindi le cartelle vengono create con i diritti 755 (`rwxr-xr-x`) e i file normali con 644 (`rw-r--r--`); cfr. pag. 8.

Menuprogramm è uno script di shell per cui abbiamo previsto l'alias `c` e che fa in modo che, se *alfa* è un file normale, `c alfa` equivale a `less alfa`, mentre per una cartella equivale a `cd alfa; ls`.

Ogni utente può modificare a piacere il proprio *.bashrc*!

Diritti d'accesso

Con `ls -l` viene visualizzato per esempio

```
total 40
drwxr-xr-x  2  rossi  users  28672  Mar   8   2000  RPMS
-rw-r--r--  1  root   root     173  Mar   9   2000  TRANS.TBL
dr-xr-xr-x  2  rossi  users   4096  May  26  10:30  base
drwxr-xr-x  2 root    root   49152  Jun   7  23:12  trand
drwx------  4 miller  users   4096  Jul   9  11:44  miller
drwx------  2 munn    users   4096  May   4  20:23  mathem
drwxr-xr-x  2 root    root    4096  Mar  18  23:32  onthego
-rw-r--r--  1  rossi  users     43  Jan  19   1999  prova
-rwxr-xr-x  6  rossi  users   7234  Mar   8   2000  programma
```

La lettera `d` all'inizio della seconda riga significa che si tratta di una cartella, mentre l'ultima colonna indica il nome (`RPMS` in questo caso). Il trattino iniziale nella terza e quinta riga significa che si tratta di file normali. Nella terza colonna sta il nome del *proprietario* del file o della cartella, nella quarta il nome del *gruppo*. Seguono indicazioni sulle dimensioni e la data dell'ultima modifica. Il significato della seconda colonna verrà spiegato fra poco.

La prima colonna consiste di 10 lettere, di cui la prima, come visto, indica il tipo del file. Le 9 lettere che seguono contengono i *diritti d'accesso*. Vanno divise in tre triple, la prima per il proprietario, la seconda per il gruppo, la terza per tutti gli altri utenti. Nel caso più semplice ogni tripla è della forma `abc`, dove `a` può essere `r` (diritto di lettura - *read*) o `-` (diritto di lettura negato), `b` può essere `w` (diritto di scrittura - *write*) o `-` (diritto di scrittura negato), `c` può essere `x` (diritto di esecuzione - *execution*) o `-` (diritto di esecuzione negato).

Per i file normali i concetti di lettura, scrittura (modifica) e esecuzione hanno il significato che intuitivamente ci si aspetta. Quindi il file `TRANS.TBL.` i cui diritti d'accesso sono `rw-r--r--` può essere letto è modificato dall'utente root, il gruppo root e gli altri utenti lo possono solo leggere. Il file `programma` nell'ultima riga ha i diritti d'accesso `rwxr-xr-x` e quindi può essere letto e eseguito da tutti, ma modificato solo dal propietario. Osserviamo che la cancellazione di un file non viene considerata una modifica del file, ma della cartella che lo contiene e quindi chi ha il diritto (`w`) di modifica di una cartella può cancellare in essa anche quei file per i quali non ha il diritto di scrittura.

Per le cartelle i diritti d'accesso vanno interpretati in modo leggermente diverso. Si tenga conto che una cartella non è una raccolta di file, ma un file a sua volta che contiene un elenco di file. Il diritto di lettura (`r`) significa qui che questo elenco può essere letto (ad esempio da `ls`). Il diritto di esecuzione (`x`) per le cartelle significa invece che sono accessibili al comando `cd`. Questo è anche necessario per la lettura, quindi se un tipo di utente deve poter leggere il contenuto di una cartella, bisogna assegnargli il diritto `r-x`.

Come osservato sopra, per le directory il diritto di modifica (`w`) ha il significato che possono essere creati o cancellati file nella cartella, *anche quelli di un altro proprietario*. In tal caso normalmente `rm` (che si accorge comunque del fatto che l'utente sta tentando di cancellare un file che non gli appartiene) chiede conferma, a meno che non si stia usando l'opzione di cancellazione forzata `rm -f`. Si può fare però in modo che in una cartella *alfa* in cui più utenti hanno il diritto di scrittura, solo il proprietario di un file lo possa cancellare, con il comando `chmod +t alfa`. Si vede che allora (se prima erano `rwxr-xr-x`) i diritti d'accesso di *alfa* vengono dati come `rwxr-xr-t`. Il `t` qui viene detto *sticky bit*.

Tipicamente una cartella in cui tutti possono entrare, ma solo il proprietario può creare e cancellare dei file, avrà i diritti d'accesso `rwxr-xr-x`, mentre la cartella di entrata di un utente, il cui contenuto non deve essere visto dagli altri, ha tipicamente i diritti `rwx------`.

chown e chgrp

`chown P alfa` fa in modo che `P` diventi proprietario del file *alfa*. Naturalmente chi esegue questo comando deve avere il diritto di farlo, ad esempio essere root oppure il proprietario del file. Il nome del proprietario viene indicato prima del nome del file, questo permette di usare lo stesso comando per cambiare il proprietario per più file, ad esempio `chown P alfa beta gamma`.

Per cambiare il gruppo si usa `chgrp G alfa beta`. Spesso devono essere cambiati sia il proprietario che il gruppo, allora si può usare la forma abbreviata `chown P.G alfa beta gamma`.

chmod

Questo comando viene usato per l'assegnazione dei diritti d'accesso. La prima (e più generale) forma del comando è `chmod UOD alfa`, dove *alfa* è il nome di un file (anche qui possono essere indicati più file), mentre U sta per utenti, O per operazione, D per diritti. D può essere `r`, `w`, `x` oppure una combinazione di questi tre e può anche mancare (nessun diritto). O può essere `=` (proprio i diritti indicati), `+` (oltre ai diritti già posseduti anche quelli indicati), `-` (i diritti posseduti meno quelli indicati). Nella specifica degli utenti la scelta delle lettere non è felice e causa di frequente confusione: U può essere `u` (proprietario), `g` (gruppo), `o` (altri), una combinazione di questi oppure mancare (tutti i tipi di utente). Ci può essere anche più di un UOD, allora gli UOD vanno separati con virgole. Alcuni esempi (sono sempre da aggiungere a destra i nomi dei file); `.` nel risultato significa nessun cambiamento rispetto alla situazione di partenza:

```
chmod uo=rw →  rw-...rw-
chmod =rx   →  r-xr-xr-x
chmod o=x   →  ......--x
chmod go=   →  ...------
chmod +x    →  ..x..x..x
chmod ug+rw →  rw.rw....
chmod o-wx  →  ......r--
chmod go-w  →  ....-..-.
chmod u=rwx,g=rx,o=r → rwxr-xr--
```

Link e link simbolici

Le informazioni riguardanti un file sono raccolte nel suo *inode* o *file header*. Il termine inode è un'abbreviazione di *index node*. Gli inode sono numerati; il numero dell'inode di un file si chiama *indice* (*index number*) del file e può essere visualizzato con `ls -i` (per una cartella si ottengono gli indici di tutti i file contenuti in essa). `ls -i` senza argomento visualizza gli indici dei file della cartella in cui ci troviamo. Gli indici in partizioni diverse sono indipendenti, mentre all'interno di una partizione l'indice determina univocamente il file.

Un file può avere più nomi (anche nella stessa cartella), e bisogna distinguere il file fisico dai nomi con i quali è registrato nelle cartella. In inglese ogni registrazione (o nome) di un file viene detta *link*. L'inode riguarda il file fisico, a nomi diversi dello stesso file corrisponde quindi sempre lo stesso inode e perciò anche lo stesso indice.

Con il comando `ln alfa beta` si crea una nuova registrazione (un nuovo link) *beta* del file registrato come *alfa*. La nuova registrazione si può anche trovare in un'altra cartella (in questo caso si utilizzerà per esempio il comando `alfa gammabeta/`, ma non in un'altra partizione della memoria periferica, proprio perché l'indice è unico solo all'interno di una partizione.

Se un file è registrato come *alfa* e *beta*, ogni modifica di *alfa* è automaticamente anche una modifica di *beta*, perché fisicamente il file è sempre lo stesso. Il comando `rm alfa` invece si riferisce solo alla registrazione *alfa*, cancella cioè questa dall'elenco delle registrazioni del file. Soltanto con la cancellazione dell'ultima registrazione di un file questo viene eliminato dal file system. Il numero delle registrazioni di un file fisico è indicato alla destra dei diritti d'accesso dal comando `ls -l`.

Il comando `ln alfa beta` è permesso solo se si rimane nella stessa partizione. Link a cartelle possono essere creati solo dall'utente root e normalmente vengono evitati. È però possibile creare dei *link simbolici* anche di cartelle e superando i confini tra le partizioni. Si usa il comando `ln -s alfa beta`.

Dopo il comando `ln -s alfa beta` il significato di *alfa* e *beta* non è simmetrico. Ad esempio, se *alfa* è l'unico link (normale) rimasto di un file, allora il comando `rm alfa` lo cancellerà fisicamente, anche se esiste il link simbolico *beta* che a questo punto diventa un riferimento vuoto e inutile.

Si può invece cancellare con `rm beta` il riferimento simbolico *beta* senza influenzare minimamente *alfa*.

Per quasi tutti gli altri comandi i link simbolici funzionano invece come i link: Se si tratta di cartelle, possono essere aperte, se si tratta di file, ci si può scrivere, ecc.

Se *alfa* e *beta* sono due link dello stesso file fisico, essi corrispondono allo stesso indice. Un link simbolico ha invece un indice diverso, infatti è un file di riferimento a quel file fisico. Se ad esempio creiamo con `ln alfa beta` un link *beta* di *alfa* e con `ln -s alfa gamma` un link simbolico, il comando `ls -i alfa beta gamma` darà un output simile a questo:

```
77933 alfa      77933 beta      77947 gamma
```

III. PYTHON

Installazione

Python è in questo momento forse il miglior linguaggio di programmazione: per la facilità di apprendimento e di utilizzo, per le caratteristiche di linguaggio ad altissimo livello che realizza i concetti sia della programmazione funzionale che della programmazione orientata agli oggetti, per il recente perfezionamento della libreria per la programmazione insiemistica, per il supporto da parte di numerosi programmatori, per l'ottima documentazione disponibile in rete e la ricerca riuscita di meccanismi di leggibilità, per la grafica con Tkinter (basata su Tcl/Tk), per la semplicità dell'aggancio ad altri linguaggi.

Nelle distribuzioni di Linux Python è sempre presente; per Windows installare *Enthought Python 2.4.3* dal sito del corso.

Fahrenheit e Celsius

Scriviamo due semplici funzioni per la conversione tra Fahrenheit e Celsius. Se f è la temperatura espressa in gradi Fahrenheit e c la stessa temperatura espressa in gradi Celsius, allora vale la relazione

$$c = (f - 32)/1.8$$

e quindi $f = 1.8c + 32$.

```
def celsiodafahrenheit (f):
  return (f-32)/1.8

def fahrenheitdacelsio (c):
  return 1.8*c+32
```

Queste funzioni vengono utilizzate nel modo seguente.

```
cdf=celsiodafahrenheit
fdc=fahrenheitdacelsio

for n in (86,95,104): print n,cdf(n)

print

for n in (20,30,35,40): print n,fdc(n)
```

Otteniamo l'output

```
86 30.0
95 35.0
104 40.0

20 68.0
30 86.0
35 95.0
40 104.0
```

Commenti

Se una riga contiene, al di fuori di una stringa, il carattere `#`, tutto il resto della riga è considerato un *commento*, compreso il carattere `#` stesso.

Molti altri linguaggi interpretati (Perl, R, la shell di Unix) usano questo simbolo per i commenti. In C e C++ una funzione analoga è svolta dalla sequenza `//`.

Come eseguire un programma

In una cartella *Python* (oppure, per progetti più importanti, in un'apposita sottocartella per quel progetto) creiamo i file sorgente come file di testo puro con l'estensione *.py*, utilizzando l'editor incorporato di Python. Con lo stesso editor, piuttosto comodo, scriviamo anche, usando l'estensione *.r*, le sorgenti in R che vogliamo affiancare ai programmi in Python. I programmi possono essere eseguiti mediante il tasto *F5* nella finestra dell'editor. Soprattutto in fase di sviluppo sceglieremo questa modalità di esecuzione, perché così vengono visualizzati anche i messaggi d'errore.

Possiamo creare i file sorgente anche con il nostro editor ECP disponibile sul sito del corso.

Successivamente i programmi possono essere eseguiti anche tramite il clic sull'icona del file oppure, in un terminale (prompt dei comandi) e se il file si chiama *alfa.py*, con il comando `python alfa.py`.

Teoricamente i programmi possono essere scritti con un qualsiasi editor che crea file in formato testo puro, ad esempio il *Blocco note* di Windows, ma, quando non usiamo ECP, preferiamo utilizzare l'editor di Python per una più agevole correzione degli errori, per l'indentazione automatica e perché prevede la possibilità di usare combinazioni di tasti più comode di quelle disponibili per il Blocco note.

Elenchiamo alcune combinazioni di tasti:

Ctrl Q	uscire
F5	esecuzione
Ctrl N	nuovo file
Ctrl O	aprire un file
Alt G	trova una riga
Ctrl A	seleziona tutto
Ctrl C	copia il testo selezionato
Ctrl V	incolla
Ctrl X	cancella il testo selezionato
Ctrl K	cancella il resto della riga
Inizio	inizio della riga
Fine	fine della riga
Ctrl E	fine della riga
Alt P	lista dei comandi dati: indietro
Alt N	lista dei comandi dati: avanti

Per importare le istruzioni contenute in un file `beta.py` si usa il comando `import beta`, tralasciando l'estensione. Con lo stesso comando si importano anche i moduli delle librerie incorporate o prelevate in rete:

```
import os, math, scipy
```

---

os.system

Il comando `os.system` permette di eseguire, dall'interno di un programma in Python, comandi di Windows (o comunque del sistema operativo), ad esempio

```
import os

os.system('dir > catalogo')
```

per scrivere l'elenco dei file nella cartella attiva nel file *catalogo*.

`os.system` apparentemente funziona solo all'interno dei programmi, non nell'editor di Python.

Primi esempi in Python

```
a=range(5,13,2)
print a
# output: [5, 7, 9, 11]

a=range(5,13)
print a
# output: [5, 6, 7, 8, 9, 10, 11, 12]

a=range(11)
print a
# output:
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Si noti che il limite destro non viene raggiunto.

```
a=xrange(5,13,2)
print a
# output: xrange(5, 13, 2)

for x in a: print x,
# output: 5 7 9 11
```

La differenza tra `range` e `xrange` è questa: Mentre `range(1000000)` genera una lista di un milione di elementi, `xrange(1000000)` è un *iteratore* (pagina 11) che crea questi elementi uno allo volta in ogni passaggio di un ciclo in cui il comando viene utilizzato.

Si noti il doppio punto (`:`) alla fine del comando `for`.

Sono possibili assegnazioni, confronti e scambi simultanei:

```
if 3<5<9: print 'o.k.'
# output: o.k.

a=b=c=4
for x in [a,b,c]: print x,
print
# output: 4 4 4

a=5; b=3; a,b=b,a; print [a,b]
# output: [3, 5]
```

Vettori associativi (dizionari o tabelle di hash) vengono definiti nel modo seguente:

```
latino = {'casa': 'domus',
  'villaggio': 'pagus',
  'nave':'navis', 'campo':'ager'}
voci=sorted(latino.keys())
for x in voci:
  print '%-9s = %s' %(x,latino[x])
# output:
# campo     = ager
# casa      = domus
# nave      = navis
# villaggio = pagus
```

Stringhe sono racchiuse tra apici o virgolette, stringhe su più di una riga tra triplici apici o virgolette:

```
print 'Carlo era bravissimo.'
# output: Carlo era bravissimo.

print "Carlo e' bravissimo."
# output: Carlo e' bravissimo.

print '''Stringhe a piu' righe si
usano talvolta nei commenti.'''
# output:
# Stringhe a piu' righe si
# usano talvolta nei commenti.
```

Funzioni in Python

```
def f(x): return 2*x+1

def g(x):
  if (x>0): return x
  else: return -x

for x in xrange(0,10): print f(x),
print
# output: 1 3 5 7 9 11 13 15 17 19

for x in xrange(-5,5): print g(x),
print
# output: 5 4 3 2 1 0 1 2 3 4
```

A differenza da R e Lisp, il `return` è obbligatorio.

La virgola alla fine di un comando `print` fa in modo che la stampa continui sulla stessa riga. Come si vede nella definizione di g, Python utilizza l'indentazione per strutturare il programma. Anche le istruzioni `if` ed `else` richiedono il doppio punto.

Una funzione di due variabili:

```
import math

def raggio (x,y):
  return math.sqrt(x**2+y**2)

print raggio(2,3)
# output: 3.60555127546

print math.sqrt(13)
# output: 3.60555127546
```

Funzioni possono essere non solo argomenti, ma anche risultati di altre funzioni:

```
def sommax (f,g,x): return f(x)+g(x)

def compx (f,g,x): return f(g(x))

def u(x): return x**2

def v(x): return 4*x+1

print sommax(u,v,5)
# output: 46

print compx(u,v,5)
# output: 441
```

Possiamo però anche definire

```
def somma (f,g):
  def s(x): return f(x)+g(x)
  return s

def comp (f,g):
  def c(x): return f(g(x))
  return c

def u(x): return x**2

def v(x): return 4*x+1

print somma(u,v)(5)
# output: 46

print comp(u,v)(5)
# output: 441
```

Funzioni con un numero variabile di argomenti: Se una funzione `f` è dichiarata nella forma `def f(x,y,*a):`, l'espressione `f(2,4,5,7,10,8)` viene calcolata in modo che gli ultimi argomenti vengano riuniti in una tupla `(5,7,10,8)` che nel corpo del programma è utilizzata come se questa tupla fosse a:

```
def somma (*a):
  s=0
  for x in a: s+=x
  return s

print somma(1,2,3,10,5)
# output: 21
```

Lo schema di Horner per il calcolo dei valori $f(\alpha)$ di un polinomio $f = a_0x^n + \ldots + a_n$ consiste nella ricorsione

$$b_{-1} = 0$$
$$b_k = \alpha b_{k-1} + a_k$$

per $k = 0, \ldots, n$. Possiamo quindi definire

```
def horner (alfa,*a):
  alfa=float(alfa); b=0
  for t in a: b=b*alfa+t
  return b

print horner(10,6,2,0,8)
# output: 6208.0
```

Analizzare i seguenti esempi; per `apply` vedere pagina 21.

```
def f(x,a=1): print a+x

def g(*u,**s): apply(f,u,s)

g(7,a=3) # 10

def comp (*f):
  n=len(f)
  if n==0: return lambda x: x
  if n==1: return f[0]
  def g (x):
    return f[0](apply(comp,f[1:])(x))
  return g

def u (x): return x+2
def v (x): return x*x
def w (x): return 3*x+1

f=comp(u,v,w); print f(3)
# 102
```

L'istruzione def

Per il suo formato particolare si sarebbe indotti a credere che la definizione di una funzione mediante un'istruzione

```
def f (x): ...
```

assomigli nel significato alla definizione di una funzione in C e che quindi le funzioni vengano create prima dell'esecuzione del programma, cosicché in particolare una tale definizione non possa essere ripetuta con lo stesso nome. Non è così invece: Ogni `def` è eseguito dall'interprete nel punto in cui si trova nel programma ed è piuttosto equivalente a un'istruzione `f = function (x) ...` di R. Non soltanto i `def` possono essere annidati, come abbiamo visto negli ultimi esempi, ma lo stesso nome può essere riutilizzato in un altro `def`:

```
def f(x): print x+3
f(7) # 10

def f(x,y): print x+y
f(3,9) # 12
```

Liste

Liste sono successioni finite modificabili di elementi non necessariamente dello stesso tipo. Python fornisce numerose funzioni per liste che non esistono per le tuple. Come le tuple anche le liste possono essere annidate, la lunghezza di una lista `v` la si ottiene con `len(v)`, l'`i`-esimo elemento è `v[i]`. A differenza dalle tuple, la lista che consiste solo di un singolo elemento `x` è denotata con `[x]`.

```
v=[1,2,5,8,7]

for i in xrange(5): print v[i],
print
# 1 2 5 8 7

for x in v: print x,
print
# 1 2 5 8 7

a=[1,2,3]; b=[4,5,6]
v=[a,b]

print v
# [[1, 2, 3], [4, 5, 6]]

for x in v: print x
# [1, 2, 3]
# [4, 5, 6]

print len(v)
# 2
```

Sequenze

Successioni finite in Python vengono dette *sequenze*, di cui i tipi più importanti sono liste, tuple e stringhe. Tuple e stringhe sono sequenze non modificabili. Esistono alcune operazioni comuni a tutte le sequenze che adesso elenchiamo.

`a` e `b` siano sequenze:

`x in a`	Vero, se `x` coincide con un elemento di `a`.
`x not in a`	Vero, se `x` non coincide con nessun elemento di `a`.
`a + b`	Concatenazione di `a` e `b`.
`a * k`	Concatenazione di `k` copie di `a`.
`a[i]`	`i`-esimo elemento di `a`.
`a[-1]`	Ultimo elemento di `a`.
`a[i:j]`	Sequenza che consiste degli elementi `a[i]`, ..., `a[j-1]` di `a`.
`a[:]`	Copia di `a`.
`len(a)`	Lunghezza di `a`.
`min(a)`	Più piccolo elemento di `a`. Per elementi non numerici viene utilizzato l'ordine alfabetico.
`max(a)`	Più grande elemento di `a`.
`sorted(a)`	*Lista* che contiene gli elementi di `a` in ordine crescente. Si noti che il risultato è sempre una lista, anche quando `a` è una stringa o una tupla.
`reversed(a)`	*iteratore* che corrisponde agli elementi di `a` elencati in ordine invertito.
`list(a)`	Converte la sequenza in una lista con gli stessi elementi.

Come abbiamo visto, l'espressione `x in a` può apparire anche in un ciclo `for`.

Iteratori

Iteratori sono oggetti che forniscono uno dopo l'altro tutti gli elementi di una sequenza senza creare questa sequenza in memoria. Ad esempio sono iteratori gli oggetti che vengono creati tramite l'istruzione `xrange`.

La funzione `list` può essere applicata anche agli iteratori, per cui per generare la lista `b` che si ottiene da una sequenza `a` invertendo l'ordine in cui sono elencati i suoi elementi, possiamo utilizzare `b=list(reversed(a))`.

Esempi:

```
if 'a' in 'nave': print "Si'."
# Si'.

if 7 in [3,5,1,7,10]: print "Si'."
# "Si'."

if 2 in [3,5,1,7,10]: print "Si'."
else: print "No."
# No.

a=[1,2,3,4]; b=[5,6,7,8,9]
print a+b
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

a='Mario'; b='Rossi'
print a+' '+b
# Mario Rossi

a=[1,2,3]*4
print a
# [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

a='==.==*3
print a
# ==.====.====.==

a=[10,11,12,13,14,15,16,17,18,19,20]
for i in xrange(0,11,3): print a[i],
print
# 10 13 16 19

a='01234567'
print a[2:5]
# 234

a=[3,5,3,1,0,1,2,1]
b=sorted(a)
print b
# [0, 1, 1, 1, 2, 3, 3, 5]
# Non funziona nella 2.3.5.

a=[1,2,3,4,5]
c=reversed(a)
print c
# <listreverseiterator object at ...>
# L'indirizzo dopo at cambia ogni volta.
# Non funziona nella 2.3.5.

d=list(reversed(a))
print d
# [5, 4, 3, 2, 1]
# Non funziona nella 2.3.5.

a='MARIO'
print list(a)
# ['M', 'A', 'R', 'I', 'O']
```

„Python stands out as the language of choice for scripting in computational science because of its very clean syntax, rich modularization features, good support for numerical computing, and rapidly growing popularity.“ (Langtangen, pag. v)

Generatori

Generatori sono oggetti simili a iteratori, ma più generali, perché possono essere creati mediante apposite funzioni per cui un generatore può generare successioni anche piuttosto complicate.

Il modo più semplice per creare un generatore è, nella sintassi molto simile al map implicito (pagina 17): dobbiamo soltanto sostituire le parentesi quadre con parentesi tonde:

```
a=[n*n for n in xrange(8)]
print a
# [0, 1, 4, 9, 16, 25, 36, 49]

a=(n*n for n in xrange(8))
print a
# <generator object at 0x596e4c>

for k in xrange(4): print a.next(),
# 0 1 4 9
```

Per creare un generatore possiamo anche definire una funzione in cui al posto di un `return` appare un'istruzione `yield`. Ogni volta che il generatore viene invocato, mediante il metodo `next` o nei passaggi di un ciclo, viene fornito il prossimo elemento della successione; l'esecuzione dell'algoritmo viene poi *fermata* fino alla prossima invocazione. Con alcuni esempi il meccanismo diventa più comprensibile. Definiamo prima ancora un generatore di numeri quadratici:

```
def quadrati ():
  n=0
  while 1:
    yield n*n; n+=1

q=quadrati()
for k in xrange(8): print q.next(),
# 0 1 4 9 16 25 36 49
```

In questo caso abbiamo generato addirittura *una successione infinita!* Infatti ogni chiamata `q.next()` fornirebbe un nuovo quadrato. Possiamo anche creare una successione infinita di fattoriali:

```
def genfatt ():
  f=1; i=2
  while 1: yield f; f*=i; i+=1

fattoriali=genfatt()

for k in xrange(8):
  print fattoriali.next(),
# 1 2 6 24 120 720 5040 40320
```

Spesso però si vorrebbe una successione finita, simile a un `xrange`, da usare in un ciclo `for`. Allora possiamo modificare l'esempio nel modo seguente:

```
def quadrati (n):
  for k in xrange(n): yield k*k

q=quadrati(8)
for x in q: print x,
# 0 1 4 9 16 25 36 49
```

In modo simile possiamo creare un generatore per i numeri di Fibonacci, imitando la funzione `fib` a pagina 19:

```
def generafib (n):
  a=0; b=1
  for k in xrange(n):
    a,b=a+b,a
    yield a

fib=generafib(9)
for x in fib: print x,
# 1 1 2 3 5 8 13 21 34
```

Se la successione contenuta in un generatore è finita, può essere trasformata in una lista:

```
fib=generafib(9)
print list(fib)
# [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Un `for` eseguito su un generatore lo svuota:

```
fib=generafib(9)
for x in fib: print x,
print list(fib)
# []

fib=generafib(9)
for k in xrange(5):
  print fib.next(),
# 1 1 2 3 5
print
print list(fib)
# [8, 13, 21, 34]
```

Successioni infinite possono talvolta sostituire variabili globali. Assumiamo ad esempio che in un programma interattivo ogni volta che l'utente lo richieda vorremmo che venga creato un nuovo oggetto, come un elemento grafico o una nuova scheda per un libro che viene aggiunta al catalogo di una biblioteca, con un unico numero che lo identifica. Invece di mantenere una variabile globale che ogni volta viene aumentata, possiamo definire un generatore all'incirca in questo modo:

```
def generanumeri ():
  n=0
  while 1: yield n; n+=1

numeri=generanumeri

def schemadiprogramma ():
  while 1:
    e=evento()
    if e.azione=='fine': break
    if e.azione=='inserimento':
      n=numeri.next()
      inserisci(e.libro,n)
```

Alcuni operatori per sequenze, ad esempio `max`, possono essere applicati anche a generatori. Dopo l'uso però il generatore risulta vuoto:

```
cos=math.cos
g=(cos(x) for x in (0.9,0.1,0.4,0.4))
print max(g)
# 0.995004165278
print list(g)
# []
```

pass

`pass` è l'istruzione che non effettua nessuna operazione. Esempi si trovano alle pagg. 27, 35, 38, 40.

Funzioni per liste

Per le liste sono disponibili alcune funzioni speciali che non possono essere utilizzate per tuple o (con eccezione di `index` e `count`) per stringhe.

`v` sia una lista.

`v.append(x)`	`x` viene aggiunto alla fine di `v`. Equivalente a `v=v+[x]`, ma più veloce.
`v.extend(w)`	Aggiunge la lista `w` a `v`. Equivalente a `v=v+w`, ma più veloce.
`v.count(x)`	Il risultato indica quante volte `x` appare in `v`.
`v.index(x)`	Indice della prima posizione in cui `x` appare in `v`; provoca un errore se `x` non è elemento di `v`.
`v.insert(i,x)`	Inserisce l'elemento `x` come `i`-esimo elemento della lista.
`v.remove(x)`	Elimina `x` nella sua prima apparizione in `v`; provoca un errore se `x` non è elemento di `v`.
`v.pop()`	Toglie dalla lista il suo ultimo elemento che restituisce come risultato; errore, se il comando viene applicato alla lista vuota.
`v.sort()`	Ordina la lista che viene modificata.
`v.reverse()`	Inverte l'ordine degli elementi in `v`. La lista viene modificata.

Esempi:

```
v=[1,2,3,4,5,6]; v.append(7)
print v
# [1, 2, 3, 4, 5, 6, 7]

v=[2,8,2,7,2,2,3,3,5,2]
print v.count(2)
# 5
print v.count(77)
# 0

print v.index(7)
# 3

v=[10,11,12,13,14,15,16]
v.insert(4,99); print v
# [10, 11, 12, 13, 99, 14, 15, 16]

v=[2,3,8,3,7,6,3,9]
v.remove(3); print v
# [2, 8, 3, 7, 6, 3,9]

v.pop()
print v
# [2, 8, 3, 7, 6, 3]

v.sort()
print v
# [2, 3, 3, 6, 7, 8]

v=[7,0,1,0,2,3,3,0,5]
v.sort()
print v
# [0, 0, 0, 1, 2, 3, 3, 5, 7]

v=[0,1,2,3,4,5,6,7]
v.reverse()
print v
# [7, 6, 5, 4, 3, 2, 1, 0]
```

Tuple

Tuple sono successioni finite *non modificabili* di elementi non necessariamente dello stesso tipo che sono elencati separati da virgole e possono facoltativamente essere incluse tra parentesi tonde. Per ragioni misteriose una tupla con un solo elemento deve essere scritta nella forma `(x,)` perché `(x)` è lo stesso come `x`. Ciò non vale per le liste: `[x]` è una lista. Tuple possono essere annidate.

Esempi:

```
x=3,5,8,9
print x
# (3, 5, 8, 9)

y=(3,5,8,9)
print y
# (3, 5, 8, 9)

s=(7)
print s
# 7

t=(7,)
print t
# (7,)

z=(x)
print z
# (3, 5, 8, 9)

u=(x,)
print u
# ((3, 5, 8, 9),)

v=(x,y)
print v
# ((3, 5, 8, 9), (3, 5, 8, 9))

w=1,2,(3,4,5),6,7
print w
# (1, 2, (3, 4, 5), 6, 7)

vuota=()
print vuota
# ()
```

La lunghezza di una tupla `v` la otteniamo con `len(v)`; l'`i`-esimo elemento di `v` è `v[i]`, contando (come in C e a differenza da R!) cominciando da 0.

```
x=3,5,8,9
print len(x)
# 4

for i in xrange(0,4): print x[i],
print
# 3 5 8 9

for a in x: print a,
# 3 5 8 9
```

Tuple vengono elaborate più velocemente e consumano meno spazio in memoria delle liste; per sequenze molto grandi (con centinaia di migliaia di elementi) o sequenze che vengono usate in migliaia di operazioni le tuple sono perciò talvolta preferibili alle liste. Liste d'altra parte non solo possono essere modificate, ma prevedono anche molte operazioni flessibili che non sono disponibili per le tuple. Le liste costituiscono una delle strutture fondamentali di Python. Con dati molto grandi comunque l'utilizzo di liste, soprattutto nei calcoli intermedi, può effettivamente rallentare notevolmente l'esecuzione di un programma.

Nomi ed assegnamento

A differenza dal C, il Python non distingue tra il nome `a` di una variabile e il suo indirizzo (che in C viene denotato con `&a`). Ciò ha implicazioni a prima vista sorprendenti nelle assegnazioni `b=a` in cui `a` è un oggetto mutabile (ad esempio una lista o un dizionario), mentre nel caso che `a` non sia mutabile (ad esempio un numero, una stringa o una tupla) non si avverte una differenza con quanto ci si aspetta.

Dopo `b=a` infatti `a` e `b` sono nomi diversi per lo stesso indirizzo, e se modifichiamo l'oggetto che si trova all'indirizzo `a`, lo troviamo cambiato anche quando usiamo il nome `b` proprio perché si tratta sempre dello stesso oggetto.

```
a=[1,5,0,2]; b=a; b.sort(); print a
# [0, 1, 2, 5]

b[2]=17; print a
# [0, 1, 17, 5]

b=a[:].sort(); print a
# [0, 1, 17, 5]
# a non e' cambiata.

b=a[:].reverse(); print a
# [0, 1, 17, 5]
# a non e' cambiata.
```

Se gli elementi di `a` sono immutabili, è sufficiente, come sopra, creare una copia `b=a[:]` oppure `b=list(a)` affinché cambiamenti in `b` non influenzino `a`. Ciò non basta più quando gli elementi di `a` sono mutabili, come ad esempio nel caso che `a` sia una lista annidata:

```
a=[[1,2],[3,4]]
b=a[:] # oppure b=list(a)
b[1][0]=17; print a
# [[1, 2], [17, 4]]
```

In questo caso bisogna creare una *copia profonda* utilizzando la funzione `copy.deepcopy` che naturalmente richiede il modulo `copy`:

```
import copy

a=[[1,2],[3,4]]
b=copy.deepcopy(a)
b[1][0]=17; print a
# [[1, 2], [3, 4]]
# a non e' cambiata.
```

Per verificare se due nomi denotano lo stesso oggetto, si può usare la funzione `is`, mentre l'operatore di uguaglianza `==` controlla soltanto l'uguaglianza degli elementi, non degli oggetti che corrispondono ai nomi `a` e `b`:

```
a=[1.5,0,2]; b=a
print b is a # True

a=[[1,2],[3,4]]; b=a[:]
print b is a
# False
print b[0] is a[0]
# True

a=[[1,2],[3,4]]
b=copy.deepcopy(a)
print b is a
# False
print b[0] is a[0] # False
print b==a # True
```

IV. LOGICA E CONTROLLO

Valori di verità

Vero e falso in Python sono rappresentati dai valori `True` e `False`. In un contesto logico, cioè nei confronti o quando sono argomenti degli operatori logici, anche ad altri oggetti è attribuito un valore di verità; come vedremo però, a differenza dal C, essi conservano il loro valore originale e il risultato di un'espressione logica in genere non è un valore di verità, ma uno degli argomenti da cui si partiva.

Con

```
a="Roma"; b="Torino"
for x in (3<5, 3<5<7, 3<5<4,
  6==7, 6==6,a=='Roma',a<b):
  print x,
```

otteniamo

```
True True False False True True True
```

perché la stringa `"Roma"` precede alfabeticamente `"Torino"`. Con

```
for x in (0,1,0.0,[],(),[0],
  None,'','alfa'):
  print "%-6s%s" %(x, bool(x))
```

otteniamo

```
0     False
1     True
0.0   False
[]    False
()    False
[0]   True
None  False
      False
alfa  True
```

Vediamo così che il numero 0, l'oggetto `None`, la stringa vuota, e la lista o la tupla vuota hanno tutti il valore di verità falso, numeri diversi da zero, liste, tuple e stringhe non vuote il valore di verità vero.

In un contesto numerico i valori di verità vengono trasformati in 1 e 0:

```
print (3<4)+0.7
# 1.7

v=[3<4,3<0]
for x in v: print x>0.5,
# True False

print
from rpy import r

a=[True,True,False,True,False]
print r.sum(a)
# 3
```

Uguaglianza:

```
a==b # a e b hanno lo stesso valore
a!=b # a e b hanno valori diversi
a=b # assegnamento
```

Operatori logici

Come abbiamo accennato, gli operatori logici `and` e `or`, a differenza dal C, non convertono i loro argomenti in valori di verità. Inoltre questi operatori (come in C, ma a differenza dalla matematica) non sono simmetrici. Più precisamente

```
A1 and A2 and ... and An
```

è uguale ad `An`, se tutti gli `Ai` sono veri, altrimenti il valore dell'espressione è il primo `Ai` a cui corrisponde il valore di verità falso. Ad esempio:

```
print 2 and 3 and 8
# 8
print 2 and [] and 7
# []
```

Similmente

```
A1 or A2 or ... or An
```

è uguale ad `An`, se nessuno degli `Ai` è vero, altrimenti è uguale al primo `Ai` che è vero:

```
print 0 or '' or []
# []
print 0 or [] or 2 or 5
# 2
```

Se per qualche ragione (ad esempio nella visualizzazione di uno stato) si desidera come risultato di queste operazioni un valore di verità, è sufficiente usare la funzione `bool`:

```
print bool(2 and 3 and 8)
# True
print bool(2 and [] and 7)
# False

print bool(0 or '' or [])
# False
print bool(0 or [] or 2 or 5)
# True
```

È molto importante nell'interpretazione procedurale degli operatori logici che in queste espressioni i termini che non servono non vengono nemmeno calcolati.

```
print math.log(0)
# Errore - il logaritmo di 0
# non e' definito.

print 1 or math.log(0)
# 1 - In questo caso il logaritmo
# non viene calcolato.
```

L'operatore di negazione logica `not` restituisce invece sempre un valore di verita:

```
print not []
# True

print not 5
# False
```

`not` lega più fortemente di `and` e questo più fortemente di `or`. Perciò le espressioni

```
(a and b) or c
(not a) and b
(not a) or b
```

possono essere scritte senza parentesi:

```
a and b or c
not a and b
not a or b
```

L'operatore di decisione

L'operatore ternario di decisione (o di diramazione) in C è rappresentato da un'espressione della forma `A ? x : y`.

Questa costituisce *un valore* che è uguale ad `x`, se `A` è vera, altrimenti uguale ad `y`.

L'operatore di diramazione è molto importante nell'informatica teorica (ad esempio nella tecnica dei *diagrammi binari di decisione* nello studio delle funzioni booleane) ed è spesso scritto nella forma $[A, x, y]$.

Per imitare questo operatore in Python definiamo la funzione

```
def iftern (A,x,y): # if ternario
  if A: return x
  else: return y
```

Saremmo tentati a utilizzare questa funzione per il calcolo del fattoriale con

```
def f (n): # Brutta sorpresa!
  return iftern(n==0,1,n*f(n-1))
```

ma quando usiamo questa funzione, ci aspetta una brutta sorpresa: l'interprete ci segnala che abbiamo innescato una ricorsione infinita, come se mancasse la condizione di terminazione. Ed è proprio così, perché non riusciamo a calcolare il fattoriale di 0. Infatti questo corrisponde a

```
iftern(True,1,0*f(-1)
```

e vediamo che dobbiamo calcolare

```
iftern(False,1,-1*f(-2)
```

ecc. Vediamo quindi che gli operatori logici della matematica, che sono simmetrici, non si prestano a un utilizzo procedurale. Per questa ragione nei linguaggi di programmazione gli operatori logici sono definiti nel modo non simmetrico che abbiamo visto in precedenza; su questa interpretazione procedurale dell'asimmetria degli operatori logici si basa la *programmazione logica*, su essa il linguaggio Prolog.

Il linguaggio R, molto usato in statistica e molto simile al Python, possiede una funzione `ifelse` di R che può essere considerata una generalizzazione vettoriale di `iftern` a cui è equivalente nel caso di vettori di lunghezza 1.

```
x=(1,0,1,0,1,0,1,0,1,0,1,0)
a=(7,None,8,None,9,None)
b=(None,1,None,2)
u=r.ifelse(x,a,b); print u
# [7, 1, 8, 2, 9, 1, 7, 2, 8, 1, 9, 2]
```

Logica procedurale

Esaminiamo ancora la funzione `f` vista a pagina 13. La funzione fallisce perché cerca di definire la condizione di terminazione all'interno dell'espressione

```
iftern(n==0,1,n*f(n-1))
```

che richiede il calcolo di `f(n-1)` anche quando il risultato non è usato perché la condizione iniziale `n==0` è soddisfatta. Possiamo invece ridefinire la funzione nel modo seguente:

```
def fatt (n):
  return iftern(n==0,1,n and n*fatt(n-1))
```

Se proviamo la funzione, vediamo che viene calcolata correttamente. Esaminiamo separatamente i casi `n==0` e `n>0` (naturalmente l'algoritmo ricorsivo non può essere usato per valori negativi di `n`):

`n==0`: Bisogna calcolare non soltanto il risultato 1, ma anche l'espressione `0 and 0*fatt(-1)`. Siccome però 0 in un contesto logico è falso, la definizione asimmetrica di `and` implica che il secondo termine `0*fatt(-1)` non viene più calcolato e quindi non provoca errori.

`n>0`: In questo caso bisogna di nuovo calcolare 1 (benché non sia il risultato) e l'espressione `n and n*fatt(n-1)`. Siccome `n` stavolta è vero, perché diverso da zero, l'espressione è uguale ad `n*fatt(n-1)`. Infatti la regola per il calcolo di `and` implica (applicata al caso particolare di due argomenti) che `True and X` è sempre uguale ad `X`, indipendentemente dal valore di verità che ha `X`.

Diamo alcuni altri esempi per l'utilizzo procedurale degli operatori logici.

```
def positivo (x): return x>0 or False

def segno (x):
  return x>0 and 1 or x<0 and -1 or 0

def bin (n,k): # Numeri binomiali.
  return k==0 and 1 or n<k and 0
    or (float(n)/k)*bin(n-1,k-1)

def prod (a,b): # a(a+1) ... b
  return iftern(a>b,1,a<=b and
    iftern(a==b,a,a<b and b*prod(a,b-1)))

for b in xrange(2,7): print prod(3,b),
# 1 3 12 60 360
```

Giustificare gli algoritmi per `bin` e `prod`.

Per ridefinire `iftern` mediante gli operatori logici non possiamo utilizzare direttamente

```
def iftern (A,x,y): # Non corretto!
  return A and x or y
```

perché allora `iftern(True,0,y)` sarebbe uguale ad `y`, ma piuttosto

```
def iftern (A,x,y):
  return (A and [x] or [y])[0]
```

Si vede che l'uso procedurale degli operatori logici, nonostante l'importanza teorica di queste costruzioni, non è sempre facilmente leggibile; in pratica in Python è spesso preferibile l'utilizzo di `if ... else`.

if ... elif ... else

Le istruzioni condizionali in Python vengono utilizzate con la sintassi

```
if A: alfa()

if A: alfa()
else: beta()

if A:
  if B: alfa()
  else: beta()
else: gamma()
```

Non dimenticare i doppi punti. Spesso si incontrano diramazioni della forma

```
if A: alfa()
else:
  if B: beta()
  else:
    if C: gamma()
    else: delta()
```

In questi casi i doppi punti e la necessità delle indentazioni rendono la composizione del programma difficoltosa; è prevista perciò in Python l'abbreviazione `elif` per un `else ... if` come nell'ultimo esempio che può essere riscritto in modo più semplice:

```
if A: alfa()
elif B: beta()
elif C: gamma()
else: delta()
```

Esempio:

```
def segno (x):
  if x>0: return 1
  elif x==0: return 0
  else: return -1
```

Purtroppo il Python non prevede la costruzione `switch ... case` del C. Con un po' di attenzione la si può comunque emulare con l'impiego di una serie di `elif` oppure, come nel seguente esempio, mediante l'impiego adeguato di un dizionario:

```
operazioni = {'Roma' : 'print "Lazio"',
  'Ferrara' : 'print "Romagna"',
  'Cremona' : 'x=5; print x*x'}
for x in ['Roma','Ferrara','Cremona']:
  exec(operazioni[x])
```

`exec(a)` esegue la stringa `a` come se fosse un'istruzione del programma.

Operatori aritmetici

Interi possono avere un numero arbitrario di cifre. Il quoziente intero di due interi a e b lo si ottiene con `a/b`, quindi ad esempio `28/11` è uguale a 6, e il resto di a modulo b è dato da `a%b`. Purtroppo gli operatori `/` e `%`, come peraltro in C, non funzionano correttamente (dal punto di vista matematico) per $b < 0$, ad esempio `40%(-6)` è -2, mentre in matematica si vorrebbe che il resto r modulo b soddisfi $0 \leq r < |b|$. Una soluzione di questo problema si trova alle pagine 17-18 del corso di Programmazione 2004/05. Python fornisce anche la funzione `divmod` che restituisce la coppia `(a/b,a%b)`. Per ottenere il quoziente reale di due numeri interi bisogna definire uno degli operandi come numero reale, come ad esempio in `20.0/7` oppure `float(20)/7`. Si può anche calcolare il quoziente intero di numeri reali (con lo stesso difetto indicato prima) mediante l'operatore `//` che è stato definito in Python 2.2 con lo scopo di poter ridefinire in futuro (probabilmente dalla versione 3.0) l'operatore `/` come operatore di divisione reale. Le potenze x^a (ad esponenti reali) si ottengono con `x**a`:

```
print 10**math.log10(17)
# 17.0
```

filter

Questa utilissima funzione con la sintassi `filter(f,v)` estrae da un vettore `v` tutti gli elementi `x` per cui `f(x)` è vera e restituisce come risultato il vettore che consiste di tutti questi elementi. In R filtri sono realizzati mediante vettori di indici logici.

Dopo aver definito

```
def pari (x): return x%2==0
```

possiamo estrarre tutti i numeri pari da un vettore:

```
v=xrange(1,21)
print filter(pari,v)
# [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Similmente da un elenco di parole possiamo estrarre le parole che hanno lunghezza ≥ 5:

```
v=['Roma','Ferrara','Bologna','Pisa']
print filter(lambda x: len(x)>4,v)
# ['Ferrara', 'Bologna']
```

L'espressione `lambda x: f(x)` corrisponde alla nostra notazione $\bigcirc_x f(x)$. Filtri si usano anche nella gestione di basi di dati; possiamo ad esempio trovare tutti gli impiegati di una ditta che guadagnano almeno 3000 euro al mese:

```
imp = [['Rossi',2000],['Verdi',3000],
  ['Gentili',1800],['Bianchi',3400],
  ['Tosi',1600],['Neri',2800]]
imp3000=filter(lambda x: x[1]>=3000,imp)
for x in imp3000: print x[0],
# Verdi Bianchi
```

Il crivello di Eratostene

Questo antico metodo per trovare i numeri primi $\leq n$ può essere programmato molto facilmente con Python:

```
def Eratostene (n):
  v=range(2,n+1); u=[]
  r=math.sqrt(n); p=2
  while p<=r:
    p=v[0]; u.append(p)
    v=filter(lambda x: x%p>0,v)
  return u+v

v=Eratostene(100); m=len(v)
for i in xrange(0,m):
  if i%8==0: print
  print v[i],

# 2 3 5 7 11 13 17 19 23 29
# 31 37 41 43 47 53 59 61 67 71
# 73 79 83 89 97
```

for

La sintassi di base del `for` ha la forma

```
for x in v: istruzioni
```

dove `v` è una sequenza o un iteratore.

Dal `for` si esce con `break` (o naturalmente, da una funzione, con `return`), mentre `continue` interrompe come in C il passaggio corrente di un ciclo e porta all'immediata esecuzione del passaggio successivo, cosicché

```
for x in xrange(0,21):
  if x%2>0: continue
  print x,
# 0 2 4 6 8 10 12 14 16 18 20
```

stampa sullo schermo i numeri pari compresi tra 1 e 20.

Il `for` può essere usato anche nella forma

```
for x in v: istruzioni
else: istruzionefinale
```

Quando le istruzioni nel `for` stesso non contengono un `break`, questa sintassi è equivalente a

```
for x in v: istruzioni
```

Un `break` invece *salta* la parte `else` che quindi viene eseguita solo se tutti i passaggi previsti nel ciclo sono stati effettuati. Questa costruzione viene utilizzata quando il percorso di tutti i passaggi è considerato come una condizione di eccezione: Assumiamo ad esempio che cerchiamo in una sequenza un primo elemento con una determinata proprietà - una volta trovato, usciamo dal ciclo e continuiamo l'elaborazione con questo elemento; se invece un elemento con quella proprietà non si trova, abbiamo una situazione diversa che trattiamo nel `else`. Nei due esempi che seguono cerchiamo il primo elemento positivo di una tupla di numeri:

```
for x in (-1,0,0,5,2,-3,4):
  if x>0: print x; break
else: print 'Nessun elemento positivo.'
# 5

for x in (-1,0,0,-5,-2,-3,-4):
  if x>0: print x; break
else: print 'Nessun elemento positivo.'
# Nessun elemento positivo.
```

Cicli `for` possono essere annidati; con

```
for i in xrange(4):
  for j in xrange(5):
    print i+j,
  print
```

otteniamo

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

Se gli elementi della sequenza che viene percorsa sono a loro volta sequenze tutte della stessa lunghezza, nel `for` ci possiamo riferire agli elementi di queste sequenze con nomi di variabili:

```
u=[[1,10],[2,10],[3,10],[4,20]]
for x,y in u: print x+y,
print
# 11 12 13 24

v=['Aa','Bb','Cc','Dd','Ee']
for x,y in v: print y+'.'+x,
print
# a.A b.B c.C d.D e.E

w=[[1,2,5],[2,3,6],[11,10,9]]
for x,y,z in w: print x*y+z,
# 7 12 119
```

Combinando il `for` con `zip` possiamo calcolare il prodotto scalare di due vettori:

```
def prodottoscalare (u,v):
  s=0
  for x,y in zip(u,v): s+=x*y
  return s

u=[1,3,4]
v=[6,2,5]

print prodottoscalare(u,v)
# 32
```

while

Il `while` controlla cicli più generali del `for` e gli è molto simile nella sintassi:

```
while A: istruzioni
```

oppure

```
while A: istruzioni
else: istruzionefinale
```

`break` e `continue` vengono utilizzati come nel `for`. Se è presente un `else`, l'istruzione finale viene eseguita se l'uscita dal ciclo è avvenuta perché la condizione `A` non era più soddisfatta, mentre viene saltata se si è usciti con un `break` o un `return`.

```
x=0; v=[]
while not x in v:
  v.append(x)
  x=(7*x+13)%17

for x in v: print x,
print
# 0 13 2 10 15 16 6 4 7 11 5 14 9 8 1 3

while 1:
  nome=raw_input('Come ti chiami? ')
  if nome=='': break
  print 'Ciao, %s!' %(nome)
```

zip

Questa funzione utilissima corrisponde essenzialmente alla formazione della trasposta di una matrice. Esempio:

```
a=[1,2,3]; b=[11,12,13]
c=[21,22,23]; d=[31,32,33]
for x in zip(a,b,c,d): print x
# Output:

(1, 11, 21, 31)
(2, 12, 22, 32)
(3, 13, 23, 33)
```

Il risultato di `zip` è sempre una lista i cui elementi sono tuple (cfr. la funzione `matricedavettore` a pagina 22). Quando gli argomenti non sono tutti della stessa lunghezza, viene usata la lunghezza minima, come in

```
a=[1,2,3,4]; b=[11,12]
c=[21,22,23,24]
for x in zip(a,b,c): print x
# Output:

(1, 11, 21)
(2, 12, 22)
```

Il tipo dei dati non ha importanza:

```
a=[0,1,2,3]; b=['a',7,'geo',[9,10]]
c=[11,12,13,14]
d=['A','B','C',['E','F']]
for x in zip(a,b,c,d): print x
# Output:

(0, 'a', 11, 'A')
(1, 7, 12, 'B')
(2, 'geo', 13, 'C')
(3, [9, 10], 14, ['E', 'F'])
```

Uso di `zip` con `for x,y`:

```
nomi=('Verdi','Rossi','Bianchi')
stipendi=(2000,1800,2700)
for x,y in zip(nomi,stipendi):
  print x,y
# Output:

Verdi 2000
Rossi 1800
Bianchi 2700
```

try ... except

Non sempre è possibile prevedere se un'istruzione può effettivamente essere eseguita correttamente. Ciò si verifica ad esempio se dovrebbe essere aperto un file che potrebbe anche non esistere o non accessibile, o per qualche altra ragione.

Per questi casi Python prevede la costruzione `try ... except`, simile ad un `if ... else`, in cui la prima parte contiene le istruzioni che vorremmo eseguire nel caso che tutto funzioni, la seconda parte le istruzioni da effettuare nel caso che si verifichi un errore. Nella seconda parte quindi possiamo prevedere messaggi d'errore, oppure una soluzione alternativa, oppure che non venga effettuata nessuna operazione.

Illustriamo la sintassi con due esempi presi dal nostro editor ECP:

```
try: tk(campotesto,'edit undo'); undo=1
except: return

try:
  (fi,foe)=os.popen4(comando,'t'); a=foe.read()
  fi.close(); foe.close()
  Scrivirisultato(a,infile=infile)
  if catalogo: Catalogo(conaiuto=0)
except: Scrivirisultato('Errore',infile=0)
```

enumerate

Un'altra funzione utile è `enumerate` che da un vettore `v` genera un iteratore che corrisponde alle coppie `(i,v[i])`. Se trasformiamo l'iteratore in una lista, vediamo che l'oggetto che si ottiene è essenzialmente equivalente a `zip(xrange(len(v)),v)`:

```
v=['A','D','E','N']

e=enumerate(v)
print e
# <enumerate object at 0x1f726c>

a=list(enumerate(v))
b=zip(xrange(len(v)),v)
print a==b
# True

# Infatti:
print a
# [(0,'A'), (1,'D'), (2,'E'), (3,'N')]

print b
# [(0,'A'), (1,'D'), (2,'E'), (3,'N')]
```

`enumerate` è però più efficiente di `zip` e viene tipicamente usato quando nel percorrere un vettore bisogna tener conto sia del valore degli elementi sia della posizione in cui si trovano nel vettore. Assumiamo ad esempio che vogliamo calcolare la somma degli indici di quegli elementi in un vettore numerico che sono maggiori di 10:

```
v=[8,13,0,5,17,8,6,24,6,15,3]

s=0
for i,x in enumerate(v):
  if x>10: s+=i
print s
# 21, perche' sono maggiori di 10
# gli elementi con gli indici
# 1,4,7,9 e 1+4+7+9=21.
```

Possiamo con questa tecnica anche definire una funzione che calcola il valore di una somma trigonometrica

$$\sum_{n=0}^{N} a_n \cos nx$$

con

```
def sommatrigonometrica (a,x):
  s=0
  for n,an in enumerate(a):
    s+=an*math.cos(n*x)
  return s

a=[2,3,0,4,7,1,3]

print sommatrigonometrica(a,0.2)
# 14.7458647279
```

Nello stesso modo potremmo anche calcolare il valore $f(\alpha)$ di un polinomio $\sum_{n=0}^{N} a_n x^n$, ma vedremo che lo schema di Horner fornisce un algoritmo più efficiente sia per i polinomi che per le somme trigonometriche. Nonostante ciò anche in questo contesto `enumerate` può risultare utile, ad esempio per calcolare somme della forma generale $\sum_{n=0}^{N} \varphi(n, x)$.

map semplice

f sia una funzione (o un'espressione `lambda`) con un singolo argomento, a una sequenza o un iteratore con gli elementi $a_0, \ldots, a_n$. Allora `map(f,a)` è la lista $[f(a_0), \ldots, f(a_n)]$. Esempi:

```
a='ABCDabcd'
print map(ord,a)
# [65, 66, 67, 68, 97, 98, 99, 100]
```

Infatti, per un carattere `x` si ottiene con `ord(x)` il suo codice ASCII.

```
u=map(lambda x: x**2, xrange(10))
print u
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

L'esempio che segue esprime in poche righe la potenza di Python:

```
def vocale (x):
  return x.lower() in 'aeiou'

print map(vocale,'Crema')
# [False, False, True, False, True]
```

Usiamo `map` per costruire il grafico di una funzione:

```
def grafico (f,a):
  return zip(a,map(f,a))

a=(0,1,2,3)
print grafico(lambda x: x**4,a)
# [(0, 0), (1, 1), (2, 16), (3, 81)]
```

La funzione `str` trasforma un oggetto in una stringa e, utilizzata insieme a `join`, permette ad esempio di trasformare una lista di numeri in una stringa i cui i numeri appaiono separati da spazi:

```
a=(88,20,17,4,58)
v=map(str,a)
print v
# ['88', '20', '17', '4', '58']

print ' '.join(v)
# 88 20 17 4 58
```

Se X è un insieme, la diagonale n-esima di X è il sottoinsieme di X^n che consiste delle tuple della forma $(x, \ldots, x)$:

```
def diagonale (a,n=2):
  return map(lambda x: (x,)*n,a)
# Non dimenticare la virgola.

a=(2,3,5,6)
print diagonale(a)
# [(2, 2), (3, 3), (5, 5), (6, 6)]
```

Alcune regolarità del codice genetico possono essere studiate meglio se le lettere G,A,C,T vengono sostituite con 0,1,2,3:

```
def dnanumerico (x):
  x=x.lower()
  sost={'g': 0, 'a': 1, 'c': 2, 't': 3}
  return sost[x]

g='TGAATGCTAC'
print map(dnanumerico,g)
# [3, 0, 1, 1, 3, 0, 2, 3, 1, 2]
```

Cesare codificava talvolta i suoi messaggi sostituendo ogni lettera con la lettera che si trova a tre posizioni dopo la lettera originale, calcolando le posizioni in maniera ciclica. Assumiamo di avere un alfabeto di 26 lettere (senza minuscole, spazi o interpunzioni):

```
def cesare (a):
  o=ord('A')
  def codifica(x):
    n=(ord(x)-o+3)%26
    return chr(o+n)
  v=map(codifica,a)
  return ''.join(v)

a='CRASCASTRAMOVEBO'
print cesare(a)
# FUDVFDVWUDPRYHER
```

I `map` e i `lambda` possono essere annidati. Per costruire una matrice identica possiamo usare (non è questo il modo più efficiente) la seguente costruzione:

```
a=map(lambda i:
  map (lambda j:
    i==j,xrange(4)), xrange(4))
for x in a: print x
```

ottenendo

```
[True, False, False, False]
[False, True, False, False]
[False, False, True, False]
[False, False, False, True]
```

Addizionando 0 a `True` o `False` otteniamo il valore 1 o 0 corrispondente. Modificando leggermente la funzione precedente, troviamo quindi la matrice identica numerica:

```
a=map(lambda i:
  map (lambda j:
    (i==j)+0,xrange(4)), xrange(4))
for x in a: print x
```

con output

```
[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
```

Usando la funzione `rapp2` che sarà definita a pagina 24 possiamo ottenere un ipercubo:

```
def ipercubo (m):
  def rapp(x): return rapp2(x,cifre=m)
  return map(rapp,xrange(2**m))

for x in ipercubo(4): print x # Output:
```

```
[0, 0, 0, 0]
[0, 0, 0, 1]
[0, 0, 1, 0]
[0, 0, 1, 1]
[0, 1, 0, 0]
[0, 1, 0, 1]
[0, 1, 1, 0]
[0, 1, 1, 1]
[1, 0, 0, 0]
[1, 0, 0, 1]
[1, 0, 1, 0]
[1, 0, 1, 1]
[1, 1, 0, 0]
[1, 1, 0, 1]
[1, 1, 1, 0]
[1, 1, 1, 1]
```

map multivariato

$a = (a_0, \ldots, a_n)$ e $b = (b_0, \ldots, b_n)$ siano due sequenze o iteratori della stessa lunghezza ed f una funzione di 2 argomenti. Allora `map(f,u,v)` è la lista

$$[f(a_0, b_0), \ldots, f(a_n, b_n)]$$

In modo simile sono definite le espressioni `map(f,u,v,w)` per funzioni di tre argomenti ecc. Esempi:

```
def somma (*a): # Come a pagina 2.
  s=0
  for x in a: s+=x
  return s

print map(somma,
  (0,1,5,4),(2,0,3,8),(6,2,2,7))
# [8, 3, 10, 19]
```

`None`, pur non essendo una funzione, nel `map` è considerata come la funzione identica:

```
print map(None,(1,2,3))
# [1, 2, 3]

print map(None,(11,12,13),(21,22,23))
# [(11, 21), (12, 22), (13, 23)]

v=map(None,
  (11,12,13),(21,22,23),(31,32,33))
for r in v: print r
```

L'ultimo output è

```
(11, 21, 31)
(12, 22, 32)
(13, 23, 33)
```

Vediamo che `map(None,...)` è equivalente all'utilizzo di `zip`. Un altro esempio:

```
a=(2,3,5,0,2); b=(1,3,8,0,4)

def f(x,y): return (x+y,x-y)

for x in map(f,a,b): print x,
# (3,1) (6,0) (13,-3) (0,0) (6,-2)
```

map implicito

Il Python prevede una costruzione che permette spesso di sostituire il `map` con un `for`. Se `f` è una funzione in una variabile,

```
[f(x) for x in a]
```

è equivalente a `map(f,a)`; similmente per una funzione di due variabili

```
[f(x,y) for x,y in ((a1,b1),(a2,b2),...)]
```

corrisponde a `map(f,a,b)`. Questa forma è spesso più intuitiva e talvolta più breve, soprattutto nel caso di espressioni lambda sufficientemente semplici:

```
a=('Roma','Pisa','Milano','Trento')
print [x[0] for x in a]
# ['R', 'P', 'M', 'T']

a=[x*x+3 for x in xrange(8)]
print a
# [3, 4, 7, 12, 19, 28, 39, 52]

a=[x+y for x,y in ((3,2),(5,6),(1,9))]
print a
# [5, 11, 10]
```

Scrivere su più righe

Siccome Python usa l'indentazione per strutturare il testo sorgente, per poter scrivere un'istruzione su più righe, dobbiamo indicarlo all'interprete. Ciò avviene ponendo un \ alla fine di ogni riga che si vuole continuare su quella successiva. Il simbolo \ non è necessario, quando si distribuiscono su più righe parametri separati da virgole.

```
# Qui \ e' necessario.
print x+3*x*x+math.log(4*x+\
  math.sqrt(1+x*x))

# Non e' richiesto un \.
print f(math.log(x),math.log(y),
  x+y+z)
```

Abbiamo già visto che testi su più righe possono essere racchiusi tra apici o virgolette triplici:

```
a='''Taciti, soli, sanza compagnia
n'andavam, l'un dinanzi e l'altro dopo,
come i frati minor vanno per via.
...
La' giu' trovammo una gente dipinta,
che giva intorno assai con lenti passi'''
```

Come si vede, un apice singolo può apparire all'interno di apici triplici, bisogna però evitare un '''' finale, perché allora i primi tre apici verrebbero letti come fine della stringa lunga.

Libri su Python

J. Goerzen: Foundations of Python network programming. Apress 2004.

S. Holden: Python web programming. New Riders 2002.

J. Kiusalaas: Numerical methods in engineering with Python. Cambridge UP 2005.

H. Langtangen: Python scripting for computational science. Springer 2004.

M. Lutz/D. Ascher: Programmare con Python. Hoepli 2004.

A. Martelli: Python in a nutshell. O'Reilly 2003.

A. Martelli/A. Ravenscroft/D. Ascher (ed.): Python cookbook. O'Reilly 2005.

D. Mertz: Text processing in Python. Addison-Wesley 2003.

M. Pilgrim: Dive into Python. Apress 2004.

J. Zelle: Python programming. Franklin 2004.

sort

Sappiamo dalla pagina 12 che il metodo `sort` permette di ordinare una lista. Per una lista `a` la sintassi completa è

```
a.sort(cmp=None,key=None,reverse=False)
```

Se non reimpostiamo il parametro `cmp`, il Python utilizza una funzione di confronto naturale, essenzialmente l'ordine alfabetico (inteso in senso generale). `key` è una funzione che viene applicata ad ogni elemento della lista prima dell'ordinamento; lasciando `key=None`, gli elementi vengono ordinati così come sono. `reverse=True` implica un ordinamento in senso decrescente.

Come esempio dell'uso del parametro `key` assumiamo che vogliamo ordinare una lista di numeri tenendo conto dei valori assoluti:

```
a=[1,3,-5,0,-3,7,-10,3]
a.sort(key=abs)
print a
# [0, 1, 3, -3, 3, -5, 7, -10]
```

Similmente possiamo usare come elementi di confronto i resti modulo 100; ciò è equivalente naturalmente a considerare solo le ultime due cifre di un numero naturale:

```
a=[3454,819,99,4545,716,310]
a.sort(key=lambda x: x%100)
print a
# [310, 716, 819, 4545, 3454, 99]
```

Utilizzando `key=string.lower` possiamo ordinare una lista di stringhe alfabeticamente, prescindendo dalla distinzione tra minuscole e maiuscole.
Impostando `cmp=f` è anche possibile utilizzare una propria funzione di confronto `f`. Questa deve essere una funzione di due variabili `x,y` che restituisce -1 (o un numero negativo) se consideriamo `x` minore di `y`, 0 se i due numeri sono considerati uguali, 1 (o un numero positivo) se consideriamo `x` maggiore di `y`. La funzione dovrebbe soddisfare la condizione `f(x,x)==0` e indurre una relazione transitiva.

Consideriamo ad esempio una lista `a` minore di una lista `b` se `a` ha meno elementi di `b`:

```
def conflun (a,b):
  if len(a)<len(b): return -1
  if len(a)==len(b): return 0
  return 1

v=[[6,2,0],[9,1],[4,5,8,3],[3,1,7]]
v.sort(cmp=conflun)
print v
# [[9,1],[6,2,0],[3,1,7],[4,5,8,3]]
```

Naturalmente in questo caso lo stesso effetto l'avremmo potuto ottenere con

```
v.sort(key=len)
```

Infatti molto spesso nella pratica (e teoricamente sempre) è più conveniente impostare il parametro `key` piuttosto di impostare `cmp`.

V. ALGORITMI ELEMENTARI

Lo sviluppo di Taylor

Sia dato un polinomio

$$f = a_0x^n + \ldots + a_n$$

Fissato α, con lo schema di Horner otteniamo i valori $b_0, \ldots, b_n$ che sono tali che $b_n = f(\alpha)$ e

$$g := b_0x^{n-1} + b_1x^{n-2} + \ldots + b_{n-1}$$

è il quoziente nella divisione con resto di f per $x - \alpha$. Ponendo $f_0 := f$ ed $f_1 := g$ possiamo scrivere

$$f_0 = (x - \alpha)f_1 + b_n$$

Se f ha grado 1, allora f_1 è costante e la rappresentazione ottenuta è lo sviluppo di Taylor di f. Altrimenti possiamo applicare lo schema di Horner (con lo stesso α) ad f_1, ottenendo valori $c_0, \ldots, c_{n-1}$ tale che con $f_2 := c_0x^{n-2} + \ldots + c_{n-2}$ si abbia

$$f_1 = (x - \alpha)f_2 + c_{n-1}$$

cosicché

$$f_0 = (x - \alpha)^2f_2 + (x - \alpha)c_{n-1} + b_n$$

Se f ha grado 2, questo è lo sviluppo di Taylor, altrimenti continuiamo, ottenendo valori $d_0, \ldots, d_{n-2}$ tali che con $f_3 := d_0x^{n-3} + \ldots + d_{n-3}$ si abbia

$$f_2 = (x - \alpha)f_3 + d_{n-2}$$

cosicché

$$f_0 = (x - \alpha)^3f_3 + (x - \alpha)^2d_{n-2} + (x - \alpha)c_{n-1} + b_n$$

Se f ha grado 3, questo è lo sviluppo di Taylor, altrimenti continuiamo nello stesso modo.

Per calcolare lo sviluppo di Taylor con una funzione in Python dobbiamo quindi prima definire una funzione `hornercompleto` che calcola non solo il valore b_n, ma tutti i b_k. Da essa otteniamo la funzione `taylor` che ripete il procedimento fino a quando il vettore dei coefficienti è vuoto:

```
def hornercompleto (a,x):
  b=0; v=[]
  for ak in a: b=b*x+ak; v.append(b)
  return v

def taylor (a,x):
  v=a # Corretto, ma cfr. terza colonna.
  w=[]
  while v:
    v=hornercompleto(v,x)
    w.append(v.pop())
  return w

a=(3,5,6,8,7)
print taylor(a,2)
# [135, 188, 108, 29, 3]
```

Lo sviluppo di Taylor di

$$f = 3x^4 + 5x^3 + 6x^2 + 8x + 7$$

per $x = 2$ è quindi

$$f = 135 + 188(x - 2) + 108(x - 2)^2 + 29(x - 2)^3 + 3(x - 2)^4$$

Esercizio: Calcolare le derivate $f^{(k)}(2)$.

Argomenti opzionali

Gli argomenti di una funzione in Python nella chiamata della funzione possono essere indicati con i nomi utilizzati nella definizione, permettendo in tal modo di modificare l'ordine in cui gli argomenti appaiono:

```
def f(x,y):
  return x+2*y

print f(y=2,x=7)
# 11
```

Quando argomenti con nome nella chiamata vengono utilizzati insieme ad argomenti senza nomi, questi ultimi vengono identificati per la loro posizione; ciò implica che argomenti senza nomi devono sempre precedere eventuali argomenti con nome:

```
def f(x,y,z): return x+2*y+3*z

print f(2,z=4,y=1)
# 16
```

Un argomento a cui già nella definizione viene assegnato un valore, è opzionale e può essere tralasciato nelle chiamate della funzione:

```
def f(x,y,z=4): return x+2*y+3*z

print f(2,1)
# 16

print f(2,1,5)
# 19

print f(2,z=5,y=1)
# 19

def tel (nome,numero,prefisso='0532'):
  return [nome,prefisso+'-'+numero]

print tel('Rossi','974002')
# ['Rossi', '0532-974002']
```

Utilizziamo questa possibilità nelle definizioni delle funzioni `diagonale` (pagina 16) e `rapp2` (pagina 24).

Bisogna qui stare attenti al fatto che il valore predefinito viene assegnato solo in fase di definizione; quando questo valore è mutabile, ad esempio una lista, in caso di più chiamate della stessa funzione viene utilizzato ogni volta il valore fino a quel punto raggiunto:

```
def f(x,v=[]):
  v.append(x); print v

f(7)
# [7]

f(8)
# [7, 8]
```

Ciò è in accordo con il comportamento descritto nella terza colonna su questa pagina. Invece

```
def f(x,y=3):
  print x+y; y+=1

f(1)
# 4

f(1)
# 4 - perche' y non e' mutabile.
```

Il più piccolo divisore

Per trovare il più piccolo divisore $d > 1$ di un numero intero $n \neq \pm 1, 0$, possiamo usare il seguente algoritmo, in cui (come nel crivello di Eratostene) usiamo che se n non è primo, necessariamente si deve avere $d^2 \leq \sqrt(n)$ e quindi anche $d^2 \leq [\sqrt(n)]$.

```
def divmin (n):
  r=int(math.sqrt(n))
  for d in xrange(2,r+1):
    if n%d==0: return d
  else: return n

print divmin(323)
# 17
print divmin(31)
# 31

for x in xrange(321,342,2):
  print divmin(x),
# 3 17 5 3 7 331 3 5 337 3 11
# Si presta per un algoritmo
# di crivello.
```

Il più piccolo divisore di 0 è 2, perché 0 è divisibile per ogni numero intero. Il nostro algoritmo però non funziona in questo caso, perché `n%0` provoca un errore. Potremmo anche inserire all'inizio della funzione le righe

```
  if n==0: return 2
  if n==1 or n==-1: return None
```

Una trappola pericolosa

Abbiamo già visto a pagina 12 che i nomi del Python devono essere considerati come nomi di puntatori nel C e che quindi un'assegnazione `b=a` dove `a` è un nome (e non una costante) implica che `b` ed `a` sono nomi diversi per lo stesso indirizzo. Ciò vale (come per i puntatori del C!) anche all'interno di funzioni, per cui una funzione può effettivamente modificare il valore dei suoi argomenti (più precisamente il valore degli oggetti a cui i nomi degli argomenti corrispondono). Mentre in C, se il programmatore ha davanti agli occhi la memoria su cui sta lavorando, ciò è piuttosto evidente e comprensibile, la sintattica semplice del Python induce facilmente a dimenticare questa circostanza.

```
def f(x,a):
  b=a; b.append(x); return b

a=[1,2]

f(4,a)
print a
# [1, 2, 4]

f(5,a)
print a
# [1, 2, 4, 5]
```

Nella funzione `taylor` su questa pagina abbiamo utilizzato semplicemente un'assegnamento `v=a`. Ciò è corretto, ma soltanto perché successivamente `v` viene riassegnato al valore di `hornercompleto` prima che entrino in azione i `v.pop()` del ciclo che modificano solo il nuovo `v` e non `a`. Comunque bisogna stare molto attenti in questi casi e soltanto la particolare forma dell'algoritmo rende corretta l'assegnamento semplice; altrimenti avremmo dovuto usare `b=a[:]` o addirittura `b=copy.deepcopy(a)`.

Il massimo comune divisore

Tutti i numeri $a, b, c, d, \ldots$ considerati sono interi, cioè elementi di $\mathbb{Z}$. Usiamo l'abbreviazione $\mathbb{Z}d := \{nd \mid n \in \mathbb{Z}\}$.

Diciamo che a è un *multiplo* di d se $a \in \mathbb{Z}d$, cioè se esiste $n \in \mathbb{Z}$ tale che $a = nd$. In questo caso diciamo anche che d *divide* a o che d è un *divisore* di a e scriviamo $d|a$.

Definizione 19.1. Per $(a, b) \neq (0, 0)$ il *massimo comune divisore* di a e b, denotato con $\mathrm{mcd}(a, b)$, è il più grande $d \in \mathbb{N}$ che è un comune divisore di a e b, cioè tale che $d|a$ e $d|b$. Poniamo invece $\mathrm{mcd}(0, 0) := 0$. In questo modo $\mathrm{mcd}(a, b)$ è definito per ogni coppia (a, b) di numeri interi.

Perché esiste $\mathrm{mcd}(a, b)$? Per $(a, b) = (0, 0)$ è uguale a 0 per definizione. Assumiamo che $(a, b) \neq (0, 0)$. Adesso $1|a$ e $1|b$ e se $d|a$ e $d|b$ e ad esempio $a \neq 0$, allora $d \leq |a|$, per cui vediamo che esiste solo un numero finito (al massimo $|a|$) di divisori comuni ≥ 1, tra cui uno ed uno solo deve essere il più grande. $\mathrm{mcd}(a, b)$ è quindi univocamente determinato e uguale a 0 se e solo se $a = b = 0$. Si noti che $d|a \iff -d|a$, per cui possiamo senza perdita di informazioni assumere che $d \in \mathbb{N}$.

L'algoritmo euclideo

Questo algoritmo familiare a tutti e apparentemente a livello solo scolastico, è uno dei più importanti della matematica ed ha numerose applicazioni: in problemi pratici (ad esempio nella grafica al calcolatore), in molti campi avanzati della matematica (teoria dei numeri e analisi complessa), nell'informatica teorica. L'algoritmo euclideo si basa sulla seguente osservazione (*lemma di Euclide*):

Lemma 19.2. *Siano a, b, c, q, d numeri interi e $a = qb + c$. Allora*

$$(d|a \text{ e } d|b) \iff (d|b \text{ e } d|c)$$

Quindi i comuni divisori di a e b sono esattamente i comuni divisori di b e c. In particolare le due coppie di numeri devono avere lo stesso massimo comune divisore: $\mathrm{mcd}(a, b) = \mathrm{mcd}(b, c)$.

Dimostrazione. Se $d|a$ e $d|b$, cioè $dx = a$ e $dy = b$ per qualche x, y, allora $c = a - qb = dx - qdy = d(x - qy)$ e vediamo che $d|c$.
E viceversa.

Calcoliamo $d := \mathrm{mcd}(7464, 3580)$:

$$\begin{aligned}
7464 &= 2{\cdot}3580 + 304 &&\Longrightarrow d = \mathrm{mcd}(3580, 304)\\
3580 &= 11{\cdot}304 + 236 &&\Longrightarrow d = \mathrm{mcd}(304, 236)\\
304 &= 1{\cdot}236 + 68 &&\Longrightarrow d = \mathrm{mcd}(236, 68)\\
236 &= 3{\cdot}68 + 32 &&\Longrightarrow d = \mathrm{mcd}(68, 32)\\
68 &= 2{\cdot}32 + 4 &&\Longrightarrow d = \mathrm{mcd}(32, 4)\\
32 &= 8{\cdot}4 + 0 &&\Longrightarrow d = \mathrm{mcd}(4, 0) = 4
\end{aligned}$$

Si vede che il massimo comune divisore è l'ultimo resto diverso da 0 nell'algoritmo euclideo. L'algoritmo in Python è molto semplice (dobbiamo però prima convertire i numeri negativi in positivi):

```
def mcd (a,b):
  if a<0: a=-a
  if b<0: b=-b
  while b: a,b=b,a%b
  return a
```

Altrettanto semplice è la versione ricorsiva:

```
def mcd (a,b):
  if a<0: a=-a
  if b<0: b=-b
  if b==0: return a
  return mcd(b,a%b)
```

dove usiamo la relazione

$$\mathrm{mcd}(a, b) = \begin{cases} a & \text{se } b = 0 \\ \mathrm{mcd}(b, a\%b) & \text{se } b > 0 \end{cases}$$

Sia $d = \mathrm{mcd}(a, b)$. Si può dimostrare che esistono sempre $x, y \in \mathbb{Z}$ tali che $d = ax + by$, seguendo ad esempio il seguente ragionamento ricorsivo. Se abbiamo

$$\begin{aligned} a &= \alpha b + c \\ d &= bx' + cy' \end{aligned}$$

allora

$$d = bx' + (a - \alpha b)y' = ay' + b(x' - \alpha y')$$

per cui $d = ax + by$ con $x = y'$ ed $y = x' - \alpha y'$. L'algoritmo euclideo esteso restituisce la tupla (d, x, y):

```
def mcde (a,b):
  if a<0: a=-a
  if b<0: b=-b
  if b==0: return a,1,0
  d,x,y=mcde(b,a%b); alfa=a/b
  return d,y,x-alfa*y
```

I numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$. Quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Un programma iterativo in Python per calcolare l'n-esimo numero di Fibonacci:

```
def fib (n):
  if n<=1: return 1
  a=b=1
  for k in xrange(n-1): a,b=a+b,a
  return a
```

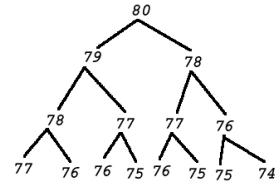
Possiamo visualizzare i numeri di Fibonacci da F_0 a F_9 e da F_{50} a F_{61} con le seguenti istruzioni:

```
for n in xrange(10): print fib(n),
# 1 1 2 3 5 8 13 21 34 55

print
for n in xrange(50,62): print fib(n)
# Output:

20365011074
32951280099
53316291173
86267571272
139583862445
225851433717
365435296162
591286729879
956722026041
1548008755920
2504730781961
4052739537881
```

La definizione stessa dei numeri di Fibonacci è di natura ricorsiva, e quindi sembra naturale usare invece una funzione ricorsiva:

```
def fibr (n):
  if n<=1: return 1
  return fibr(n-1)+fibr(n-2)
```

Se però adesso per la visualizzazione sostituiamo `fib` con `fibr`, ci accorgiamo che il programma si blocca nella seconda serie, cioè che anche un computer a 3 GHz non sembra in grado di calcolare F_{50}. Infatti qui incontriamo il fenomeno di una ricorsione con *sovrapposizione dei rami*, cioè una ricorsione in cui le operazioni chiamate ricorsivamente vengono eseguite molte volte.



Questo fenomeno è frequente nella ricorsione doppia e diventa chiaro se osserviamo l'illustrazione che mostra lo schema secondo il quale avviene il calcolo di F_{80}. Si vede che F_{78} viene calcolato due volte, F_{77} tre volte, F_{76} cinque volte, ecc. Si ha l'impressione che riappaia la successione di Fibonacci ed è proprio così, quindi un numero esorbitante di ripetizioni impedisce di completare la ricorsione. È infatti noto che

$$\left| F_n - \tfrac{1}{\sqrt{5}} \left(\tfrac{1+\sqrt{5}}{2}\right)^{n+1} \right| < \frac{1}{2}$$

per ogni $n \in \mathbb{N}$ e da ciò segue che questo algoritmo è di complessità esponenziale.

Il sistema di primo ordine

In analisi si impara che un'equazione differenziale di secondo ordine può essere ricondotta a un sistema di due equazioni di primo ordine. In modo simile possiamo trasformare la ricorrenza di Fibonacci in una ricorrenza di primo ordine in due dimensioni. Ponendo $x_n := F_n, y_n := F_{n-1}$ otteniamo il sistema

$$\begin{aligned} x_{n+1} &= x_n + y_n \\ y_{n+1} &= x_n \end{aligned}$$

per $n \geq 0$ con le condizioni iniziali $x_0 = 1, y_0 = 0$ (si vede subito che ponendo $F_{-1} = 0$ la relazione $F_{n+1} = F_n + F_{n-1}$ vale per $n \geq 1$). Per applicare questo algoritmo dobbiamo però in ogni passo generare due valori, avremmo quindi bisogno di una funzione che restituisce due valori numerici. Ciò in Python, dove una funzione può restituire come risultato una lista, è molto facile:

```
def fibs (n):
  if n==0: return [1,0]
  (a,b)=fibs(n-1); return [a+b,a]
```

Per la visualizzazione usiamo le istruzioni

```
for n in xrange(50,61):
  print fibs(n)[0]
```

ottenendo in modo fulmineo il risultato.

La lista dei numeri di Fibonacci

Per ottenere i numeri di Fibonacci da F_0 ad F_n come lista, possiamo usare la funzione

```
def fib (n):
  v=[0,1]
  for i in xrange(n-1):
    v.append(v[-2]+v[-1])
  return v
```

Si noti che non si tratta di una ricorsione doppia (infatti la funzione stessa non è ricorsiva) e l'esecuzione è velocissima.

Conversione di numeri

La funzione `int` può essere usata in vari modi. Se x è un numero, `int(x)` è l'intero che si ottiene da x per arrotondamento verso lo zero (quindi -13.2 diventa -13). Se a è una stringa, bisogna indicare un numero intero b compreso tra 2 e 36 e allora `int(a,base=b)` è l'intero rappresentato in base b dalle cifre che compongono la stringa a, interpretate nel modo usato anche da noi per `rappcomestringa`. Se la stringa inizia con `-`, viene calcolato il corrispondente intero negativo. In questa seconda forma `int(a,base=b)` è probabilmente equivalente a `string.atoi(a,base=b)`. Esempi:

```
for x in (3.5,4,6.9,-3.7): print int(x),
# 3 4 6 -3

print
print int('-88345',base=10)
# -88345

print int('kvjm',base=36)
# 974002

print string.atoi('kvjm',base=36)
# 974002
```

`float` converte un numero o una stringa adatta a un numero a virgola mobile:

```
for x in (3,8.88,'6.25','-40'):
  print float(x),
# 3.0 8.88 6.25 -40.0

x=100; y=17
print x/y, float(x)/y
# 5 5.88235294118
```

Le funzioni `bool` e `str` sono state introdotte alle pagine 13 e 16.

Nella funzione `cesare` a pagina 8 e in `rappcomestringa` abbiamo utilizzato gli operatori di conversione `ord` (codice ASCII di un carattere) e `chr` (carattere che corrisponde a un codice ASCII).

`hex` e `oct` forniscono le rappresentazioni esadecimali e ottali di un numero intero nel formato talvolta utilizzato da Python:

```
for x in (15,92,187): print hex(x),
# 0xf 0x5c 0xbb
print
for x in (9,13,32): print oct(x),
# 011 015 040
```

Funzioni matematiche di base

Per calcolare il massimo di una sequenza `a` si può usare `max(a)`. Alternativamente questa funzione può essere usata anche con argomenti multipli, separati da virgole, di cui calcola il massimo. Nello stesso modo si usa `min` per il minimo.

```
a=[3,5,0,2,12,7,33,6,2]
print max(a)
# 33

print max(3,5,0,2,12,7,33,6,2)
# 33
print max('alfa','beta',17)
# beta

print min(3,8,[4,7],'otto')
# 3

print max([[3,6,2],[1,5,10]])
# [3,6,2]

print min([2,0,4],[1,7,15,18])
# [1,7,15,18]
```

`abs(x)` restituisce il valore assoluto del numero reale o complesso `x`:

```
for x in (7,-3,5+1j): print abs(x),
# 7 3 5.09901951359
```

Con `round(x)` si ottiene un arrotondamento del numero `x` al più vicino intero. Per $x \in \frac{1}{2}\mathbb{Z}$ tra $x - \frac{1}{2}$ e $x + \frac{1}{2}$ viene scelto l'intero più lontano dallo zero:

```
for x in (1.4, 1.5, 1.6):
  print round(x),
# 1.0 2.0 2.0

print
for x in (-1.4, -1.5, -1.6):
  print round(x),
# -1.0 -2.0 -2.0
```

`round` può essere anche usato con un secondo argomento che indica il numero di cifre decimali a cui si vuole arrotondare:

```
print round(1.68454,2)
# 1.68

print round(1.685,2)
# 1.69

print round(-1.68454,2)
# -1.68

print round(-1.685,2)
# -1.69
```

La potenza x^a può essere calcolata come `x**a` oppure con `pow(x,a)`. Nell'aritmetica intera si calcolano talvolta potenze in $\mathbb{Z}/m$: `pow(x,n,m)` è equivalente a `pow(x,n)%m`, ma più veloce.

```
for n in xrange(2,5):
  print pow(5,n,17),
# 8 6 13

print pow(math.e,math.log(2))
# 2.0
```

Con `divmod(a,m)` si ottiene la tupla `(a/m,a%m)`, anche in questo caso per numeri negativi calcolata come in C, cioè diversamente da quanto si farebbe in matematica (cfr. pagina 14):

```
print divmod(30,11)
# (2, 8)
for x in (30/11,30%11): print x,
# 2 8

print
print divmod(30,-11)
# (-3, -3)
for x in (30/(-11),30%(-11)): print x,
# -3 -3
```

Potremmo usare questa funzione per abbreviare leggermente la funzione `rapp` a pagina 24:

```
def rapp (n,base):
  if n<base: return [n]
  q,r=divmod(n,base)
  return rapp(q,base)+[r]
```

Numeri complessi

`complex(real=a,imag=b)` restituisce il numero complesso $a + bi$. I parametri `real` e `imag` sono preimpostati a zero. Invece di numeri `a` e `b` si può anche utilizzare come unico argomento una stringa che rappresenta il numero complesso nel formato previsto da Python:

```
print complex(7)
# (7+0j)
print complex(imag=8)
# 8j

print complex(imag=8,real=1)
# (1+8j)

print complex('7+4j')
# (7+4j)
```

La parte reale e la parte immaginaria di un numero complesso `z` si ottengono con `z.real` e `z.imag`, il coniugato complesso con `z.conjugate()`:

```
z=3+2j

print z.real
# 3.0

print z.imag
# 2.0

print z.conjugate()
# (3-2j)
```

Gli operatori aritmetici possono essere applicati direttamente in espressioni che contengono numeri complessi:

```
z=4+5j; w=6-1j
print z+w, z-w, z*w
print z/w
print pow(z,3)
# Output:

(10+4j) (-2+6j) (29+26j)
(0.513513513514+0.918918918919j)
(-236+115j)
```

Il modulo math

Questo modulo contiene le seguenti funzioni e costanti, corrispondenti per la maggior parte ad analoghe funzioni del C.

Funzioni trigonometriche: `math.cos`, `math.sin` e `math.tan`.

Funzioni trigonometriche inverse: `math.acos`, `math.asin`, `math.atan` e `math.atan2`.

`math.atan2(y,x)` calcola l'angolo nella rappresentazione polare di un punto $(x, y) \neq (0, 0)$ nel piano. Attenti all'ordine degli argomenti!

```
print math.degrees(math.atan2(1,0))
# 90

print math.degrees(math.atan2(1,-1))
# 135.0

print math.degrees(math.atan2(-1,-1))
# -135.0
```

Funzioni per la conversione di angoli a gradi: `math.degrees(alfa)` esprime l'angolo `alfa` in gradi, `math.radians(gradi)` calcola il valore in radianti di un angolo espresso in gradi.

```
for x in (90,135,180,270,360):
  print math.radians(x)
# Output:

1.57079632679
2.35619449019
3.14159265359
4.71238898038
6.28318530718

for x in (1.6,2.4,3.1,4.7,6.3):
  print math.degrees(x)
# Output:
91.6732472209
137.509870831
177.616916491
269.290163711
360.963410932
```

Funzioni iperboliche: `math.cosh`, `math.sinh` e `math.tanh`.

Ricordiamo le definizioni di queste importanti funzioni:

$$\cosh x := \frac{e^x + e^{-x}}{2}$$

$$\sinh x := \frac{e^x - e^{-x}}{2}$$

$$\tanh x := \frac{\sinh x}{\cosh x} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Esponenziale e logaritmi: `math.exp`, `math.pow`, `math.log`, `math.log10`, `math.ldexp`, `math.sqrt` e `math.hypot`.

Il logaritmo naturale di x lo si ottiene con `math.log(x)`, il logaritmo in base b con `math.log(x,b)`, il logaritmo in base 10 anche con `math.log10(x)`. `ldexp(a,n)` è uguale ad $a2^n$ e difficilmente ne avremo bisogno. Abbiamo già visto che `math.sqrt` calcola la radice di un numero reale ≥ 0. Con `math.hypot(x,y)` otteniamo $\sqrt{x^2 + y^2}$. Invece di `math.pow` possiamo usare `pow`.

```
for x in (0,1,2,math.log(7)):
  print math.exp(x),
# 1.0 2.71828182846 7.38905609893 7.0

print
print math.ldexp(1,6)
# 64.0
print math.ldexp(2.5,6)
# 160.0

for b in (2,math.e,7,10):
  print round(math.log(10,b),4),
# 3.3219 2.3026 1.1833 1.0

print math.log(1024,2)
# 10.0
```

Alcune funzioni aritmetiche: `math.floor`, `math.ceil`, `math.fmod`.

`math.floor(x)` restituisce la parte intera del numero reale x, cioè il più vicino intero alla sinistra di x, `math.ceil` il più vicino intero alla destra di x. Entrambe le funzioni restituiscono valori reali e funzionano correttamente anche per argomenti negativi:

```
for x in (-1.3,-4,0.5,2,2.7):
  print math.floor(x), math.ceil(x)
# Output:

-2.0 -1.0
-4.0 -4.0
 0.0  1.0
 2.0  2.0
 2.0  3.0
```

Le funzioni `math.modf`, `math.fmod` e `math.frexp` non funzionano correttamente.

`math.fabs(x)` è il valore assoluto di x, ma possiamo usare la funzione `abs`.

Le costanti `math.e` e `math.pi` rappresentano i numeri e e π:

```
print math.e
# 2.71828182846

print math.pi
# 3.14159265359
```

In entrambi i casi l'ultima cifra è arrotondata in alto.

Il modulo cmath

Questo modulo fornisce (con il prefisso `cmath.`) le funzioni `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`, `exp`, `log`, `log10` e `sqrt` per argomenti complessi:

```
for k in xrange(1,8):
  print cmath.sin(k*math.pi*1j)
# Output:

11.5487393573j
267.744894041j
6195.82386361j
143375.656567j
3317811.99967j
76776467.6977j
1776660640.42j
```

Si vede chiaramente che sulla retta $\mathbb{R}i$ il seno cresce in modo esponenziale.

apply

f sia una funzione di n argomenti, dove n può essere anche variabile, e v una sequenza di lunghezza n. Allora `apply(f,v)` è uguale ad $f(v_1, \ldots, v_n)$. Con un'abbreviazione sintattica introdotta di recente si può scrivere semplicemente `f(*v)`. Esempi:

```
def somma (*a):
  s=0
  for x in a: s+=x
  return s

v=[1,2,4,9,2,8]
s=apply(somma,v); t=somma(*v)
print s,t # 26, 26
```

---

reduce

f sia una funzione di due argomenti. Come in algebra scriviamo $a \cdot b := f(a, b)$. Per una sequenza $v = (a_1, \ldots, a_n)$ di lunghezza $n \geq 1$ l'espressione $\mathrm{reduce}(f, v)$ è definita come il prodotto (in genere non associativo) da sinistra verso destra degli elementi di v:

$$\begin{aligned}
\mathrm{reduce}(f, [a_1]) &= a_1 \\
\mathrm{reduce}(f, [a_1, a_2]) &= a_1 \cdot a_2 \\
\mathrm{reduce}(f, [a_1, a_2, a_3]) &= (a_1 \cdot a_2) \cdot a_3 \\
&\ldots \\
\mathrm{reduce}(f, [a_1, \ldots, a_{n+1}]) &= (a_1 \cdots a_n) \cdot a_{n+1}
\end{aligned}$$

`reduce` può essere anche usata con un argomento iniziale a_0; in questo caso il valore è definito semplicemente mediante

$$\mathrm{reduce}(f, [a_1, \ldots, a_n], a_0) := \mathrm{reduce}(f, [a_0, a_1, \ldots, a_n])$$

In genetica si usa la composizione non associativa $a \cdot b := \dfrac{a+b}{2}$:

```
def f(x,y): return (x+y)/2.0

print reduce(f,[3]), reduce(f,[3,4]),
print reduce(f,[3,4,5]),
print reduce(f,[3,4,5,8])
# 3 3.5 4.25 6.125
```

Se, fissato x, nello schema di Horner definiamo $b \cdot a := bx + a$, possiamo riprogrammare l'algoritmo mediante `reduce`:

```
def hornerr (a,x):
  def f(u,v): return x*u+v
  return reduce(f,a,0)

a=[7,2,3,5,4]; print hornerr(a,10)
# 72354
```

Se per numeri reali $a, b > 0$ definiamo $a \cdot b := \dfrac{1}{a} + b$, otteniamo le frazioni continue, con gli argomenti invertiti nell'ordine:

```
def frazcont (v):
  def f(x,y): return y+1.0/x
  w=v[:]; w.reverse()
  return reduce(f,w)

print frazcont([2,3,1,5])
# 2.26086956522 = 52/23.0
```

Espressioni lambda

L'espressione `lambda x: f(x)` corrisponde all'espressione `function (x) f(x)` di R; con più variabili diventa `lambda x,y: f(x,y)`. A differenza da R però in Python `f(x)` deve essere un'espressione unica che soprattutto non può contenere istruzioni di assegnamento o di controllo; manca anche il `return`. Ciononostante il valore di un'espressione lambda è una funzione che può essere usata come valore di un'altra funzione:

```
def u(x): return x**2

def v(x): return 4*x+1

def comp (f,g):
  return lambda x: f(g(x))

print comp(u,v)(5)
# 441
```

Qui abbiamo ridefinito la funzione `comp` vista a pagina 10. Possiamo anche assegnare un nome alla funzione definita da un'espressione lambda:

```
f=lambda x,y: x+y-1

print f(6,2)
# y
```

Come abbiamo visto a pagina 16, i `lambda` possono essere annidati. Un'espressione lambda senza variabile è una funzione costante:

```
def costante (x): return lambda : x

f=costante(10); print f()
# 10
```

Espressioni lambda vengono spesso usate come argomenti di altre funzioni, ad esempio possiamo creare un filtro che restituisce i numeri pari di una successione di interi senza definire un'apposita funzione `pari` (cfr. pagina 14):

```
print filter(lambda x: x%2==0,
  xrange(15))
# [0, 2, 4, 6, 8, 10, 12, 14]
```

Per uno spazio vettoriale V possiamo definire un'iniezione canonica

$$j := \bigcirc_{\alpha} \alpha(v) : V \longrightarrow V''$$

di V nel suo doppio duale; se V è di dimensione finita, j è un isomorfismo. Possiamo imitare j in Python con

```
def incan (v):
  def veval (a): return a(v)
  return veval
```

oppure più brevemente con

```
def incan (v):
  return lambda a: a(v)
```

Studiare attentamente questo esempio:

```
def alfa (v):
  (x,y)=v; return x+y

print incan([2,7])(alfa)
# 9
```

Il λ-calcolo

Siccome in matematica bisogna distinguere tra la funzione f e i suoi valori $f(x)$, nel corso di Algoritmi abbiamo introdotto la notazione $\bigcirc_x f(x)$ per la funzione che manda x in $f(x)$. Ad esempio $\bigcirc_x \sin(x^2+1)$ è la funzione che manda x in $\sin(x^2+1)$. È chiaro che ad esempio $\bigcirc_x x^2 = \bigcirc_y y^2$ (per la stessa ragione per cui $\sum_{i=0}^{n} a_i = \sum_{j=0}^{n} a_j$), mentre $\bigcirc_x xy \neq \bigcirc_y yy$ (così come $\sum_i a_{ij} \neq \sum_j a_{jj}$) e, come non ha senso l'espressione $\sum_i \sum_i a_{ii}$, così non ha senso $\bigcirc_x \bigcirc_x x$. Siccome in logica si scrive $\lambda x.f(x)$ invece di $\bigcirc_x f(x)$, la teoria molto complicata di questo operatore si chiama λ-calcolo. Su esso si basano il Lisp e molti concetti dell'intelligenza artificiale.

H. Barendregt: The lambda calculus. Elsevier 1990.

Matrici da vettori

Per trasformare una sequenza v in una matrice con m colonne possiamo utilizzare la seguente funzione:

```
# m e' il numero di colonne.
def matricedavettore (v,m):
  return map(lambda i: v[m*i:m*i+m],
    xrange(len(v)/m))

v=[0,1,2,3,4,5]
A=matricedavettore(v,2)
print A
# [[0, 1], [2, 3], [4, 5]]

B=zip(*A)
print B
# [(0, 2, 4), (1, 3, 5)]

C=map(list,zip(*A))
print C
# [[0, 2, 4], [1, 3, 5]]

v=[0,1,2,3,4,5,6,7]
A=matricedavettore(v,4)
print A
# [[0, 1, 2, 3], [4, 5, 6, 7]]
```

Come si vede nell'esempio, se usiamo `zip` per costruire la trasposta di una matrice, e se vogliamo che anche le righe della trasposta siano liste e non tuple, dobbiamo effettuare una conversione mediante `list`.

Non è necessario che i coefficienti del vettore siano numeri:

```
def matricedavettore (v,m):
  return map(lambda i: v[m*i:m*i+m],
    xrange(len(v)/m))

v=['Roma',2600,'Pisa',92,'Padova',210]
T=matricedavettore(v,2)
for x in T: print x
# Output:

['Roma', 2600]
['Pisa',92]
['Padova', 210]
```

Concatenazione di più liste

Usiamo il metodo `extend` definito per le liste per creare una funzione che concatena un numero arbitrario di liste:

```
def concatl (*v):
  a=[]
  for x in v: a.extend(x)
  return a

a=[0,1,2,3,4,5]
b=[6,7,8]
c=[9,10,11,12]

print concatl(a,b,c)
# [0,1,2,3,4,5,6,7,8,9,10,11,12]
```

Linearizzare una lista annidata

Per linearizzare una lista annidata possiamo usare la seguente funzione:

```
def lineare (a):
  v=[]
  for x in a:
    if isinstance(x,(list,tuple)):
      v.extend(lineare(x))
    else: v.append(x)
  return v
```

Studieremo la funzione `isinstance` più in dettaglio nell'ambito della programmazione orientata agli oggetti; qui evidentemente viene utilizzata per verificare se un oggetto è una lista o una tupla. Esempio:

```
a=[(1,3),5,[6,[8,1,2],5],8]

print lineare(a)
# [1, 3, 5, 6, 8, 1, 2, 5, 8]
```

Impostare il limite di ricorsione

Se definiamo il fattoriale con

```
def fatt (n):
  if n==0: return 1
  return n*fatt(n-1)
```

riusciamo a calcolare con `fatt(998)` il fattoriale 998!, mentre il programma termina con un errore se proviamo `fatt(999)`. Abbiamo infatti superato il limite di ricorsione, più precisamente dello stack utilizzato per l'esecuzione delle funzioni, inizialmente impostato a 1000. Si può ridefinire questo limite con la funzione `sys.setrecursionlimit`:

```
sys.setrecursionlimit(2000)

print fatt(1998)
# Funziona.
```

Per sapere il limite di ricorsione attuale si può usare la funzione `sys.getrecursionlimit`:

```
print sys.getrecursionlimit()
# 1000 (se non reimpostato)

sys.setrecursionlimit(2000)
print sys.getrecursionlimit()
# 2000
```

Le torri di Hanoi

Abbiamo tre liste `a`, `b` e `c`. All'inizio `b` e `c` sono vuote, mentre `a` contiene i numeri $0, 1, \ldots, N-1$ in ordine strettamente ascendente. Vogliamo trasferire tutti i numeri da `a` in `b`, seguendo queste regole:

(1) Si possono utilizzare tutte e tre le liste nelle operazioni.
(2) In ogni passo si può trasferire solo l'elemento più in alto (cioè l'elemento più piccolo) di una lista a un altra, anteponendola agli elementi di questa seconda lista.
(3) In ogni passo, gli elementi di ciascuna delle tre liste devono rimanere in ordine naturale.

Si tratta di un gioco inventato dal matematico francese Edouard Lucas (1842-1891), noto anche per i suoi studi sui numeri di Fibonacci e per un test di primalità in uso ancora oggi. Nell'interpretazione originale `a`, `b` e `c` sono aste con una base. All'inizio `b` e `c` sono vuote, mentre su `a` sono infilati dei dischi perforati in ordine crescente dall'alto verso il basso, in modo che il disco più piccolo si trovi in cima. Bisogna trasferire tutti i disci sul paletto `b`, usando tutte e tre le aste, ma trasferendo un disco alla volta e in modo che un disco più grando non si trovi mai su uno più piccolo.

L'algoritmo più semplice è ricorsivo:

(1) Poniamo $n = N$.
(2) Trasferiamo i primi $n-1$ numeri da a all'inizio di c.
(3) Poniamo il primo numero di a all'inizio di b.
(4) Trasferiamo i primi $n-1$ numeri di c all'inizio di b.

In Python definiamo la seguente funzione, che contiene anche un comando di stampa affinché il contenuto delle tre liste venga visualizzato dopo ogni passaggio:

```
def hanoi (a,b,c,n=None):
  print a,b,c
  if n==None: n=len(a)
  if n==1: b.insert(0,spezza(a))
  else:
    hanoi(a,c,b,n-1); hanoi(a,b,c,1)
    hanoi(c,b,a,n-1)
```

Abbiamo usato la funzione `spezza` che verrà definita a pagina 27.

```
a=range(0,4); b=[]; c=[]
hanoi(a,b,c)
print a,b,c
# Output:

[0, 1, 2, 3] [] []
[0, 1, 2, 3] [] []
[0, 1, 2, 3] [] []
[0, 1, 2, 3] [] []
[1, 2, 3] [] [0]
[0] [1] [2, 3]
[2, 3] [] [0, 1]
[0, 1] [2] [3]
[0, 1] [3] [2]
[1] [2] [0, 3]
[0, 3] [1, 2] []
[3] [] [0, 1, 2]
[0, 1, 2] [3] []
[0, 1, 2] [] [3]
[0, 1, 2] [3] []
[1, 2] [] [0, 3]
[0, 3] [1] [2]
[2] [3] [0, 1]
[0, 1] [2, 3] []
[0, 1] [] [2, 3]
[1] [2, 3] [0]
[0] [1, 2, 3] []
[] [0, 1, 2, 3] []
```

Minilisp

Il Lisp è un linguaggio molto vasto che si basa però su un'idea semplice, benché estremamente efficace: Un programma in Lisp è una lista i cui elementi sono dati atomari (cioè dati che non sono liste) oppure altre liste. Se il primo elemento di una lista è una funzione, viene calcolato il valore di questa funzione sugli altri elementi della stessa lista che sono considerati suoi argomenti. Possiamo saggiare la potenza delle operazioni in Python fin qui discusse, definendo una funzione `esegui` che, in modo ancora primitivo, corrisponde a questa idea. Si noti come in questo esempio si rivela utile la funzione `apply`, per la quale usiamo ancora la sintassi abbreviata.

```
import types

def esegui (a):
  if type(a)!=types.ListType: return a
  f=a[0]
  if type(f)!=types.FunctionType:
    # f non e' una funzione.
    return map(esegui,a)
  return f(*map(esegui,a[1:]))

def somma (*a):
  s=0
  for x in a: s+=x
  return s

def prodotto (*a):
  p=1
  for x in a: p*=x
  return p

programma = [somma,5,[prodotto,2,3,7],8]
print esegui(programma)
# 55
```

Numeri casuali

`random.randint(1,6)` fornisce un numero intero casuale tra 1 e 6.

`random.random()` fornisce un numero reale casuale in $[0, 1]$.

```
for k in xrange(10):
  print random.randint(1,6),
print
# 5 5 2 5 6 5 3 4 2 6

for k in xrange(10):
  print random.randint(1,6),
print
# 2 5 3 6 2 6 6 3 4 1

for k in xrange(3):
  print round(random.random(),2),
print
# 0.89 0.01 0.43
```

L'algoritmo del contadino russo

Esiste un algoritmo leggendario del contadino russo per la moltiplicazione di due numeri, uno dei quali deve essere un numero naturale. Nella formulazione matematica ricorsiva questo algoritmo si presenta nel modo seguente. Sia f la funzione di due variabili definita da $f(x, n) := nx$. Allora

$$f(x,n) = \begin{cases} 0 & \text{se } n = 0 \\ f(2x, n/2) & \text{se } n \text{ è pari } \neq 0 \\ x + f(x, n-1) & \text{se } n \text{ è dispari} \end{cases}$$

In Python ciò corrisponde alla funzione

```
# Moltiplicazione russa.
def f (x,n):
  if n==0: return 0
  if n%2==0: return f(x+x,n/2)
  return x+f(x,n-1)
```

Se questa funzione la inseriamo nel file *russo.py*, nel file *alfa* che funge da programma principale possiamo scrivere

```
import russo

x=10; n=87
print russo.f(x,n)
# Output: 870
```

Naturalmente il prodotto nx die due numeri in Python lo otteniamo più semplicemente con `n*x`. L'algoritmo può però essere utile in un contesto di calcolo (in un $\mathbb{N}$-*modulo*, come si dice in algebra, ad esempio in un *gruppo abeliano*) per il quale il Python non fornisce direttamente una funzione per la moltiplicazione con fattori interi. Lo stesso vale per il calcolo di potenze x^n (che, per numeri, in Python si ottengono con `x**n`). Anche qui esiste un algoritmo del contadino russo che può essere formulato così: Sia g la funzione di 2 variabili definita da $g(x, n) := x^n$. Allora

$$g(x,n) = \begin{cases} 1 & \text{se } n = 0 \\ g(x^2, n/2) & \text{se } n \text{ è pari } \neq 0 \\ x \cdot g(x, n-1) & \text{se } n \text{ è dispari} \end{cases}$$

In Python definiamo la funzione nel modo seguente:

```
# Potenza russa.
def g(x,n):
  if n==0: return 1
  if n%2==0: return g(x*x,n/2)
  return x*g(x,n-1)
```

Se questa funzione la inseriamo nel file *russo.py*, nel file *alfa* che funge da programma principale possiamo scrivere

```
x=2; n=16
print russo.g(x,n)
# Output: 65536

x=2; n=32
print russo.g(x,n)
# Output: 4294967296
```

Confrontando i due casi, ci si accorge che l'algoritmo è sempre lo stesso - cambia soltanto l'operazione di $\mathbb{N}$ sugli argomenti!

VI. LO SCHEMA DI HORNER

Rappresentazione binaria

Ogni numero naturale n possiede una rappresentazione binaria, cioè una rappresentazione della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_1 2 + a_0$$

con coefficienti (o cifre binarie) $a_i \in \{0, 1\}$. Per $n = 0$ usiamo $k = 0$ ed $a_0 = 0$; per $n > 0$ chiediamo che $a_k \neq 0$. Con queste condizioni k e gli a_i sono univocamente determinati. Sia $r_2(n) = (a_k, \ldots, a_0)$ il vettore i cui elementi sono queste cifre. Dalla rappresentazione binaria si deduce la seguente relazione ricorsiva:

$$r_2(n) = \begin{cases} (n) & \text{se } n \leq 1 \\ (r_2(\frac{n}{2}), 0) & \text{se } n \text{ è pari} \\ (r_2(\frac{n-1}{2}), 1) & \text{se } n \text{ è dispari} \end{cases}$$

È molto facile tradurre questa idea in una funzione di Python. Siccome Python per operandi interi esegue una divisione intera, anche per n dispari possiamo scrivere `n/2`, in tal caso automaticamente uguale a `(n-1)/2`.

```
def rapp2pv (n): # Prima versione.
  if n<=1: v=[n]
  else:
    v=rapp2pv(n/2)
    if n%2==0: v.append(0)
    else: v.append(1)
  return v
```

La versione definitiva della funzione prevede un secondo parametro facoltativo `cifre`; quando questo è maggiore del numero di cifre necessarie per la rappresentazione binaria di n, i posti iniziali vuoti vengono riempiti con zeri.

```
def rapp2 (n,cifre=0):
  if n<=1: v=[n]
  else:
    v=rapp2(n/2)
    if n%2==0: v.append(0)
    else: v.append(1)
  d=cifre-len(v)
  if d>0: v=[0]*d+v
  return v
```

Per provare la funzione usiamo la possibilità di costruire una stringa da una lista di numeri mediante la funzione `str`, come abbiamo visto in precedenza, ottenendo l'output

```
  0  0 0 0 0 0 0 0 0
  1  0 0 0 0 0 0 0 1
  2  0 0 0 0 0 0 1 0
  3  0 0 0 0 0 0 1 1
  4  0 0 0 0 0 1 0 0
  5  0 0 0 0 0 1 0 1
  6  0 0 0 0 0 1 1 0
  7  0 0 0 0 0 1 1 1
  8  0 0 0 0 1 0 0 0
  9  0 0 0 0 1 0 0 1
 10  0 0 0 0 1 0 1 0
 11  0 0 0 0 1 0 1 1
 12  0 0 0 0 1 1 0 0
 19  0 0 0 1 0 0 1 1
 48  0 0 1 1 0 0 0 0
 77  0 1 0 0 1 1 0 1
106  0 1 1 0 1 0 1 0
135  1 0 0 0 0 1 1 1
164  1 0 1 0 0 1 0 0
194  1 1 0 0 0 0 1 0
221  1 1 0 1 1 1 0 1
```

con

```
def strdalista (a,sep=' '):
  return sep.join(map(str,a))

for n in range(13) + \
    [19,48,77,106,135,164,194,221]:
  print "%3d  %s" \
    %(n,strdalista(rapp2(n,cifre=8)))
```

Se usiamo invece

```
for n in xrange(256):
  print "%3d  %s" \
    %(n,strdalista(rapp2(n,cifre=8)))
```

otteniamo gli elementi dell'ipercubo 2^8 (cfr. pagina 25).

Una rappresentazione binaria a lunghezza fissa di valori numerici viene anche usata nella trasmissione di segnali, ad esempio dei valori di grigio di un'immagine bianco-nera satellitare.

Rappresentazione b-adica

Sia $b \in \mathbb{N}+2$. Allora ogni numero naturale n possiede una rappresentazione b-adica, cioè una rappresentazione della forma

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0$$

con coefficienti $a_i \in \{0, 1, \ldots, b-1\}$. Per $b = 2$ otteniamo la rappresentazione binaria, per $b = 16$ la rappresentazione esadecimale. Di nuovo per $n = 0$ usiamo $k = 0$ ed $a_0 = 0$; per $n > 0$ chiediamo che $a_k \neq 0$. Con queste condizioni k e gli a_i sono univocamente determinati. Per calcolare questa rappresentazione (in forma di una lista di numeri $[a_k, \ldots, a_0]$) osserviamo che per $n < b$ la rappresentazione b-adica coincide con $[n]$, mentre per $n \geq b$ la otteniamo eseguendo la divisione intera $q = n/b$ e aggiungendo il resto $n\%b$ alla rappresentazione b-adica di q. Ad esempio si ottiene la rappresentazione 10-adica di $n = 7234$ aggiungendo il resto 4 di n modulo 10 alla rappresentazione 10-adica $[7, 2, 3]$ di 723. Possiamo facilmente tradurre questo algoritmo in una funzione in Python, il cui secondo argomento è la base b:

```
def rapp (n,base):
  if n<base: return [n]
  q,r=n/base,n%base
  return rapp(q,base)+[r]
```

Naturalmente questa funzione per $b = 2$ potrebbe sostituire la `rapp2pv` definita in precedenza. Definiamo inoltre una funzione che per $b \leq 36$ trasforma la lista numerica ottenuta in una stringa, imitando l'idea utilizzata per i numeri esadecimali, sostituendo cioè le cifre da 10 a 35 con le lettere `a,...,z`:

```
def rappcomestringa (n,base):
  v=rapp(n,base)
  def codifica (x):
    if x<10: return str(x)
    return chr(x-10+ord('a'))
  return ''.join(map(codifica,v))
```

Esempi:

```
for x in (8,12,24,60,80,255,256):
  print rappcomestringa(x,16),
# 8 c 18 3c 50 ff 100
print
print rappcomestringa(974002,36)
# kvjm
```

È un comodo metodo per ricordarsi il numero telefonico del Dipartimento di Matematica o un qualsiasi altro numero telefonico che non inizia con 0. Ma bisogna saper tornare indietro:

```
def rappcomenumero (a,base):
  def decodifica(x):
    s=ord(x)
    if 48<=s<=57: return s-48
    return s-87
  v=map(decodifica,a)
  return horner(v,base)

print rappcomenumero('kvjm',36)
# 974002
```

Numeri esadecimali

Nei linguaggi macchina e assembler molto spesso si usano i numeri esadecimali o, più correttamente, la rappresentazione esadecimale dei numeri naturali, cioè la loro rappresentazione in base 16.

Per $2789 = 10 \cdot 16^2 + 14 \cdot 16 + 5 \cdot 1$ potremmo ad esempio scrivere

`2789 = (10,14,5)`$_{16}$.

In questo senso `10`,`14` e `5` sono le cifre della rappresentazione esadecimale di `2789`. Per poter usare lettere singole per le cifre si indicano le cifre `10`,...,`15` mancanti nel sistema decimale nel modo seguente:

10	A
11	B
12	C
13	D
14	E
15	F

In questo modo adesso possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che `(F)`$_{16}$ `= 15` assume nel sistema esadecimale lo stesso ruolo come il 9 nel sistema decimale. Quindi `(FF)`$_{16}$ `= 255` $= 16^2 - 1 = 2^8 - 1$. Un numero naturale n con $0 \leq n \leq 255$ si chiama un *byte*, un *bit* è invece uguale a 0 o a 1.

Esempi:

	0	$(0)_{16}$
	14	$(E)_{16}$
	15	$(F)_{16}$
	16	$(10)_{16}$
	28	$(1C)_{16}$
2^5	32	$(20)_{16}$
2^6	64	$(40)_{16}$
	65	$(41)_{16}$
	97	$(61)_{16}$
	127	$(7F)_{16}$
2^7	128	$(80)_{16}$
	203	$(CB)_{16}$
	244	$(F4)_{16}$
	255	$(FF)_{16}$
2^8	256	$(100)_{16}$
2^{10}	1024	$(400)_{16}$
2^{12}	4096	$(1000)_{16}$
	65535	$(FFFF)_{16}$
2^{16}	65536	$(10000)_{16}$

Si vede da questa tabella che i byte sono esattamente quei numeri per i quali sono sufficienti due cifre esadecimali.

L'ipercubo

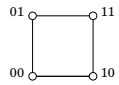
Sia $X = \{1, \ldots, n\}$ con $n \geq 0$.

Identificando $\mathcal{P}(X)$ con 2^n, geometricamente otteniamo un *ipercubo* che può essere visualizzato nel modo seguente.

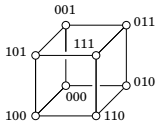
$n = 0$:

$n = 1$:

$n = 2$:



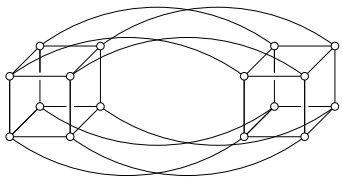
$n = 3$:



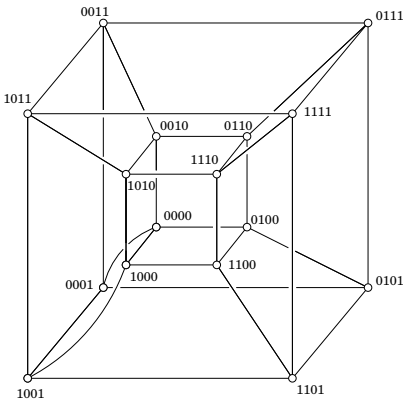
$n = 4$: L'ipercubo a 4 dimensioni si ottiene dal cubo 3-dimensionale attraverso la relazione

$$2^4 = 2^3 \times \{0, 1\}$$

Dobbiamo quindi creare due copie del cubo 3-dimensionale. Nella rappresentazione grafica inoltre sono adiacenti e quindi connessi con una linea quei vertici che si distinguono in una sola coordinata. Oltre ai legami all'interno dei due cubi dobbiamo perciò unire i punti $(x, y, z, 0)$ e $(x, y, z, 1)$ per ogni x, y, z.



La figura diventa molto più semplice, se si pone uno dei due cubi (quello con la quarta coordinata $= 0$) all'interno dell'altro (quello con la quarta coordinata $= 1$):



$n \geq 5$: Teoricamente anche qui si può usare la relazione

$$2^n = 2^{n-1} \times \{0, 1\}$$

ma la visualizzazione diventa difficoltosa.

Ogni vertice dell'ipercubo corrisponde a un elemento di 2^n che nell'interpretazione insiemistica rappresenta a sua volta un sottoinsieme di X (il punto 0101 ad esempio l'insieme $\{2, 4\}$ se $X = \{1, 2, 3, 4\}$).

La distanza di Hamming

La distanza di Hamming è definita come il numero delle coordinate in cui due elementi di 2^n differiscono e che in Python può essere calcolata con la seguente semplicissima funzione:

```
def hamming (a,b):
  return map(lambda x,y:
    x==y and 1 or 0, a,b).count(1)

a=(0,1,0,1,0,1,1,1,0,1)
b=(1,0,0,1,0,0,1,1,1,0)

print hamming(a,b) # 5
```

Questa distanza ha le proprietà di una metrica ed è molto utilizzata nella teoria dei codici.

Lo schema di Horner

Sia dato un polinomio

$$f = a_0 x^n + a_1 x^{n-1} + \cdots + a_n \in A[x]$$

dove A è un qualsiasi anello commutativo.

Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$.
Sia ad esempio $f = 3x^4 + 5x^3 + 6x^2 + 8x + 7$. Poniamo

$$\begin{aligned}
b_0 &= 3 \\
b_1 = b_0\alpha + 5 &= 3\alpha + 5 \\
b_2 = b_1\alpha + 6 &= 3\alpha^2 + 5\alpha + 6 \\
b_3 = b_2\alpha + 8 &= 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\
b_4 = b_3\alpha + 7 &= 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 7
\end{aligned}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{aligned}
b_0 &= a_0 \\
b_1 &= b_0\alpha + a_1 \\
&\cdots \\
b_k &= b_{k-1}\alpha + a_k \\
&\cdots \\
b_n &= b_{n-1}\alpha + a_n
\end{aligned}$$

con $b_n = f(\alpha)$, come dimostriamo adesso. Consideriamo il polinomio

$$g := b_0 x^{n-1} + b_1 x^{n-2} + \cdots + b_{n-1}.$$

Allora, usando che $\alpha b_k = b_{k+1} - a_{k+1}$ per $k = 0, \ldots, n-1$, abbiamo

$$\begin{aligned}
\alpha g &= \alpha b_0 x^{n-1} + \alpha b_1 x^{n-2} + \cdots + \alpha b_{n-1} \\
&= (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \ldots \\
&\quad + (b_{n-1} - a_{n-1})x + b_n - a_n \\
&= (b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1}x + b_n) \\
&\quad - (a_1 x^{n-1} + a_2 x^{n-2} + \ldots a_{n-1}x + a_n) \\
&= x(g - b_0 x^{n-1}) + b_n - (f - a_0 x^n) \\
&= xg - b_0 x^n + b_n - f + a_0 x^n \\
&= xg + b_n - f
\end{aligned}$$

quindi

$$\boxed{f = (x - \alpha)g + b_n}$$

e ciò implica $f(\alpha) = b_n$.

$b_0, \ldots, b_{n-1}$ sono perciò i coefficienti del quoziente nella divisione con resto di f per $x - \alpha$, mentre b_n è il resto, uguale a $f(\alpha)$.

Questo algoritmo è detto *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece semplicemente l'operazione `x**n`.

Abbiamo ricordato l'algoritmo di Horner già a pagina 10; modifichiamo leggermente la funzione che vogliamo usare in Python:

```
def horner (a,x):
  b=0
  for ak in a: b=b*x+ak
  return b
```

Una frequente applicazione dello schema di Horner è il calcolo del valore corrispondente a una rappresentazione binaria o esadecimale.

Otteniamo così $(1, 0, 0, 1, 1, 0, 1, 1, 1)_2$ come

```
horner([1,0,0,1,1,0,1,1,1],2)
```

e $(A, F, 7, 3, 0, 5, E)_{16}$ come

```
horner([10,15,7,3,0,5,14],16):

x=horner([1,0,0,1,1,0,1,1,1],2)
print x # 311

y=horner([10,15,7,3,0,5,14],16)
print y # 183971934
```

Somme trigonometriche

Per calcolare somme trigonometriche della forma $\sum_{n=0}^{N} a_n \cos nx$ possiamo usare lo schema di Horner, ottenendo così un algoritmo notevolmente più efficiente del calcolo diretto visto a pagina 16. La rappresentazione

$$\cos x = \frac{e^{ix} + e^{-ix}}{2}$$

ci permette infatti di scrivere la somma nella forma

$$\frac{1}{2}\left(\sum_{n=0}^{N} a_n e^{inx} + \sum_{n=0}^{N} a_n e^{-inx}\right)$$

Ponendo $z_1 := e^{ix}$ e $z_2 := e^{-ix}$, la somma diventa quindi

$$\frac{1}{2}\left(\sum_{n=0}^{N} a_n z_1^n + \sum_{n=0}^{N} a_n z_2^n\right)$$

che può esser calcolata con lo schema di Horner. Per l'esponenziale complesso utilizziamo il modulo `cmath` di Python (pag. 21).

```
def sommacoseni (a,x):
  a=a[:]; a.reverse()
  ix=x*1j
  z1=cmath.exp(ix)
  z2=cmath.exp(-ix)
  return (horner(a,z1)+horner(a,z2))/2

a=[2,3,0,4,7,1,3]

print sommacoseni(a,0.2)
# (14.7458647279+0j)
```

VI. DIZIONARI

Dizionari

Abbiamo incontrato esempi di dizionari nelle pagine 9 e 14. Come voci (o chiavi) si possono usare oggetti non mutabili (ad esempio numeri, stringhe oppure tuple i cui elementi sono anch'essi non mutabili). Un dizionario, una volta definito, può essere successivamente modificato:

```
stipendi = {('Rossi','Trento') : 4000,
('Rossi','Roma') : 8000,
('Gardini','Pisa') : 3400}

stipendi[('Neri','Roma')]=4500

voci=list(stipendi.keys())
voci.sort()
for x in voci:
  print '%-s %7s %d' \
    %(x[0],x[1],stipendi[x])

# Output:

Gardini Pisa    3400
Neri    Roma    4500
Rossi   Roma    8000
Rossi   Trento  4000
```

Si osservi il modo in cui le voci sono state ordinate alfabeticamente, prima rispetto alla prima colonna, poi, in caso di omonimia, rispetto alla seconda colonna.

Se `d` è un dizionario, `d.keys()` è una lista che contiene le chiavi di `d`. L'ordine in cui queste chiavi appaiono nella lista non è prevedibile per cui spesso, ad esempio nell'output, essa viene ordinata come abbiamo fatto in questo esempio.

`d.has_key(x)` indica (mediante un valore di verità) se `d` possiede una voce `x`. Con `del d[x]` la voce `x` viene cancellata dal dizionario. Questa istruzione provoca un errore, se la voce `x` non esiste. Cfr. pag. 27.

`diz.items()` è la lista delle coppie di cui consiste il dizionario.

```
diz = {'a' : 1, 'b' : 2, 'c' : 3,
  'd' : 4, 'x' : 5, 'y' : 6, 'z' : 7}

print diz.items()
# Output che riscriviamo su
# piu' righe:

[('a', 1), ('c', 3), ('b', 2),
('d', 4), ('y', 6), ('x', 5),
('z', 7)]
```

Se `d` è un dizionario, `d.values()` è una lista che contiene i valori delle singole voci; questa funzione non è particolarmente utile, perché in genere non si sa a quale voce un determinato valore è assegnato.

```
diz = {'a' : 1, 'b' : 2, 'c' : 3,
  'd' : 4, 'x' : 5, 'y' : 6, 'z' : 7}

print diz.values()
# [1, 3, 2, 4, 6, 5, 7]
```

Benché non sappiamo appunto a quale voce corrisponda il valore 3, possiamo ad esempio calcolare la media di tutti i valori e effettuare qualche altra analisi statistica.

dict

La funzione `dict` può essere usata, con alcune varianti di sintassi, per generare dizionari da sequenze già esistenti:

```
d = dict(); print d
# {} - dizionario vuoto

a=[('u',1), ('v',2)]
d=dict(a); print d
# {'u': 1, 'v': 2}
```

La funzione permette anche di creare un dizionario senza dover scrivere gli apici per le voci, se queste sono tutte stringhe. Nel codice genetico (esempio nella terza colonna) potremmo quindi scrivere

```
cogen1=dict(Gly='ggg gga ggc ggt',
  Ala='gcg gca gcc gct',...)
```

Utilissimo è

```
voci=['Rossi','Verdi','Bianchi']
stipendi=[1800,1500,2100]

print dict(zip(voci,stipendi))
# Output:

{'Bianchi': 2100, 'Rossi': 1800,
'Verdi': 1500}
```

Sottodizionari

Per estrarre da un dizionario le informazioni che corrispondono a un sottoinsieme di voci possiamo usare la seguente funzione:

```
def sottodizionario (diz,chiavi):
  return dict([(x,diz.get(x))
    for x in chiavi])
```

Esempio:

```
diz=dict(a=7,b=3,c=4,d=5,e=11)
sd=sottodizionario(diz,['a','e'])
print sd
# {'a': 7, 'e': 11}
```

Invertire un dizionario

Per invertire un dizionario invertibile (in cui esiste cioè una biiezione tra voci e valori) usiamo la seguente funzione che a sua volta utilizza il metodo `iteritems` che restituisce un iteratore che corrisponde alle coppie (voce,valore) del dizionario.

```
def inverti (diz):
  return dict([(y,x) for x,y \
    in diz.iteritems()])

diz=dict(a='0100',b='1100',i='0010')

def decodifica (a,diz):
  inv=inverti(diz); p=''
  for k in xrange(0,len(a),4):
    p+=inv[a[k:k+4]]
  return p

codifica='11000100110011000010'
print decodifica(codifica,diz)
# babbi
```

Il codice genetico

Dizionari possono essere molto utili in alcuni compiti della bioinformatica. Definiamo un dizionario che descrive il codice genetico:

```
cogen1 = {'Gly' : 'ggg gga ggc ggt',
  'Ala' : 'gcg gca gcc gct',
  'Val' : 'gtg gta gtc gtt',
  'Leu' : 'ctg cta ctc ctt ttg tta',
  'Ile' : 'ata atc att',
  'Ser' : 'tcg tca tcc tct agc agt',
  'Thr' : 'acg aca acc act',
  'Cys' : 'tgc tgt', 'Met' : 'atg',
  'Asp' : 'gac gat', 'Glu' : 'gag gaa',
  'Asn' : 'aac aat', 'Gln' : 'cag caa',
  'Phe' : 'ttc ttt', 'Tyr' : 'tac tat',
  'Lys' : 'aag aaa', 'His' : 'cac cat',
  'Trp' : 'tgg',
  'Arg' : 'cgg cga cgc cgt agg aga',
  'Pro' : 'ccg cca ccc cct',
  'STOP' : 'tga tag taa'}

cogen = {}
for amina in cogen1.keys():
  v=cogen1[amina].split()
  for x in v: cogen[x]=amina

dna='ttagattgcttggagtcatacttagataca'
n=len(dna)

for i in xrange(0,n,3):
  tripla=dna[i:i+3]
  print cogen[tripla],

# Output scritto su due righe:

Leu Asp Cys Leu Glu
Ser Tyr Leu Asp Thr
```

In questo esempio abbiamo prima creato una tabella `cogen1` in cui ad ogni aminoacido è assegnata una stringa che contiene le triple di nucleotidi che corrispondono a questo aminoacido. Spezzando queste stringhe mediante `split` riusciamo poi a creare una tabella `cogen` in cui per ogni tripla è indicato l'aminoacido definito dalla tripla. Infine abbiamo tradotto un pezzo di DNA nella sequenza di aminoacidi a cui esso corrisponde secondo il codice genetico.

„The Python world has definitely increased its growth rate over the last three years. The primary network newsgroup for the language has seen traffic grow to the point where only some sort of filtering allows the average reader to keep up. The number of questions from new users continues to increase, and the range of application areas with Python solutions is growing daily. Just the same, the programming community at large still manages to resist Python's charms, for reasons often unstated but doubtless including inertia and ignorance ...

Python ... is becoming increasingly popular with the web community. There are many reasons for this, not least of which are the amazing range of library code that comes with the system or can be obtained from other sources; the high levels of productivity that can be maintained over large projects; the wide range of platforms ...

Python is an excellent first language, but it also appeals to experienced programmers. Its clean syntax, coupled with a lack of static typing, make it ideal for the pragmatists among us who are primarily interested in programming languages as tools for getting jobs done," (Holden, xvii, 3, 21)

Traduzione di nomenclature

Un problema apparentemente banale, ma molto importante in alcuni campi della ricerca scientifica è la traduzione di nomenclature. Assumiamo che un gruppo di scienziati abbia acquistato un microchip di DNA che permette di studiare l'espressione di 1000 geni che sono identificati secondo una nomenclatura definita dalla ditta produttrice e supponiamo che il gruppo di lavoro abbia studiato (per semplicità) gli stessi geni, ma sotto altro nome. Allora è necessario creare un dizionario che converte i nomi dei geni da una nomenclatura all'altra. Un problema simile si pone spesso quando si effettua una ricerca di una sostanza nota sotto vari nomi in Internet.

Un semplice automa

A ed X siano insiemi finiti e $\varphi : X \times A \longrightarrow X$ un'applicazione. Allora la tripla (X, A, φ) si chiama un'*automa finito*. Questa è una delle possibili definizioni, di cui esistono anche varianti più generali.

Definiamo adesso l'insieme A^* delle *parole* sull'alfabeto A ponendo

$$A^* := \bigcup_{n=0}^{\infty} A^n$$

e denotiamo con ε la parola vuota, l'unico elemento di A^0. A^* è un semigruppo (altamente non commutativo) con la concatenazione naturale delle parole in cui ε funge da elemento neutro, per cui A^* è anche un monoide. Per ogni $a \in A$ abbiamo un'applicazione $\varphi_a := \bigcirc_x \varphi(x, a) : X \longrightarrow X$

e ciò ci permette di definire per ogni parola non vuota $p = a_1 \dots a_n$ un'applicazione
$\varphi_p : X \longrightarrow X$ ponendo

$$\varphi_p(x) := \varphi_{a_n} \circ \cdots \circ \varphi_{a_1}$$

mentre $\varphi_\varepsilon(x) := x$. In questo modo otteniamo un sistema dinamico

$$X \times A^* \longrightarrow X$$

Realizzazione in Python: Siano $A = \{a, b\}$, $X = \{0, 1, 2, 3, 4\}$, e φ data dalla tabella

	a	b
0	1	4
1	3	4
2	2	3
3	0	1
4	1	0

che corrisponde al dizionario annidato

```
phi=dict(a={0:1,1:3,2:2,3:0,4:1},
  b={0:4,1:4,2:3,3:1,4:0})
```

I calcoli sono effettuati nel modo seguente:

```
def valautoma (phi):
  def val (p,x):
    for a in p: x=phi[a][x]
    return x
  return val

for x in xrange(5):
  print valautoma(phi)('baa',x),
# 3 3 1 0 3
```

Fusione di dizionari

Con `tab.update(tab1)` il dizionario `tab` viene fuso con il dizionario `tab1`. Ciò significa più precisamente che alle voci di `tab` vengono aggiunte, con i rispettivi valori, le voci di `tab1`; voci già presenti in `tab` vengono sovrascritte.

```
tab=dict(a=1, b=2)
tab1=dict(b=300, c=4, d=5)

tab.update(tab1)
print tab
# {'a': 1, 'c': 4, 'b': 300, 'd': 5}
```

Argomenti associativi

Se definiamo una funzione

```
def f (**tab):
  print tab
```

la possiamo utilizzare nel modo seguente:

```
f(u=4,v=6,n=13)
# {'n': 13, 'u': 4, 'v': 6}
```

Gli argomenti associativi individuati dal doppio asterisco possono essere preceduti da argomenti fissi o opzionali, evitando interferenze. Come si vede, nella tabella che viene generata i nomi delle variabili vengono trasformati in stringhe che corrispondono alle voci della tabella.

Menu interattivi

Possiamo usare dizionari i cui valori sono funzioni per creare piccoli menu interattivi sul terminale.

```
def menu (scelte, testo=''):
  while 1:
    risposta=raw_input(testo)
    if risposta=='fine': break
    if scelte.has_key(risposta):
      scelte[risposta]()

def curva (): print 'Disegno una curva.'
def media (): print 'Calcolo la media.'
def file (): print 'Carico un file.'

scelte=dict(c=curva, m=media, f=file)

menu(scelte,'scelta: ')
```

Per aiutare l'utente si potrebbe visualizzare sul terminale anche l'elenco delle scelte possibili.

Con `d.get(x)` si ottiene `d[x]`; questo metodo può essere usato con un secondo argomento con cui si può impostare il valore che viene restituito quando la chiave indicata nel primo non esiste. Possiamo effettuare le seguenti modifiche nel nostro programma:

```
def menu (scelte, testo=''):
  while 1:
    risposta=raw_input(testo)
    if risposta=='fine': break
    scelte.get(risposta,passa)()

def passa (): pass
```

del

Il comando `del` cancella un *nome* che quindi non può più essere usato per riferirsi a un oggetto. Se lo stesso oggetto è conosciuto all'interprete sotto più nomi, gli altri nomi rimangono validi:

```
a=7; b=a; del a; print b
# 7
print a
# Errore: il nome a non e' definito

c=7; d=c; del d; print c
# 7
print d
# Errore: il nome a non e' definito
```

Si possono anche cancellare più nomi con un solo comando `del`:

```
a=88; b=a; c=a; del a,c
print b
# 88
```

Il comando `del` può essere anche utilizzato per eliminare un elemento da una lista:

```
a=[0,1,2,3,4,5,6,7]
del a[1]; print a
# [0, 2, 3, 4, 5, 6, 7]

b=[0,1,2,3,4,5,6,7]
del b[2:5]; print b
# [0, 1, 5, 6, 7]
```

Ciò ci permette di definire una funzione analoga al `pop`, la quale toglie il primo elemento da una lista e lo restituisce come risultato (`pop` fa la stesso cosa con l'ultimo elemento di una lista):

```
def spezza (a):
  x=a[0]; del a[0]; return x

a=[0,1,2,3,4,5,6]; x=spezza(a)
print x, a
# 0 [1, 2, 3, 4, 5, 6]
```

`del` può essere anche usato per dizionari:

```
voti = {'Rossi' : 28, 'Verdi' : 27,
  'Bianchi' : 25}
del voti['Verdi']
print voti
# {'Bianchi': 25, 'Rossi': 28}
```

Non confondere `del` con il metodo `remove` delle liste:

```
a=[0,1,2,3,4,5,2]
a.remove(2); print a
# [0, 1, 3, 4, 5, 2]
```

Il metodo clear

Con `diz.clear()` vengono cancellate tutte le voci in un dizionario `diz` che quindi dopo l'esecuzione del comando risulta uguale al dizionario vuoto `{}`.

```
diz=dict(a=17,b=18,c=33,d=45)
print diz
# {'a': 17, 'c': 33, 'b': 18, 'd': 45}
diz.clear(); print diz
# {}
```

VII. STRINGHE

Output formattato

Abbiamo già usato varie volte (alle pagine 9, 24, 26) la possibilità di definire stringhe formattate mediante il simbolo %, secondo una modalità molto simile a quella utilizzata nelle istruzioni `printf`, `sprintf` e `fprintf` del C. Consideriamo un altro esempio:

```
m=124; x=87345.99
a='%3d %-12.4f' %(m,x)
print a
124 87345.9900
```

La prima parte dell'espressione che corrisponde ad `a` è sempre una stringa. Questa può contenere delle *sigle di formato* che iniziano con % e indicano la posizione e il formato per la visualizzazione degli argomenti aggiuntivi. Nell'esempio `%3d` *tiene il posto* per il valore della variabile `m` che (da un'istruzione `print`) verrà visualizzata come intero su uno spazio di tre caratteri, mentre `%-12.4f` indica una variabile di tipo `double` che è visualizzata su uno spazio di 12 caratteri, arrotondata a 4 cifre dopo il punto decimale, e allineata a sinistra a causa del - (altrimenti l'allineamente avviene a destra). Quando i valori superano lo spazio indicato dalla sigla di formato, viene visualizzato lo stesso il numero completo, rendendo però imperfetta l'intabulazione che avevamo in mente. Esempio:

```
a='u=%6.2f\nv=%6.2f' %(u,v)
print a
# Output:

u=12345.67
v= 20.00
```

Nonostante avessimo utilizzato la stessa sigla di formato `%6.2f` per `u` e `v`, prevedendo lo spazio di 6 caratteri per ciascuna di esse, l'allineamento in `v` non è più corretto, perché la `u` impiega più dei 6 caratteri previsti.

I formati più usati sono:

%c	carattere
%d	intero
%f	`double`
%s	stringa (utilizzando `str`)
%x	rappr. esadecimale minusc.
%X	rappr. esadecimale maiusc.
%o	rappr. ottale

Per specificare un segno di % nella sigla di formato si usa %%.

All'interno della specificazione di formato si possono usare - per l'allineamento a sinistra e 0 per indicare che al posto di uno spazio venga usato 0 come carattere di riempimento negli allineamenti a destra. Quest'ultima opzione viene usata spesso per rappresentare numeri in formato esadecimale byte per byte. Vogliamo ad esempio scrivere la tripla di numeri $(58, 11, 6)$ in forma esadecimale e byte per byte (cioè riservando due caratteri per ciascuno dei tre numeri). Le rappresentazioni esadecimali sono `3a`, `b` e `6`, quindi con

```
a=58; b=11; c=6
print '%2x%2x%2x' %(a,b,c)
# 3a b 6
```

otteniamo `3a b 6` come output; per ottenere il formato desiderato `3a0b06` (richiesto ad esempio dalle specifiche dei colori in HTML) dobbiamo usare invece

```
print '%02x%02x%02x' %(a,b,c)
# 3a0b06
```

Come visto, nelle sigle di formato di `printf` può essere specificato il numero dei caratteri da usare; con * questo numero diventa anch'esso variabile e viene indicato negli argomenti aggiuntivi. Assumiamo che vogliamo stampare tre stringhe variabili su righe separate; per allinearle, calcoliamo il massimo `m` delle loro lunghezze, usando le funzioni `len` e `max` del Python, e incorporiamo questa informazione nella sigla di formato:

```
a='Ferrara'; b='Roma'; c='Rovigo'
m=max(len(a),len(b),len(c))
print '|%*s|\n|%*s|\n|%*s|' \
  %(m,a,m,b,m,c)
```

Si ottiene l'output desiderato:

```
|Ferrara|
|   Roma|
| Rovigo|
```

L'elemento che segue il simbolo % esterno è una tupla; invece di indicare i singoli elementi possiamo anche usare una tupla definita in precedenza:

```
u=(7,3,5)

print "cosicche' a=%d, b=%d, c=%d" %u
# cosicche' a=7, b=3, c=5
```

Tramite stringhe formattate si possono anche costruire semplici tabelle. Con

```
stati=dict(Afghanistan=[652000,21000],
  Giordania=[89000,6300],
  Kazakistan=[2725000,14900],
  Mongolia=[1587000,2580],
  Turchia=[780000,65000],
  Uzbekistan=[448000,25000])

formato='%-11s | %14s | %7s'
print formato %('stato',
  'superficie/kmq','ab/1000')
print '-'*38
voci=stati.keys(); voci.sort()
for x in voci:
  stato=stati[x]
  print formato %(x,stato[0],stato[1])
```

otteniamo

```
stato       | superficie/kmq | ab/1000
--------------------------------------
Afghanistan |         652000 |   21000
Giordania   |          89000 |    6300
Kazakistan  |        2725000 |   14900
Mongolia    |        1587000 |    2580
Turchia     |         780000 |   65000
Uzbekistan  |         448000 |   25000
```

Una tabella del coseno:

```
for k in xrange(10):
  x=10*k*math.pi/180
  print "| %2d | %6.3f |" \
    %(10*k,math.cos(x))
# Output:

|  0 |  1.000 |
| 10 |  0.985 |
| 20 |  0.940 |
| 30 |  0.866 |
| 40 |  0.766 |
| 50 |  0.643 |
| 60 |  0.500 |
| 70 |  0.342 |
| 80 |  0.174 |
| 90 |  0.000 |
```

Invertire una stringa

Per poter applicare funzioni definite per le liste a una parola, si può trasformare la parola in una lista con `list`. Vogliamo ad esempio invertire una parola:

```
a='terra'
b=list(a); b.reverse()
print a
# terra
print b
# ['a', 'r', 'r', 'e', 't']
```

`a` non è cambiata (e non può cambiare, perché stringhe sono immutabili), perché `list` crea una copia (non profonda) di `a`. Ma `b` adesso è una lista; come ricaviamo una parola? Per fare ciò si può usare il metodo `join` previsto per le liste, che abbiamo già incontrato alle pagine 16 e 24:

```
b=''.join(b); print b
# arret
```

Possiamo così definire una nostra funzione `invertistringa`:

```
def invertistringa (a):
  b=list(a); b.reverse()
  return ''.join(b)

a='acquario'
b=invertistringa(a)
print a
# acquario
print b
# oirauqca
```

Insiemi di lettere

Il modulo `string` contiene alcuni insiemi di lettere in forma di stringhe:

`.lowercase`	lettere minuscole
`.uppercase`	lettere maiuscole
`.letters`	minuscole e maiuscole
`.digits`	0123456789
`.hexdigits`	0123456789abcdefABCDEF
`.punctuation`	!"#$%&'()*+,-./:;<=>?@ [\]^_`{|}~
`.whitespace`	ASCII 9-13 e 32

Queste costanti devono essere usate con il prefisso `string`:

```
print string.lowercase
# abcdefghijklmnopqrstuvwxyz
```

Unione di stringhe

Stringhe possono essere unite, come tutte le sequenze, mediante l'operatore + oppure, come abbiamo già visto alle pagine 16, 24 e 28, utilizzando il metodo `join` del separatore che vogliamo usare. `join` è molto più veloce di +. La sintassi è

```
sep.join(v)
```

dove `sep` è il separatore, `v` una sequenza di stringhe che vogliamo unire utilizzando il separatore indicato. Un altro esempio:

```
print '--'.join(['alfa','beta','rho'])
# alfa--beta--rho
```

`join` viene usato piuttosto frequentemente, in genere con i separatori ' ' e '\n'. Le parentesi quadre necessarie quando gli elementi del vettore `v` delle stringhe da unire vengono dati esplicitamente possono ridurre la leggibilità (soprattutto quando il `join` nella stessa espressione viene usato più volte); perciò definiamo alcune funzioni ausiliarie con nomi brevi per questi casi:

```
def U (*a): return ''.join(a)

def Unr (*a): return '\n'.join(a)

def Usp (*a): return ' '.join(a)

def Uv (a): return ''.join(a)

def Uvnr (a): return '\n'.join(a)

def Uvsp (a): return ' '.join(a)
```

Esempi:

```
a=U('Africa','Asia','Europa')
print a
# AfricaAsiaEuropa

a=Usp('Africa','Asia','Europa')
print a
# Africa Asia Europa

a=Unr('Africa','Asia','Europa')
print a # Output:

Africa
Asia
Europa
---------------------------------------
a=Unr(Usp('Africa','Asia','Europa'),
    Usp('America','Australia'),
    Usp('Artide','Antartide'))
print a
# Output

Africa Asia Europa
America Australia
Artide Antartide
---------------------------------------
p=['Africa','Asia','Europa']
q=['America','Australia']
r=['Artide','Antartide']
a=Unr(Uvsp(p),Uvsp(q),Uvsp(r))
print a
# Output = precedente.
---------------------------------------
print Uvsp(string.digits)
# 0 1 2 3 4 5 6 7 8 9

url='www.unife.it'; fe='Ferrara'
print U('<a href="',url,'">',fe,'</a>')
# <a href="www.unife.it">Ferrara</a>
```

upper e lower

Per trasformare tutte le lettere di una stringa in maiuscole risp. minuscole si usano i metodi `upper` e `lower` (visto a pagina 16):

```
a='Padre Pio'; b='USA'
print a.upper(), b.lower()
# PADRE PIO usa
```

Il metodo `swapcase` trasforma minuscole in maiuscole e viceversa:

```
print 'Padre Pio'.swapcase()
# pADRE pIO
```

Il metodo `capitalize` trasforma la lettera iniziale di una stringa in maiuscola, le altre in minuscole, anche quando sono precedute da uno spazio. Se si desidera invece che la lettera iniziale della stringa e le lettere precedute da uno spazio vengano trasformate in maiuscole, bisogna usare il metodo `title` oppure la funzione `string.capwords`; quest'ultima sostituisce anche ogni spazio multiplo con uno spazio semplice e si comporta diversamente da `title` quando incontra lettere precedute da un'interpunzione:

```
print 'alfa beta rho'.capitalize()
# Alfa beta rho

print string.capwords('alfa beta rho')
# Alfa Beta Rho
print 'alfa beta rho'.title()
# Alfa Beta Rho

print string.capwords('alfa    beta')
# Alfa Beta
print 'alfa    beta'.title()
# Alfa    Beta

print string.capwords('ab ,u , u 2a xy')
# # Ab ,u , U 2a Xy
print 'ab ,u , u 2a xy'.title()
# Ab ,U , U 2A Xy
```

Verifica del tipo di carattere

I seguenti metodi restituiscono `False`, se `a` è una stringa vuota. Supponiamo quindi che `a` sia una stringa con almeno un carattere.

`a.isalpha()`	Vero, se ogni carattere di `a` è una lettera, fa cioè parte di `string.letters`.
`a.isdigit()`	Vero, se ogni carattere di `a` è una delle cifre 0,...,9.
`a.isalnum()`	Vero, se ogni carattere di `a` è una lettera o una delle cifre 0,...,9.
`a.islower()`	Vero, se tutte le lettere tra i caratteri di `a` sono minuscole.
`a.isupper()`	Vero, se tutte le lettere tra i caratteri di `a` sono maiuscole.
`a.isspace()`	Vero, se tutti i caratteri di `a` appartengono a `string.whitespace`.

```
print '3iax--ff'.islower()
# True
print 'NORD SUD'.isupper()
# True
```

Ricerca in stringhe

Per la ricerca in stringhe sono disponibili i seguenti metodi. Come osservato a pagina 12, `index` e `count` possono essere usati anche per stringhe. `index` non è però indicato nel seguente elenco perché, a differenza da `find`, provoca un errore quando la sottostringa cercata non fa parte della stringa. In seguito supponiamo che `a` ed `x` siano stringhe.

`a.find(x)`	Indice della prima posizione in cui `x` appare in `a`; restituisce -1 se `x` non fa parte di `a`.
`a.find(x,i)`	Come `find`, ma con ricerca in `a[i:]`. L'indice trovato si riferisce ad `a`.
`a.find(x,i,j)`	Come `find`, ma con ricerca in `a[i:j]`. L'indice trovato si riferisce ad `a`.
`a.rfind(x)`	Come `find`, ma la ricerca inizia a destra, con l'indice comunque sempre contato dall'inizio della stringa.
`a.rfind(x,i)`	Come per `find`.
`a.rfind(x,i,j)`	Come per `find`.
`a.count(x)`	Indica quante volte `x` appare in `a`.
`a.count(x,i)`	Indica quante volte `x` appare in `a[i:]`.
`a.count(x,i,j)`	Indica quante volte `x` appare in `a[i:j]`.
`a.startswith(x)`	Vero se `a` inizia con `x`.
`a.startswith(x,i)`	Vero, se `a[i:]` inizia con `x`.
`a.startswith(x,i,j)`	Vero, se `a[i:j]` inizia con `x`.
`a.endswith(x)`	Vero, se `a` termina con `x`.
`a.endswith(x,i)`	Vero, se `a[i:]` termina con `x`.
`a.endswith(x,i,j)`	Vero, se `a[i:j]` termina con `x`

Esempi:

```
a='01201012345014501020'

print a.find('0120')
# 0
print a.find('01',2)
# 3

print a.rfind('010')
# 15

print a.count('01')
# 5

print a.startswith('012')
# True
print a.startswith('120')
# False
print a.startswith('120',1)
# True
print a.startswith('120',1,2)
# False
```

Eliminazione di spazi

Uno dei compiti più frequenti dell'elaborazione di testi elementare è l'eliminazione (talvolta anche l'aggiunta) di spazi (o eventualmente di altri caratteri) iniziali o finali da una parola. Elenchiamo i metodi per le stringhe previste a questo scopo in Python. `a` ed `s` siano stringhe, `n` un numero naturale; `s` viene utilizzata come contenitore dei caratteri da eliminare. Tutti i metodi restituiscono una nuova stringa senza modificare `a` (infatti stringhe non sono mutabili).

`a.strip()`	Si ottiene da `a`, eliminando spazi bianchi (caratteri appartenenti a `string.whitespace`) iniziali e finali.
`a.strip(s)`	Si ottiene da `a`, eliminando i caratteri iniziali e finali che appartengono ad `s`.
`a.lstrip()`	Come `strip`, eliminando però solo caratteri iniziali.
`a.lstrip(s)`	Come `strip`, eliminando solo caratteri iniziali.
`a.rstrip()`	Come `strip`, eliminando solo caratteri finali.
`a.rstrip(s)`	Come `strip`, eliminando solo caratteri finali.
`a.ljust(n)`	Se la lunghezza di `a` è minore di `n`, vengono aggiunti `n-len(a)` spazi alla *fine* di `a`.
`a.rjust(n)`	Se la lunghezza di `a` è minore di `n`, vengono aggiunti `n-len(a)` spazi all'*inizio* di `a`.

```
a='  libro  '
print '[%s]' %(a.strip())
# [libro]

print '[%s]' %(a.lstrip())
# [libro  ]

print '[%s]' %(a.rstrip())
# [  libro]

b='aei terra iia'
print '[%s]' %(b.strip('eia'))
# [ terra ]

c='gente'
print '[%s]' %(c.rjust(7))
# [  gente]
```

split

Abbiamo incontrato questo metodo (usato spessissimo), con cui una stringa può essere decomposta in una lista di parole, a pagina 26. `a` ed `s` siano stringhe, `n` un numero naturale > 0.

`a.split()`	Lista di stringhe che si ottiene spezzando `a`, utilizzando come stringhe separatrici le stringhe consistenti di spazi bianchi.
`a.split(s)`	Lista di stringhe che si ottiene spezzando `a`, usando `s` come separatrice.
`a.split(s,n)`	Al massimo `n` separazioni.

Con il terzo argomento opzionale `n` si può prescrivere il numero massimo di separazioni, ottenendo quindi al massimo `n+1` parti. Questa opzione si usa ad esempio con `maxsplit=1`, se si vuole separare una parola solo nel primo spazio che essa contiene.

```
a='Roma,   Como, Pisa'

print a.split()
# ['Roma,', 'Como,', 'Pisa']

print a.split(',')
# ['Roma', '   Como', ' Pisa']

print map(lambda x: x.strip(),
  a.split(','))
# ['Roma', 'Como', 'Pisa']

b='0532 Comune di Ferrara'
print b.split(' ',1)
# ['0532', 'Comune di Ferrara']
```

Sostituzioni in stringhe

Se `a`, `u` e `v` sono stringhe, `a.replace(u,v)` è la stringa che si ottiene da `a` sostituendo tutte le apparizioni (non sovrapposte) di `u` con `v`.

```
a='andare, creare, stare'
b=a.replace('are','ava')
print b
# andava, creava, stava

a='ararat'
a=a.replace('ara','ava')
print a
# avarat
```

Per sostituire in `a` simultaneamente e nell'ordine tutti i caratteri di una stringa `p` con i corrispondenti caratteri di una stringa `q`, bisogna (se non si usano espressioni regolari, che verranno trattate più avanti) procedere in due passi. In un primo momento si crea con `string.maketrans` una tabella di traduzione, successivamente con il metodo `translate` si ottiene la stringa desiderata. Benché macchinosa, l'esecuzione di questa operazione è molto veloce.

```
a='CRASCASTRAMOVEBO'
cesare=string.maketrans('ABCEMORSTV',
  'DEFHPRUVWY')
print a.translate(cesare)
# FUDVFDVWUDPRYHER

b='alfa, beta, gamma'
tra=string.maketrans(', ','!-')
print b.translate(tra)
# alfa!-beta!-gamma

c='ax tay uvzz xy'
tra=string.maketrans('xyz','XYZ')
print c.translate(tra)
# aX taY uvZZ XY
```

„Python is a general-purpose programming language ... This stable and mature language is very high level, dynamic, object-oriented, and cross-platform - all characteristics that are very attractive to developers. Python runs on all major hardware platforms and operating systems ... Python provides a unique mix of elegance, simplicity, and power.“ (Martelli, pag. 3)

Simulare printf

La seguente ricetta, proposta da Tobias Klausmann e Andrea Cavalcanti nel cookbook, permette di simulare la funzione `printf` del C, mancante in Python.

```
def printf (format,*valori):
  sys.stdout.write(format %valori)
```

Essa talvolta risulta più leggibile del normale uso di stringhe formattate esposto a pagina 28. Mentre `print` passa o una nuova riga per ogni argomento che non sia seguito da una virgola, quando usiamo questa funzione la nuova riga va indicata, come in C, con il carattere `'\n'`; per questo invece di `print` abbiamo usato `sys.stdout.write`.

```
u=7; v=7.23; comune='Pisa'
printf('u=%d, v=%d\n',u,v)
printf('comune di %s\n',comune)
# Output:

u=7, v=7
comune di Pisa
```

A. Martelli/A. Ravenscroft/D. Ascher (ed.): Python cookbook. O'Reilly 2005.

Sistemi di Lindenmayer

Nel corso di Algoritmi 2005/06 abbiamo introdotto i sistemi di Lindenmayer come endomorfismi di un monoide libero A^* generato da un alfabeto finito A. Un sistema di Lindenmayer è univocamente determinato da un'applicazione $f : A \longrightarrow A^*$.

La realizzazione di sistemi di Lindenmayer in Python è particolarmente semplice, come mostrano i seguenti esempi:

```
def linden (f,a):
  return ''.join(map(f,a))

def f (x): # Successione di Morse.
  if x=='0': return '01'
  if x=='1': return '10'
  return ''

a='0'
for k in xrange(6):
  print a; a=linden(f,a)
# Output:

0
01
0110
01101001
0110100110010110
01101001100101101001011001101001

def g (x): # Cantor.
  if x=='n': return 'nbn'
  if x=='b': return 'bbb'
  return ''

a='n'
for k in xrange(4):
  print a; a=linden(g,a)
# Output:

n
nbn
nbnbbbnbn
nbnbbbnbnbbbbbbbbbnbnbbbnbn
```

VIII. FILE E CARTELLE

Comandi per file e cartelle

`os.getcwd()`	Restituisce il nome completo della cartella di lavoro.
`os.chdir(cartella)`	Cambia la cartella di lavoro.
`os.listdir(cartella)`	Lista dei nomi (corti) dei file contenuti in una cartella.
`os.path.abspath(nome)`	Restituisce il nome completo del file (o della cartella) `nome`.
`os.path.split(stringa)`	Restituisce una tupla con due elementi, di cui il secondo è il nome corto, il primo la cartella corrispondente a un file il cui nome è la stringa data. Si tratta di una semplice separazione della stringa; non viene controllato, se essa è veramente associata a un file.
`os.path.basename(stringa)`	Seconda parte di `os.path.split(stringa)`.
`os.path.dirname(stringa)`	Prima parte di `os.path.split(stringa)`.
`os.path.exists(nome)`	Vero, se `nome` è il nome di un file o di una cartella esistente.
`os.path.isdir(nome)`	Vero, se `nome` è il nome di una cartella.
`os.path.isfile(nome)`	Vero, se `nome` è il nome di un file.
`os.path.getsize(nome)`	Grandezza in bytes del file `nome`; provoca un errore se il file non esiste.

Esempi per l'utilizzo di `os.path.split`:

```
print os.path.split('/alfa/beta/gamma') # ('/alfa/beta', 'gamma')

print os.path.split('/alfa/beta/gamma/') # ('/alfa/beta/gamma', '')

print os.path.split('/') # ('/', '')
print os.path.split('//')
# ('//', '') - benche' '//' non sia correttamente formato.

print os.path.split('/stelle') # ('/', 'stelle')

print os.path.split('stelle/sirio') # ('stelle', 'sirio')
```

sys.argv

`sys.argv` è un vettore di stringhe che contiene il nome del programma e gli eventuali parametri con i quali il programma è stato chiamato dal terminale (sia sotto Linux che sotto Windows). Assumiamo che (sotto Windows) il file *prova.py* contenga le seguenti righe:

```
v=sys.argv
print v[0]

a=leggifile(v[1])
print a
```

Se adesso dal terminale diamo il comando `prova.py lettera`, verrà visualizzato prima il nome del programma, cioè *prova.py*, e poi il contenuto del file *lettera*. Se il programma contiene invece le istruzioni

```
v=sys.argv
x=int(v[1]); y=int(v[2])
print x+y
```

dopo `prova.py 7 9` dal terminale verrà visualizzato 16.

`sys.argv` viene spesso utilizzato in combinazione con `execfile` (pagina 25) oppure, e questa è una tecnica molto potente, per attivare l'elaborazione di un file (ad esempio un file di testo su cui si sta lavorando) mediante un altro programma, ad esempio Latex.

Lettura e scrittura di file

Per leggere un file utilizziamo la funzione

```
def leggifile (nome):
  f=open(nome,'r')
  # Oppure f=file(nome,'r').
  testo=f.read()
  f.close(); return testo
```

che ci restituisce il contenuto del file in un'unica stringa. Se da questa vogliamo ottenere una lista che contiene le singole righe, possiamo procedere come nel seguente esempio:

```
a=leggifile('diffus.f')
righe=a.split('\n')
```

In questo caso *diffus.f* è il nome di un file nella stessa cartella in cui si trova il programma; altrimenti sotto Linux potremmo ad esempio usare il nome `/home/rita/fortran/diffus.f` oppure, sotto Windows, `c:/rita/fortran/diffusion.f`.

Per poter scrivere un testo su un file, definiamo la funzione

```
def scrivifile (nome,testo):
  f=open(nome,'w')
  f.write(testo); f.close()
```

Se il testo consiste di più parti, possiamo unire queste parti mediante `join` oppure effettuare più operazioni di scrittura tra `file` e `close` come nella funzione `scrivifilev` che permette di scrivere su un file gli elementi di un vettore `v` di testi, separati da una stringa separatrice inizialmente impostata a `'\n'`:

```
def scrivifilev (nome,testi,sep='\n'):
  f=open(nome,'w')
  for x in testi: f.write(x+sep)
  f.close()
```

Queste operazioni sono molto semplici e quindi in un piccolo programma da una pagina spesso non si usano nemmeno funzioni apposite, scrivendo invece direttamente le istruzioni necessarie:

```
f=open('prova','w')
f.write('Maria Tebaldi\n')
f.write('Roberto Magari\n')
f.write('Erica Rossi\n')
f.close()
```

Quando un file viene aperto con la modalità di scrittura usando la sigla `'w'` come secondo argomento di `file`, il contenuto del file viene cancellato. Se vogliamo invece aggiungere un testo alla fine del file, senza cancellare il contenuto esistente, dobbiamo usare la sigla `'a'`:

```
def aggiungifile (nome,testo):
  f=open(nome,'a')
  f.write(testo); f.close()
```

Anche qui potremmo definire una funzione `aggiungifilev` per la scrittura di un vettore di testi.

Il modulo time

`time.time()`	Tempo in secondi, con una precisione di due cifre decimali, percorso dal primo gennaio 1970 (PG70).
`time.localtime()`	Tupla temporale che corrisponde all'ora attuale.
`time.localtime(s)`	Tupla temporale che corrisponde ad *s* secondi dopo il PG70.
`time.ctime()`	Stringa derivata da `localtime()`.
`time.ctime(s)`	Stringa derivata da `localtime(s)`.
`time.strftime`	Output formattato di tempo e ora. Vedere i manuali, ad esempio il compendio di Alex Martelli, pag. 247.
`time.sleep(s)`	Aspetta `s` secondi, ad esempio `time.sleep(0.25)`.

Questo modulo contiene alcune funzioni per tempo e data, di cui abbiamo elencato le più importanti. Esempi:

```
t=time.localtime(); print t
# (2006, 2, 8, 19, 48, 23, 2, 39, 0)
# anno, mese, giorno, ora, minuti, secondi,
# giorno della settimana (0 = lunedi'),
# giorno nell'anno, 1 se tempo estivo.

print time.strftime('%d %b %Y, %H:%M',t)
# 08 Feb 2006, 19:54

print time.ctime()
# Wed Feb  8 19:50:20 2006

print time.time() # Meglio time.clock().
# 1139424643.05
for i in range(1000000): pass
print time.time() # Meglio time.clock().
# 139424643.23
```

Moduli

Un modulo è un file che contiene istruzioni di Python che possono essere utilizzate da altri moduli o programmi. Il nome del file deve terminare in *.py* (almeno nel caso dei moduli da noi creati), ad esempio *matematica.py*. Il file può trovarsi nella stessa cartella che contiene il programma, oppure in una delle cartelle in cui l'interprete cerca i programmi. La lista con i nomi di queste cartelle è `sys.path`. Per aggiungere una nuova cartella si può usare il metodo `sys.path.append`. Quando il programmatore può accedere al computer in veste di amministratore di sistema (come *root* sotto Linux), esiste però una soluzione molto più comoda per rendere visibili i moduli all'interprete di Python. È infatti sufficiente inserire un file (o anche più file) con l'estensione `.pth` nella apposita cartella di Python, che sotto Linux è tipicamente

```
/usr/local/lib/python2.4/site-packages
```

e sotto Windows (nella nostra versione di Python)

```
c:/python24
```

In essa creiamo il file `cammini.pth` che contiene sotto Linux ad esempio la riga

```
/home/rita/python/moduli
```

e sotto Windows

```
c:/docs/rita/documenti/python/moduli
```

dove abbiamo usato *docs* come abbreviazione per *documents and settings*. Nello stesso modo si possono anche indicare più cartelle per i moduli. Ogni volta quando l'interprete del Python entra in azione, tiene conto di queste direttive per la collocazione dei moduli che vogliamo utilizzare. Più precisamente l'interprete effettua la ricerca dei moduli nel seguente ordine:

(1) Cartella in cui si trova il programma.
(2) Cartelle elencate nella variabile `PYTHONPATH`, se impostata.
(3) Cartelle della libreria standard di Python.
(4) Cartelle indicate nei file *.pth*, se presenti.

Il percorso di ricerca completo può essere controllato esaminando la lista `sys.path`.

Dopo aver importato il modulo `matematica` con l'istruzione

```
import matematica
```

tralasciando il suffisso *.py*, un oggetto `x` di `matematica` al di fuori del modulo stesso è identificato con `matematica.x`. Possiamo anche importare tutti i nomi (in verità solo i nomi pubblici) di `matematica` con

```
from matematica import *
```

oppure anche solo alcuni degli oggetti del modulo, ad esempio

```
from math import cos,sin,exp
```

Una soluzione talvolta preferibile all'uso dei moduli è utilizzo oculato di `execfile` (pag. 34).

Ai fini della trasparenza dei programmi i moduli utilizzati (tranne il programma principale) dovrebbero soltanto contenere definizioni di funzioni ed eventualmente impostazioni iniziali di variabili globali. Il modulo può contenere un qualsiasi insieme di istruzioni che vengono eseguite all'atto dell'importazione che però in questo modo sono difficilmente visibili al programmatore. Si consiglia quindi di usare i moduli solo come raccolte di elementi passivi che verranno attivati dal programma principale.

Anche per le variabili importate mediante un `from` da un modulo bisogna distinguere tra oggetti mutabili e immutabili. Assumiamo che il modulo `aus` contenga le istruzioni

```
x=1; a=[1,2,3]
```

e il programma principale le seguenti righe:

```
from aus import x,a
x=10; a[1]=77

import aus
print aus.x, aus.a
# 1 [1, 77, 3]
```

Vediamo che la variabile `aus.x` non è stata modificata, mentre l'istruzione `a[1]=77` ha avuto effetto sulla lista `aus.a`. Semplificando leggermente, la ragione di questo comportamente è che l'istruzione

```
from aus import x,a
```

è essenzialmente equivalente alle istruzioni

```
x=aus.x
a=aus.a
```

e quindi ciò che abbiamo osservato è in accordo con quanto detto a pagina 12.

Pacchetti

Nelle istruzioni `import` non è possibile utilizzare direttamente cammini composti (ad esempio `grafica/cerchi`). Esiste comunque un meccanismo che permette di raccogliere moduli in sottocartelle, anche annidate. Assumiamo che (per semplicità nella stessa cartella in cui si trova il programma principale) abbiamo creato una cartella *grafica* e in essa un file *cerchi.py*. Allora possiamo utilizzare il contenuto di questo file mediante l'istruzione

```
import grafica.cerchi
```

sotto la condizione però che la cartella *grafica* contenga un file *__init__.py* che può essere anche vuoto; se contiene delle istruzioni, queste vengono eseguite durante l'importazione, ma non è una cattiva idea lasciarlo vuoto. La cartella grafica può anche contenere una sottocartella *astro*; se questa contiene a sua volta un file *__init__.py* e un file *stelle.py*, possiamo importare quest'ultimo con

```
import grafica.astro.stelle
```

Ogni volta che usiamo un oggetto di questo modulo dovremmo riscrivere il lungo nome di questo modulo; per evitare ciò è possibile un'istruzione

```
import grafica.astro.stelle as stelle
```

che ci permette di scrivere semplicemente `stelle`. L'istruzione `import ... as ...` può essere anche usata per nomi complicati non contenuti in un pacchetto.

L'attributo __name__

A differenza dal C, in Python non esiste una chiara distinzione tra programma principale e file ausiliari. Infatti ogni file che contiene istruzioni di Python valide può essere utilizzato come programma principale e, se il suo nome termina con *.py*, anche essere importato da un altro file. Come già osservato, è preferibile separare in modo trasparente i moduli dal programma principale; per capire meglio il meccanismo facciamo però vedere che, tra due file *A.py* e *B.py* ciascuno possa fungere da programma principale utilizzando il secondo come modulo.

Assumiamo che *A.py* contenga le righe

```
if __name__=='__main__':
  import B
  print B.g(3)

def f(x): return x*3-1
```

e *B.py* le righe

```
if __name__=='__main__':
  import A
  print A.f(5)

def g(x): return 2*x+6
```

Se adesso dal terminale eseguiamo il comando `python A.py`, verrà visualizzato `12` (il valore di `B.g(3)`), se battiamo `python B.py`, l'output sarà `14`, il valore di `A.f(5)`.

Osserviamo in primo luogo che tralasciando l'`if`, l'esecuzione si fermerebbe con un errore perché i comandi di importazione reciproca creano una specie di corto circuito.

Possiamo invece, come nell'esempio, utilizzare il fatto che ad ogni modulo è associato un attributo `__name__` che, quando il modulo viene usato come programma principale, diventa `__main__`, mentre altrimenti coincide con il nome del modulo, cioè il nome del file senza l'estensione *.py*.

Alcune costanti in sys

Il modulo `sys` contiene alcune costanti che possono essere talvolta utili, tra cui:

`sys.executable`	Cammino completo in cui si trova l'interprete di Python.
`sys.path`	Lista con i nomi delle cartelle in cui l'interprete cerca i moduli.
`sys.platform`	Ad esempio `linux2` oppure `win32`.

Sotto Windows avremo ad esempio

```
print sys.executable
# c:\Python24\pythonw.exe
```

Variabili globali

Variabili *alla sinistra di assegnazioni* all'interno di una funzione e non riferite esplicitamente a un modulo sono *locali*, se non vengono definite globali con `global`. Non è invece necessario dichiarare `global` variabili esterne che vengono solo lette, senza che gli venga assegnato un nuovo valore.

```
x=7

def f(): x=2
f(); print x
# 7

def g(): global x; x=2

g(); print x
# 2
```

Nell'esempio

```
u=[8]

def f(): u[0]=5

f(); print u
# [5]
```

non è necessario dichiarare `u` come variabile globale, perché viene usata solo in lettura. Infatti non viene cambiata la `u`, ma solo un valore in un indirizzo a cui `u` punta.

Quindi si ha ad esempio anche

```
u=[8]

def aumenta (u): u[0]=u[0]+1

aumenta(u); print u
# [9]
```

Invece di dichiarare una variabile come globale, la si può anche riferire esplicitamente a un modulo. Assumiamo prima che la variabile `u` appartenga a un modulo esterno, ad esempio `mat`, in cui abbia inizialmente il valore 7. Allora possiamo procedere come nel seguente esempio:

```
import mat

x=7

def f ():
  mat.x+=1

f(); print x
# 8
```

Se la variabile appartiene invece allo stesso modulo come la funzione che la dovrebbe modificare, possiamo usare `sys.modules` per identificare il modulo:

```
x=7

def f ():
  sys.modules[__name__].x+=1

f(); print x
# 8
```

A questo scopo possiamo anche importare il modulo stesso in cui ci troviamo e usare la tecnica del penultimo esempio.

Si dovrebbe cercare di utilizzare variabili globali solo quando ciò si rivela veramente necessario e, in tal caso, di non definirle nel file che contiene il programma principale, ma in moduli appositi.

---

globals() e locals()

Con `globals()` si ottiene una tabella (in forma di dizionario) di tutti i nomi globali. Il programma può modificare questa tabella:

```
globals()['x']=33
print x
# 33
```

Come si vede, per ogni oggetto bisogna usare il suo nome come stringa; le istruzioni

```
x=33
```

e

```
globals()['x']=33
```

sono in pratica equivalenti. `locals()` è la lista dei nomi locali. La differenza si vede ad esempio all'interno di una funzione. Assumiamo che il nostro file contenga le seguenti istruzioni:

```
x=7

def f (u):
  print globals()
  print '------------'
  print locals()
  x=3; a=4

f(0)
```

Allora l'output (che abbiamo disposto su più righe sostituendo informazioni non essenziali con ...) sarà

```
{'f': <function ...>,
'__builtins__': <module ...>,
'__file__': './alfa',
'x': 7, '__name__': '__main__',
'__doc__': None}
------------
{'u': 0}
```

Si osservi in particolare che le variabili locali `x` ed `a` non sono state stampate, perché al tempo della chiamata di `locals()` ancora non erano visibili. Esse appaiono invece nell'elenco delle variabili locali della funzione se il file contiene le righe

```
x=7

def f (u):
  x=3; a=4
  print locals()
  x=100

f(0)
# {'a': 4, 'x': 3, 'u': 0}
```

`locals()` è stato stampato prima che venisse effettuato l'assegnamento `x=100`. Si noti che gli argomenti della funzione (in questo caso `u`) sono considerati variabili locali della funzione.

Variabili autonominative

Chiamiamo *autonominativa* una variabile il cui valore è una stringa che coincide con il nome della variabile. Potremmo definire una tale variabile ad esempio con

```
Maria='Maria'
```

ma ciò ci obbliga a scrivere due volte ognuno di questi nomi. Si può ottenere lo stesso effetto utilizzando la funzione che adesso definiamo:

```
def varauto (a):
  for x in a.split(): globals()[x]=x

varauto('Maria Vera Carla')
print Maria
# Maria
```

J. Orendorff: Comunicazione personale, febbraio 1998.

Funzioni per una pila

Nella programmazione avanzata accade abbastanza frequentemente che si voglia utilizzare una pila (in inglese *stack*) globale, che assumiamo sia definita in un modulo `pila` che potrebbe, tra altre, contenere le seguenti istruzioni. Si noti che non abbiamo bisogno nè di dichiarare la pila come globale nè di usare la tecnica dell'esplicito riferimento a un modulo.

```
pila=[]

# Aggiunge gli argomenti alla pila.
def poni (*v): pila.extend(v)

# Toglie gli ultimi k elementi
# dalla pila. Per k=1 restituisce
# l'ultimo elemento della pila,
# altrimenti la lista degli ultimi
# k elementi.
def togli (k=1):
  if k==1: return pila.pop()
  v=[]
  for i in xrange(k):
    v.append(pila.pop())
  v.reverse(); return v

# Togli gli ultimi due elementi dalla
# pila e aggiunge alla pila la
# loro somma.
def add ():
  a,b=togli(2); pila.append(a+b)
```

Nel programma principale potremmo adesso usare le istruzioni

```
import pila

pila.poni(4,3,5,2,10)
print pila.pila
# [4, 3, 5, 2, 10]

print pila.togli()
# 10
print pila.pila
# [4, 3, 5, 2]

pila.add()
print pila.pila
# [4, 3, 7]
```

eval

Se `E` è una stringa che contiene un'espressione valida di Python (ma non ad esempio un'assegnazione), `eval(E)` è il *valore* di questa espressione. Esempi:

```
u=4
print eval('u*u+7')
# 23

def f (x): return x+5

print eval('f(2)+17')
# 24

print eval('f(u+1)')
# 10

def sommaf (F,x,y):
  f=eval(F); return f(x)+f(y)

def cubo (x): return x*x*x

print sommaf('cubo',2,5)
# 133
```

Si noti che avremmo anche potuto definire

```
def sommaf (F,x,y):
  f=globals()[F]; return f(x)+f(y)
```

exec

Se la stringa `a` contiene istruzioni di Python valide, queste vengono eseguite con `exec(a)`.

`exec`, a differenza da `eval`, non restituisce un risultato. Esempi:

```
a='x=8; y=6; print x+y'

exec(a)
# 14
```

`exec` è utilizzato naturalmente soprattutto per eseguire comandi che vengono costruiti durante l'esecuzione di un programma; usato con raziocinio permette metodi avanzati e, se si vuole, lo sviluppo di meccanismi di intelligenza artificiale.

Consideriamo le istruzioni

```
x=0; comando='x=17'

def f ():
  global x
  exec comando

f(); print x
# 0
```

Come mai - nonostante che `x` in `f` sia `global`? La ragione è questa: `global` è una (anzi l'unica) direttiva per il compilatore e riguarda solo l'esecuzione in cui viene chiamata. `exec` fa ripartire il compilatore e quindi bisogna ripetere il `global` nell'espressione a cui si applica `exec`. Dovremmo quindi scrivere:

```
x=0; comando='global x; x=17'

def f ():
  exec comando

f(); print x
# 17
```

execfile

`nomef` sia il nome di un file che contenga istruzioni di Python. Allora con `execfile(nomef)` queste istruzioni vengono eseguite. La differenza pratica principale con `import` è che i nomi in `nomef` valgono come se fossero nomi del file chiamante, mentre con `import` bisogna aggiungere il prefisso corrispondente al modulo definito dal file. Essenzialmente l'effetto di `execfile` è come se avessimo letto il contenuto del file in una stringa `a` ed effettuato il comando `exec(a)`.

Talvolta `execfile` viene utilizzato per leggere parametri di configurazione da un file, ad esempio un file il cui nome appare tra gli elementi di `sys.argv`.

readline

Per leggere un file di testo, normalmente conviene usare la funzione `leggifile` definita a pagina 31 con cui otteniamo un'unica stringa che corrisponde all'intero contenuto del file. Questa stringa poi la elaboreremo con le funzioni per le stringhe fornite dal Python. In questo modo possiamo anche suddividere il testo in righe.

Quando lavoriamo con raccolte di dati di grandi dimensioni, dobbiamo però caricare in memoria molti megabyte. Ciò può essere evitato con l'uso di un generatore.

Utilizziamo in primo luogo il metodo `readline` con cui un file può essere letto una riga alla volta. Nella funzione `leggirighe` che adesso definiamo per ogni riga viene prima controllato se essa è una stringa vuota; questo accade solo se il file non contiene altre righe. Poi però eliminiamo caratteri bianchi iniziali e finali della riga (potremmo in modo simile anche eliminare commenti) e controlliamo una seconda volta se la stringa è vuota e se ciò non accade, la forniamo al generatore con `yield`.

Nell'esempio viene poi calcolata la somma degli elementi del file, che assumiamo contenga in ogni riga un numero intero, senza che questi elementi vengano caricati in memoria:

```
def leggirighe (nome):
  f=open(nome,'r')
  while 1:
    a=f.readline()
    if not a: break
    a=a.strip()
    if a: yield a
  f.close()

righe=leggirighe('dati')
numeri=(int(x) for x in righe)
print numeri
# <generator object at 0xf7d6ec>

def somma (v):
  s=0
  for x in v: s+=x
  return s

print somma(numeri)
# 8183 (ad esempio)
```

sys.exit

Per uscire da Python dall'interno di un programma si usa `sys.exit(0)`. Questa funzione viene spesso utilizzata in programmi interattivi.

Input dalla tastiera

Per l'input dalla tastiera sono previsti le funzioni `raw_input` e `input`. Entrambe accettano un argomento facoltativo che, quando è presente, viene visualizzato sullo schermo prima dell'input dell'utente. Mentre `input` aspetta dalla tastiera una espressione valida in Python che viene calcolata come se facesse parte di un programma, `raw_input` tratta la risposta dell'utente come una stringa. Se ad esempio vogliamo immettere una stringa con `input`, la dobbiamo racchiudere tra apici, mentre con `raw_input` gli apici verrebbero considerati come lettere della stringa.

```
>>> x=input('nome: ')
nome: Giacomo
...
NameError: name 'Giacomo' is not defined
>>> # Giacomo non e' una stringa.

>>> x=input('nome: ')
nome: 'Giacomo'
>>> print x
Giacomo

>>> x=raw_input('nome: ')
nome: Giacomo
>>> print x
Giacomo

>>> x=raw_input('nome: ')
nome: 'Giacomo'
>>> print x
'Giacomo'

>>> def f(x): return x**2

>>> x=input('espressione: ')
espressione: f(9)
>>> print x
81

>>> x=raw_input('espressione: ')
espressione: f(9)
>>> print x
f(9)
```

Il bytecode

Sotto Linux (ma non con Enthought sotto Windows) per ogni file α*.py* Python crea un file semicompilato (*bytecode*) α*.pyc*. Ciò rende più veloce il caricamento dei moduli (ma non l'esecuzione), ha però lo svantaggio di affollare la nostra cartella. Il modo più semplice per eliminare questo problema è di inserire, alla fine del programma stesso un'istruzione che cancella i file *.pyc* creati:

```
import os

os.system('rm *.pyc')
```

stdin, stdout, stderr

Come in C questi file virtuali (contenuti nel modulo `sys`) rappresentano il canale di input, il canale di output e il canale dei messaggi d'errore. Essi sono sempre aperti; `stdout` può essere usato come un file normale, `stdin` è utilizzato tramite le funzioni `input` e `raw_input`.

```
sys.stdout.write('Ciao!')
# Ciao!
```

X. FUNZIONI PER MATRICI

Il prodotto matriciale

Il file *matrici.py* contiene le nostre funzioni per le matrici. Come finora, matrici vengono rappresentate come liste di liste: Se A è una matrice $n \times m$, essa è rappresentata nella forma $A = [A^1, \ldots, A^n]$, dove $A^i = [A^i_1, \ldots, A^i_m]$ è la i-esima riga. `len(A)` è perciò il numero delle righe di `A`.

Le funzioni per matrici che qui presentiamo naturalmente non sono ottimizzate per quanto riguarda efficienza e precisione numerica. Nell'analisi numerica si studiano algoritmi molto migliori. D'altra parte è comodo avere disponibili velocemente funzioni per semplici compiti di calcolo matriciale.

Otteniamo la trasposta di una matrice con

```
def trasposta (A): return map(list,zip(*A))
```

Definizione 35.1. Il prodotto matriciale fv di un vettore riga $f = (f_1, \ldots, f_n)$ con un vettore colonna $\begin{pmatrix} v^1 \\ \ldots \\ v^n \end{pmatrix}$ è definito come $f_1 v^1 + \ldots + f_n v^n$. Se $f^t = \begin{pmatrix} f_1 \\ \ldots \\ f_n \end{pmatrix}$ è il vettore colonna con gli stessi coefficienti di f (cioè la trasposta di f), allora $fv = \|f^t, v\|$.

Il prodotto AB di due matrici A e B si ottiene mettendo nella i-esima riga di AB i prodotti della i-esima riga di A con le colonne di B. Il prodotto matriciale è un'applicazione $\mathbb{R}^n_p \times \mathbb{R}^p_m \longrightarrow \mathbb{R}^n_m$. Si vede facilmente che $(AB)^i_j = \sum_{k=1}^{p} A^i_k B^k_j$ per ogni i, j.

In Python possiamo usare la funzione

```
def mul (A,B):
  return map(lambda r:
    map(lambda c: geom.prodottoscalare(r,c),
      trasposta(B)), A)
```

Nella notazione matematica un vettore riga e un vettore colonna possono essere considerati come casi speciali di matrici. In Python invece matrici sono doppie liste, vettori liste semplici. Per prodotti della forma Av o fA usiamo perciò le seguente funzioni:

```
# Ax.
def mulmatvet (A,x):
  return map(lambda r: geom.prodottoscalare(r,x),A)

# xA.
def mulvetmat (x,A):
  return map(lambda c:
    geom.prodottoscalare(x,c),trasposta(A))

A=[[5,2],[6,7]]; x=[3,4]
print mulvetmat(x,A) # [39, 34]
```

Triangolarizzazione con il metodo di Gauss

Nota 35.2. Nella prima parte dell'algoritmo di eliminazione di Gauss una matrice viene portata in forma triangolare superiore. Nel calcolo a mano visto nel corso di Algoritmi non lo abbiamo fatto, ma nel calcolo automatico bisogna cercare in ogni sezione il coefficiente massimale (di valore assoluto) tra i coefficienti iniziali delle righe ancora da elaborare. Questo coefficiente è detto *perno* (in inglese *pivot*).

```
def perno (A,j):
  B=A[j:]; A=A[0:j]
  B.sort(key=lambda r: abs(r[j]),reverse=1)
  A.extend(B); return A
```

Nota 35.3. Adesso possiamo programmare la triangolarizzazione. Se nessuno dei coefficienti iniziali rimasti è $\neq 0$, l'algoritmo si ferma e restituisce `None`. Ciò non accade (a parte errori di precisone) se la matrice è invertibile.

```
def triangolare (A):
  A=copy.deepcopy(A); n=len(A)
  for j in xrange(0,n):
    A=perno(A,j)
    Aj=A[j]; ajj=float(Aj[j])
    if ajj==0: return None
    for i in xrange(j+1,n):
      A[i]=geom.diff(A[i],geom.mul(A[i][j]/ajj,Aj))
  return A
```

Nota 35.4. Nella funzione `triangolare`, a differenza dal calcolo a mano, abbiamo usato sostituzioni della forma $A^i \longmapsto A^i - \alpha A^j$. Dal corso di Geometria 1 sappiamo che il determinante non cambia con queste operazioni e che il determinante di una matrice (quadratica) triangolare è uguale al prodotto degli elementi nella diagonale principale. Possiamo quindi usare questi ragionamenti per scrivere una funzione per il determinante in Python. Se la triangolarizzazione dà il risultato `None`, ciò significa che un perno è risultato uguale a zero e (come si vede facilmente) ciò implica che il determinante è nullo:

```
def det (A):
  A=triangolare(A)
  if A==None: return 0
  n=len(A); p=1
  for i in xrange(0,n): p*=A[i][i]
  return p
```

Nota 35.5. Per la risoluzione di un sistema triangolare usiamo la seguente funzione:

```
# I coefficienti nella diagonale principale
# devono essere diversi da 0.
def risolvitriangolare (A):
  A=copy.deepcopy(A); n=len(A)
  for i in xrange(0,n): A[i]=geom.mul(1/float(A[i][i]),A[i])
  for j in xrange(n-1,-1,-1):
    Aj=A[j]
    for i in xrange(0,j):
      A[i]=geom.diff(A[i],geom.mul(A[i][j],Aj))
  return A
```

Nell'algoritmo di eliminazione di Gauss la matrice viene prima triangolarizzata, poi si usa la funzione `risolvitriangolare`:

```
def gauss (A):
  n=len(A); A=triangolare(A)
  if A==None: return None
  A=risolvitriangolare(A)
  A=trasposta(A); A=A[n:]; A=trasposta(A); return(A)
```

Il lato destro del sistema può consistere anche di più di un vettore colonna. In particolare possiamo trovare l'inversa di una matrice se a destra scriviamo la matrice identica:

```
def inversa (A):
  B=trasposta(A)
  B.extend(identica(len(A)))
  B=trasposta(B); return gauss(B)
```

Per ottenere una matrice identica usiamo lo stesso metodo come a pagina 16:

```
def identica (n):
  return map(lambda i:
    map(lambda j: int(i==j),xrange(n)),
      xrange(n))
```

XI. CLASSI

Classi

Una *classe* è uno schema di struttura composta; gli *oggetti* della classe, le sue *istanze*, possiedono componenti che possono essere dati (*attributi*) e operazioni (*metodi*) eseguibili per questi oggetti. Un linguaggio di programmazione che, come il Python, fa fortemente uso di questo meccanismo è detto un linguaggio di programmazione orientata agli oggetti. In inglese si parla di *object oriented programming* (OOP). La programmazione orientata agli oggetti è particolarmente popolare nella grafica: ad esempio una finestra di testo può avere come attributi posizione, larghezza e altezza, colore dello sfondo e colore del testo, e come metodi varie operazioni di spostamento e di scrittura. Anche un oggetto geometrico, ad esempio un cerchio, può avere come attributi le coordinate del centro e il raggio, e come metodi operazioni di disegno o di spostamento.

In molti altri tipi di applicazioni la programmazione orientata agli oggetti si rivela utile, ad esempio nell'elaborazione di testi, nella definizione di tipi matematici (matrici o strutture algebriche), ecc. Bisogna però aggiungere che talvolta l'organizzazione per classi causa una certa frammentazione di un linguaggio, sia perché bisogna naturalmente ricordarsi le classi che sono state create, sia perché spesso classi simili eseguono operazioni pressoché equivalenti, eppure non sono più riconducibili a un'unica classe.

La programmazione orientata agli oggetti richiede quindi un notevole lavoro di organizzazione preliminare e molta riflessione e disciplina nella realizzazione delle classi nonché una documentazione che dovrebbe essere allo stesso tempo completa e di facile consultazione.

Come primo esempio definiamo una classe che rappresenta vettori in $\mathbb{R}^3$.

```
class vettore:
  def __init__(A,x,y,z):
    A.x=float(x); A.y=float(y)
    A.z=float(z)
  def __add__(A,B):
    return vettore(A.x+B.x,
      A.y+B.y,A.z+B.z)
  def __mul__(A,t):
    return vettore(A.x*t,
      A.y*t,A.z*t)
  def __rmul__(A,t): return A*t
  def coeff(A): return [A.x,A.y,A.z]
```

In ogni metodo della classe il primo argomento, da noi denotato con `A`, corrisponde all'oggetto della classe a cui il metodo viene applicato. Discutiamo in dettaglio i cinque metodi definiti per la classe `vettore`.

Quando una classe contiene un metodo `__init__`, questo viene automaticamente chiamato quando un oggetto della classe viene creato con un'istruzione che nel nostro caso avrà la forma `vettore(x,y,z)`. Non sono possibili solo assegnazioni della forma `v=vettore(x,y,z)` ma, come si vede nelle definizioni di `__add__` e `__mul__`, è anche possibile definire il valore restituito da una funzione come un nuovo oggetto della classe mediante un `return vettore(x,y,z)`. Il metodo `__init__` può contenere istruzioni qualsiasi, in genere però viene utilizzato per impostare i dati dell'oggetto creato.

Un metodo `__add__` deve essere definito per due argomenti e permette l'uso dell'operatore `+` invece di `__add__`. Esempio:

```
a=vettore(2,3,5)
b=vettore(1,7,2)

c=a+b
print c.coeff()
# [3.0, 10.0, 7.0]

c=a.__add__(b)
print c.coeff()
# [3.0, 10.0, 7.0]
```

Le istruzioni `c=a+b` e `c=a.__add__(b)` sono quindi equivalenti; naturalmente la prima è più leggibile e più facile da scrivere. Si vede che il primo argomento del metodo diventa, nella chiamata, il prefisso a cui il metodo con i rimanenti argomenti viene attaccato. In altre parole, se una classe

```
class alfa:
  ...
  def f(A,x,y): ...
  ...
```

contiene un metodo `f`, questo viene usato come in

```
u=alfa(...)
u.f(x,y)
```

In modo simile il metodo `__mul__` della nostra classe `vettore` permette l'uso dell'operatore `*`. In questo caso il secondo argomento è uno scalare con cui possiamo moltiplicare il vettore:

```
a=vettore(2,3,5)
b=a*4
print b.coeff()
# [8.0, 12.0, 20.0]
```

In verità in matematica si preferisce porre lo scalare prima del vettore, mentre nella definizione dei metodi di una classe l'oggetto chiamante viene sempre per primo. Per ovviare a questa difficoltà Python prevede un metodo `__rmul` che permette di invertire, nella notazione operazionale, l'ordine degli argomenti. In modo simile sono definiti i metodi `__radd__`, `__rdiv__`, `__rmod__`, `__rsub__`, `__rpow__`. Metodi analoghi esistono per gli operatori logici, ad esempio `__rand__`.

Il metodo `coeff` della nostra classe non è speciale e restituisce semplicemente la lista dei coefficienti di un vettore. Si osservi anche qui che un vettore `v` diventa prefisso nella chiamata:

```
print v.coeff()
```

Gli operatori ridefiniti (o *sovraccaricati*) rispettano poi le stesse priorità come gli operatori matematici omonimi. In particolare dobbiamo utilizzare parentesi solo quando lo faremmo anche in un'analoga espressione matematica, guadagnando così ancora in leggibilità:

```
a=vettore(2,3,5)
b=vettore(1,7,2)
c=vettore(0,1,8)
d=vettore(4,0,-1)
e=3*(a+b)+c+d
print e.coeff()
# [13.0, 31.0, 28.0]
```

Componenti di una classe o di un singolo oggetto di una classe possono essere introdotti anche al di fuori della definizione della classe come negli esempi che seguono. Non è una buona idea però, perché in pratica equivale a una ridefinizione della classe e quindi perdiamo il controllo delle strutture che il programma utilizza.

```
class punto:
  def __add__(A,B):
    return punto(A.x+B.x,A.y+B.y)

def add (A,B): return 7

punto.__add__=add

def init (A,x,y): A.x=x; A.y=y

punto.__init__=init

u=punto(3,5);
print u+u
# 7

u.f=math.cos

print u.f(0.8)
# 0.696706709347
```

Come si vede, Python permette di ridisegnare a piacere la struttura di una classe e di aggiungere componenti a singoli oggetti.

Per la stessa ragione i metodi di una classe possono riferirsi a componenti degli oggetti della classe definiti esternamente; anche questa è una possibilità che il programmatore non dovrebbe utilizzare:

```
class cerchio:
  def perimetro (A):
    return 2*A.r*math.pi

c=cerchio()
c.r=4
print c.perimetro()
# 25.1327412287
```

Se la definizione di una classe contiene istruzioni indipendenti, queste vengono direttamente eseguite, come se si trovassero al di fuori della classe:

```
class unaclasse:
  print 7
# 7
```

Anche qui bisogna valutare se l'inserimento di istruzioni non renda meno trasparente il programma. Può essere comunque utile definire valori iniziali di alcuni componenti di una classe; così ad esempio possiamo definire una classe cerchio, i cui oggetti hanno raggio 1 quando non indicato diversamente:

```
class cerchio:
  r=1
  def perimetro (A):
    return 2*A.r*math.pi

c=cerchio()
print c.perimetro()
# 6.28318530718
```

Vediamo che, come in una funzione, variabili che si trovano alla sinistra di un'assegnazione in una classe (al di fuori di un metodo), vengono considerate componenti degli oggetti della classe.

Sovraccaricamento di operatori

Gli operatori unari e binari matematici e logici possono essere sovraccaricati mediante metodi predefiniti nella sintassi, ma liberi nella definizione degli effetti, come abbiamo visto a pagina 36 per gli operatori di addizione e moltiplicazione. Elenchiamo le più importanti di queste corrispondenze:

`__add__`	+ binario
`__sub__`	- binario
`__mul__`	*
`__div__`	/
`__floordiv__`	//
`__mod__`	%
`__pos__`	+ unario
`__neg__`	- unario
`__lshift__`	<<
`__rshift__`	>>
`__and__`	&
`__or__`	|
`__xor__`	^
`__invert__`	~
`__iadd__`	+=
`__isub__`	-=
`__imul__` ecc.	*= ecc.

Il significato di `__radd__` ecc. è stato spiegato a pagina 36.

Se ai metodi della classe `vettore` a pagina 36 aggiungiamo

```
def __iadd__(A,B):
  A=A+B; return A
```

possiamo usare le istruzioni

```
v=vettore(2,3,5); w=vettore(1,0,8)
v+=w
print v.coeff()
[3.0, 3.0, 13.0]
```

Come già osservato, mentre gli operatori così sovraccaricati devono seguire la stessa sintassi degli operatori originali, il significato è del tutto libero:

```
class lista:
  def __init__(A,*v):
    A.elementi=v
  def __lshift__(A,n):
    return A.elementi[n]
  def __pos__(A):
    s=0
    for x in A.elementi:
      s+=x
    return s

a=lista(3,5,9,6,4)
print a<<2, +a
# 9 27

class cerchio:
  def __add__(A,B): print '***'
  def __pos__(A): print 'Ciao.'

u=cerchio()
u+u
# ***
+u
# Ciao.
```

Come vediamo, non è nemmeno necessario che questi metodi restituiscano un oggetto della classe (infatti, nell'ultimo esempio entrambi i metodi visualizzano una stringa sullo schermo e restituiscono `None`).

Per la classe `vettore` potremmo così definire un prodotto scalare e sovraccaricare ad esempio l'operatore `&`:

```
def __and__(A,B):
  return A.x*B.x+A.y*B.y+A.z*B.z

a=vettore(3,2,5); b=vettore(6,1,2)
print a & b
# 30
```

Se avessimo voluto usare lo stesso algoritmo come a pagina 15, avremmo dovuto lavorare con le liste dei coefficienti:

```
def __and__(A,B):
  s=0
  for x,y in zip(A.coeff(),B.coeff()):
    s+=x*y
  return s
```

In questo caso il calcolo diretto impiegato nella prima soluzione sembra più efficiente.

__str__

Il metodo speciale `__str__` può essere usato per ridefinire l'istruzione `print` per gli oggetti di una classe. Esso dovrebbe restituire una stringa che viene poi visualizzata da `print`. Per la classe `vettore` a pagina 36 con

```
a=vettore(3,5,7)
print a
```

otteniamo

```
<__main__.vettore instance at 0x25d9cc>
```

Se però aggiungiamo `__str__` ai metodi della classe con

```
def __str__(A):
  return '%.2f %.2f %.2f' \
    %(A.x,A.y,A.z)
```

possiamo usare `print` per visualizzare i coefficienti del vettore:

```
a=vettore(3,5,7)
print a
# 3.00 5.00 7.00
```

Metodi impliciti

Siccome la natura degli argomenti di una funzione in Python non deve essere nota all'atto della definizione, possiamo definire funzioni che utilizzano componenti di una classe senza che questa appaia esplicitamente nella funzione. Chiamiamo tali funzioni *metodi impliciti*.

Possiamo ad esempio creare una funzione che calcola il prodotto scalare per due oggetti della nostra classe `vettore`, semplicemente utilizzando i loro coefficienti:

```
def scalare (a,b):
  return a.x*b.x+a.y*b.y+a.z*b.z

a=vettore(1,3,9); b=vettore(7,2,4)
print scalare(a,b)
# 49.0
```

La funzione `scalare` è definita al di fuori della classe e formalmente non esiste alcun legame tra la funzione e la classe!

__call__

Un significato speciale ha anche il metodo `__call__`. Quando definito per una classe, esso permette di usare gli oggetti della classe come funzioni. Aggiungiamo questo metodo alla classe `vettore`:

```
def __call__(A,x,y,z):
  A.x=float(x); A.y=float(y)
  A.z=float(z)
```

Adesso con `v(x,y,z)` possiamo ridefinire i coefficienti del vettore `v`:

```
a=vettore(3,5,7)
print a
# 3.00 5.00 7.00
a(8,2,1)
print a
# 8.00 2.00 1.00
```

Le stesse istruzioni usate in `__call__` appaiono anche in `__init__`. È preferibile usarle una volta sola, e quindi i due metodi per la classe `vettore` possono essere riscritti nel modo seguente:

```
def __init__(A,x,y,z): A(x,y,z)
def __call__(A,x,y,z):
  A.x=float(x); A.y=float(y)
  A.z=float(z)
```

Anche nel caso di `__call__` naturalmente si è completamente liberi nella scelta delle operazioni che il metodo deve effettuare. In una libreria grafica potremmo ad esempio usare sistematicamente i metodi `__call__` per l'esecuzione delle operazioni di disegno oppure anche solo per preparare la struttura geometrica di una figura, riservando ad esempio l'operatore `+` per effettuare il disegno, così come nel corso di Algoritmi abbiamo spesso definito una figura come una funzione (in R) che genera un insieme di punti che successivamente può essere disegnato con `lines`:

```
class cerchio:
  def __call__(A,parametri):
    ...
    A.punti=...
  def __pos__(A):
    disegna(A.punti)

a=cerchio()
a(x,y,r); +a
a(u,v,s); +a
```

Sintassi modulare

Se `f` è un metodo della classe `vettore` e se `a` è un oggetto della classe, le espressioni `a.f(x)` e `vettore.f(a,x)` sono equivalenti. Ciò mostra che in Python classi e moduli sono molto simili tra di loro: entrambi sono essenzialmente collezioni di funzioni.

```
a=vettore(7,3,5); b=vettore(2,8,1)
print vettore.__add__(a,b)
# 9.00 11.00 6.00
```

Il costruttore _init_

Il metodo speciale `__init__` si chiama *costruttore* della classe. Una classe può possedere al massimo un costruttore; questo può anche mancare. Usando argomenti opzionali in `__init__` si possono facilmente definire operazioni di inizializzazione differenti per la stessa classe. Esempio:

```
class punto:
  def __init__ (A,x=0,y=0): A(x,y)
  def __call__ (A,x,y):
    A.x=x; A.y=y
  def __str__(A):
    return '%.2f %.2f' %(A.x,A.y)

p=punto()
print p
# 0.00 0.00

p(7,4); print p
# 7.00 4.00
```

Metodi vettoriali

Si possono definire alcuni metodi speciali che permettono di estendere agli oggetti di una classe gli operatori vettoriali previsti per liste. Nell'elenco `a` sia un oggetto di una tale classe.

`__getitem__`	Per calcolare `a[k]`.
`__setitem__`	Per impostare `a[k]`.
`__contains__`	Verifica `x in a`.
`__len__`	Permette `len(a)`.

Potremmo ad esempio aggiungere alla classe `vettore` i metodi

```
  def __getitem__(A,k):
    return A.coeff()[k]
  def __setitem__(A,k,val):
    if k==0: A.x=val
    elif k==1: A.y=val
    elif k==2: A.z=val
  def __contains__(A,x):
    return x in A.coeff()
  def __len__(A): return len(A.coeff())
```

che possono essere così usati:

```
a=vettore(7,3,5)
for x in a: print x,
# 7.0 3.0 5.0

print
a[2]=99; print a, len(a)
# 7.00 3.00 99.00 3
```

__cmp__

Se una classe possiede un metodo `__cmp__`, questo viene utilizzato nei confronti tra oggetti della classe. Una tale funzione di confronto deve essere definita come l'argomento opzionale `cmp` di `sort` (cfr. pagina 17).

```
class punto:
  def __init__(A,x,y): A.x,A.y=x,y
  def __cmp__(A,B):
    return cmp((A.x,A.y),(B.x,B.y))

p=punto(3,5); q=punto(7,1)
print p<=q
# True
```

Sottoclassi

Dopo

```
class animale: ...
```

si può definire una sottoclasse con

```
class leone (animale): ...
```

In principio la sottoclasse eredita tutti i componenti della classe superiore che non vengono ridefiniti nella sottoclasse. L'introduzione di sottoclassi può però rendere complicato l'assetto di un programma e l'apprendimento delle classi create e dei legami tra di loro equivale facilmente a dover apprendere un nuovo linguaggio di programmazione.

Il modulo operator

Questo modulo contiene delle funzioni che permettono di utilizzare gli operatori standard di Python ad esempio in istruzioni `map` e `reduce`, senza dover definire apposite funzioni o espressioni lambda:

```
def somma (*a):
  return reduce(operator.add,a,0)

def prod (*a):
  return reduce(operator.mul,a,1)

print somma(7,5,8,3)
# 23

print prod(7,5,8,3)
# 840

print map(operator.sub,[8,9,4],[6,1,2])
# [2, 8, 2]

print map(operator.neg,[3,5,8,-9,-3,0])
# [-3, -5, -8, 9, 3, 0]
```

Metodi di confronto

Anche gli operatori di confronto possono essere sovraccaricati:

`__eq__`	`==`
`__ne__`	`!=`
`__le__`	`<=`
`__lt__`	`<`
`__ge__`	`>=`
`__gt__`	`>`

La semantica è anche qui del tutto libera:

```
class numero:
  def __init__(A,x): A.x=x
  def __lt__(A,B): print 'Ciao.'

a=numero(6); b=numero(9)
a<b
# Ciao.
```

dir

Con `dir(alfa)` si ottengono i nomi definiti per ogni oggetto alfa che possiede un attributo `__dict__`.

Attributi standard

`F` sia un modulo, una classe o una funzione. Allora sono definiti i seguenti attributi:

`F.__name__`	La stringa `F`.
`F.__dict__`	Elenco degli attributi e dei loro valori per `F`, in forma di un dizionario.
`F.__dir__`	Elenco degli attributi, senza i loro valori.

type e isinstance

Ogni oggetto di Python possiede un tipo univoco che si ottiene tramite la funzione `type`. Con `isinstance` si può verificare se un oggetto appartiene ad una classe.

```
print type(77)
# <type 'int'>
print isinstance(77,int)
# True

print type([3,5,1])
# <type 'list'>
print isinstance([3,5,1],tuple)
# False

print type('alfa')
# <type 'str'>

def f(x): pass
print type(f)
# <type 'function'>

a=vettore(8,0,2)
print type(a)
# <type 'instance'>
print isinstance(a,vettore)
# True
```

Il modulo types

Importando il modulo `types` si può verificare, se un oggetto è di un tipo predefinito.

Esempi:

```
if type(a)==types.IntType: ...
if type(a)==types.FloatType: ...
if type(a)==types.ComplexType: ...
if type(a)==types.FunctionType: ...
if type(a)==types.LambdaType: ...
if type(a)==types.StringType: ...
if type(a)==types.ListType: ...
if type(a)==types.DictionaryType: ...
if type(a)==types.FileType: ...
if type(a)==types.ClassType: ...
```

Una lista dei tipi predefiniti la si ottiene con `dir(types)`.

A pagina 23 abbiamo utilizzato `type` nel Minilisp.

„Python is an object-oriented programming language. Unlike some other object-oriented languages, Python doesn't force you to use the object-oriented paradigm exclusively. Python also supports procedural programming with modules and functions, so you can select the most suitable programming paradigm for each part of your program.“ (Martelli, pag. 69)