INTRODUZIONE ALL'UTILIZZO DI GNU/LINUX

- Appunti delle lezioni di laboratorio -

1.Premessa

Questa vuole essere un'introduzione all'uso del sistema operativo GNU/Linux (d'ora in avanti Linux) come semplici utenti della console. Lo scopo è quello di fornire le conoscenze per
1)aprire una console Linux
2)utilizzare i comandi in linea forniti dalla shell
3)utilizzare i comandi di ridirezione dell'output e i comandi di pipe
4)editare testi dalla console
5) utilizzare l'interprete Python dalla linea di comando della shell

2.Aprire una console Linux

Si ha la possibilità di aprire una console Linux da ambiente windows utilizzando il file eseguibile 'putty.exe' presente sotto lo shared-folder 'risorse' del fileserver 'Atlante'. Lanciando questo eseguibile si effettua una connessione sicura (mediante SSL) con il server Linux.
I parametri da inserire nella finestra che mostra putty sono:

- L'indirizzo ip del server, nel nostro caso 192.168.1.18
- Il protocollo che si vuole utilizzare, nel nostro caso ssl

Ogni volta è necessario autenticarsi fornendo, in fase login, un nome utente e una password, tali parametri sono gli stessi utilizzati per entrare nella rete di istituto.
Una volta autenticati appare una finestra simile alla shell dos (solo perchè è in modalità testo) in cui è possibile fornire i comandi al sistema operativo. I comandi si inseriscono subito dopo il prompt del s.o., questo si presenta nel seguente modo:

```
[d3ci01@thor d3ci01]$
```

dove:
d3ci01 è il login con cui ci si è autenticati,
thor è la macchina che ospita Linux,
d3ci01 è la directory in cui ci si trova,
$ indica la posizione in cui si possono digitare i comandi nella shell aperta con la console.
Per uscire dalla console è necessario digitare `exit` seguito da ritorno carrello o premere la combinazione di tasti [`control+D`].

3.I comandi in linea forniti dalla shell

La shell è un'interprete che legge ed esegue i comandi impartiti dagli utenti.
E' importante sapere che, a differenza del dos, la shell linux è case sensitive, cioè

fa differenza digitare comandi con lettere maiuscole o minuscole.
Segue un elenco di comandi con relativo esempio di utilizzo :

man: è l'help in linea

```
[d3ci01@thor d3ci01]$ man nomecomando
```

3.1.Comandi per la gestione del filesystem

cd: cambia la directory

```
[d3ci01@thor d3ci01]$ cd nomedirectory
```

pwd: indica il nome della directory corrente

```
[d3ci01@thor d3ci01]$ pwd
```

ls: crea la lista dei files presenti in una directory

```
[d3ci01@thor d3ci01]$ ls
```

rm: cancella un file o una directory

```
[d3ci01@thor d3ci01]$ rm nomefile
```

cp: copia uno o più files

```
[d3ci01@thor d3ci01]$ cp nomefile nomedir/.
```

mv: sposta uno o più files o directory

```
[d3ci01@thor d3ci01]$ mv nomefile1 nomefile2
```

mkdir: crea una directory

```
[d3ci01@thor d3ci01]$ mkdir nomedir
```

3.2.Comandi per la manipolazione dei files

cat: visualizza il contenuto di un file

```
[d3ci01@thor d3ci01]$ cat nomefile
```

more: visualizza il contenuto di un file paginandolo

```
[d3ci01@thor d3ci01]$ more nomefile
```

less: consente di navigare all'interno di un file

```
[d3ci01@thor d3ci01]$ less nomefile
```

sort: ordina le righe di un file

```
[d3ci01@thor d3ci01]$ sort nomefile
```

3.3.Comandi per la gestione degli utenti e dei processi

who: stampa l'elenco di tutti gli utenti loggati in quel momento nel sistema

```
[d3ci01@thor d3ci01]$ who
```

whoami: Stampa a video il proprio nome utente

```
[d3ci01@thor d3ci01]$ whoami
```

id: stampa a video il proprio identificativo utente (UID) e l'identificativo di gruppo (GID)

```
[d3ci01@thor d3ci01]$ id
```

ps: visualizza i processi in esecuzione appartenenti all'utente

```
[d3ci01@thor d3ci01]$ ps
```

top: visualizza tutti i processi in corso

```
[d3ci01@thor d3ci01]$ top
```

kill: termina un processo in esecuzione dato l'identificativo di processo (PID)

```
[d3ci01@thor d3ci01]$ kill -p PID -9
```

3.4.Comandi per la gestione dei permessi sui files

chmod: consente di modificare i diritti di uso e apertura dei file

```
[d3ci01@thor d3ci01]$ chmod u+x nomefile
```

chown: cambia il proprietario del file

```
[d3ci01@thor d3ci01]$ chown nomeutente nomefile
```

chgrp: cambia il gruppo di utenti in cui è inserito il file

```
[d3ci01@thor d3ci01]$ chgrp nomegruppo nomefile
```

3.4.1.Digressione sui permessi sui file

Linux è un s.o. multiutente, cioè permette a più utenti di coesistere e di utilizzare le risorse da lui messe a disposizione. Una di queste risorse è il sistema di file (filesystem).
Potete vedere i file come degli oggetti che possono contenere le informazioni più disparate, p.e. un file di testo contiene dello scritto (divinacommedia.txt), un file di musica contiene la codifica di suoni (yellowsubmarine.wav), un file grafico contiene disegni (gioconda.bmp) e così via.
Potrebbe capitare che l'utente d3ci01 non voglia far vedere i propri file all'utente d4ai09, come procedere? Ad ogni file creato il filesystem di Linux associa dei permessi che nel momento in cui il file viene richiesto vengono verificati, se la verifica dà esito positivo allora al richiedente si dà accesso al file altrimenti il richiedente riceve una risposta negativa.
I permessi possibili sono principalmente tre: autorizzazione all'esecuzione , alla lettura e alla scrittura. Questi permessi possono essere combinati come indicato nella seguente tabella:

Numero ottale	***Permesso***	***Descrizione***
0	---	*Nessuna autorizzazione*
1	--x	Esecuzione
2	-w-	Scrittura
3	-wx	Scrittura,Esecuzione
4	r--	Lettura
5	r-x	Lettura,Esecuzione
6	rw-	Lettura,Scrittura
7	rwx	Lettura,Scrittura,Esecuzione

Il *7* indica quindi che tutte le autorizzazioni sono permesse, mentre 0 indica che non è possibile eseguire alcuna operazione sul file. Con *4* il file è disponibile solo per la lettura ma non può essere modificato o eseguito.
Complichiamo un po' la situazione. E' possibile raggruppare gli utenti Linux in

gruppi, p.e.: il gruppo degli amministratori di sistema, il gruppo degli sviluppatori o il gruppo degli studenti di una certa classe. Possiamo, allora, ipotizzare che l'utente 1 del gruppo A voglia condividere in lettura i suoi files con tutti gli utenti del gruppo e voglia impedire l'accesso a tutti gli altri. Per far questo i permessi vengono suddivisi in tre categorie: permessi riferiti all'utente, permessi riferiti al gruppo di appartenenza dell'utente e quelli riferiti a tutti gli altri.
p.e. *lavoro.txt rw-r----* oppure *640* indica che lavoro.txt può essere letto e modificato dall'utente, gli appartenenti al gruppo possono leggerlo e tutti gli altri non hanno alcuna autorizzazione.
programma rwx--x--x indica che l'utente può leggere,modificare ed eseguire il file, gli appartenenti al gruppo e tutti gli altri possono solo eseguire il file.
Esistono file speciali come le directory che vengono indicate con una *d* prima dei permessi.
Esistono delle autorizzazioni speciali che non vengono in questa introduzione affrontati.

4.I comandi di ridirezione dell'output e i comandi di pipe

4.1.Ridirezione dell'output

Solitamente i comandi visualizzano il loro output a video, p.e.

```
[d3ci01@thor d3ci01]$ cat elenco.txt
```

visualizza le righe del file elenco.txt sulla finestra della console Linux. Lo standard output di ogni comando è il video. Ma se noi volessimo stampare l'output del comando sort su un file, in maniera tale da ottenere il file ordinato? Si esegue questa operazione utilizzando i comandi di ridirezione dell'output. Ne esistono due:

- ✓>
- ✓>>

Vediamo subito un'esempio. Si abbiano due files primo.txt e secondo.txt il cui contenuto è:

```
primo.txt:              secondo.txt:
     bravo                   paperino
     cattivo                   pluto
     bello                 pippo
     brutto
```

eseguiamo il seguente comando:

```
[d3ci01@thor d3ci01]$ sort primo.txt > secondo.txt
```

L'output del sort viene reindirizzato sul file secondo.txt, se questo non esiste

viene creato, altrimenti, viene sovrascritto. Se digitiamo:

```
[d3ci01@thor d3ci01]$ cat secondo.txt
```

otteniamo la stampa a video di:

```
secondo.txt:
     bello
     bravo
     brutto
     cattivo
```

cioè il contenuto del file primo.txt è stato ordinato e il contenuto del file secondo.txt è andato perduto.

Ora proviamo il seguente comando:

```
[d3ci01@thor d3ci01]$ sort primo.txt >> secondo.txt
```

Stampiamo a video, mediante il comando cat, il contenuto del file secondo.txt:

```
secondo.txt:
     bello
     bravo
     brutto
     cattivo
     bello
     bravo
     brutto
     cattivo
```

Notiamo che il contenuto ordinato di primo.txt è stato accodato al contenuto del file secondo.txt. Quindi il comando di ridirezione >> non è distruttivo come il comando >, se il file esiste la ridirezione dello standard output viene accodata al contenuto esistente. Se il file non esiste viene creato ex-novo.

4.2.Comandi di pipe

E’possibile connettere lo standard output di un programma con l'input di un altro, cioè l’uscita di un programma invece di comparire a video viene letta come ingresso dal secondo programma. Questo modo di lavorare si avvale dell’'uso dell'operatore pipe “|”. Una linea di comando contenente la pipe si chiama pipeline.
Analizziamo un semplice esempio: vogliamo esaminare il contenuto di una directory che contiene un centinaio di files. Se lanciamo semplicemente il comando ls non riusciamo a vedere i primi files dell’elenco perché scorrono

molto velocemente a video. Una soluzione è quella di usare il comando more che, come abbiamo visto, permette la paginazione di una lista molto lunga.
Come fare? Proviamo a digitare il seguente comando dalla console:

```
[d3ci01@thor d3ci01]$ ls | more
```

Abbiamo così concatenato il comando ls con il comando more, l'output di ls viene mandato in input al comando more che stampa sullo standard output il contenuto della directory paginato.

Abbiamo la possibilità di concatenare diversi comandi tra di loro:

```
comando1 | comando2 | comando3| ... | comandoN
```

L'uso della pipe dà la possibilità di agganciare semplici programmi fra loro per produrre un output complesso. Le pipe stigmatizzano la filosofia della metodologia bottom-up, cioè si raggruppano componenti e funzionalità sino ad arrivare alla sintesi dell'intero progetto (programma). In questo modo costruiamo semplici programmi ma assolutamente ben strutturati e funzionali, in maniera tale che concatenandoli tra di loro otteniamo un output ben più complesso degli output dei programmi presi singolarmente. In questa metodologia si racchiude buona parte della filosofia Unix.

5.Editare testi dalla console

In ambiente Linux sono due gli editor di testo più diffusi e importanti: EMACS e VI

- ✓EMACS (www.gnu.org/software/emacs/) è l'acronimo di Editor MACroS (la prima versione di emacs era un'estensione di un editor chiamato TECO). E' stato scritto da Richard Stallman, il fondatore della Free Software Foundation (FSF www.fsf.org) inventore della licenza GPL (Gnu General Public License).
- ✓VIM (www.vim.org) è l'acronimo di VI Improved (VI è un'editor usato sui sistemi unix, anche in questo caso abbiamo un'editor che parte sulla base di un altro più vecchio).

Noi utilizzeremo VIM (d'ora in poi vi)

5.1.Breve guida ai comandi di VI

Per editare un file già esistente o per crearne uno nuovo si digita dalla console Linux:

```
[d3ci01@thor d3ci01]$ vi nomefile
```

dove *nomefile* è il nome del file che si vuol aprire o creare. In questo modo si apre l'editor in modalità full screen, cioè tutta la finestra della console ospita la schermata di vi.

VI permette di lavorare in tre modalità:

5.1.1 Normal mode
5.1.2 Insert mode
5.1.3 Command mode

5.1.1.Normal mode

Appena entrati in ambiente VI ci si trova in normal mode. Per spostarsi si usano i tasti cursore oppure i tasti:

- ***h*** freccia sinistra
- ***l*** freccia destra
- ***k*** freccia su
- ***j*** freccia giù
- ***x*** cancella un carattere (10x cancella 10 caratteri)
- ***dd*** cancella la riga
- ***u*** esegue undo dell'ultimo comando dato
- ***s*** cancella carattere e passa alla modalità di inserimento (10s cancella 10 caratteri e passa a modalità inserimento)
- ***yy*** copia riga in buffer di memoria
- ***p*** esegue il paste del contenuto del buffer di memoria

5.1.2.Insert mode

Digitando il carattere ***i*** si entra nella modalità di inserimento, sull'ultima riga apparirà un indicatore di tale modo. Si esce dalla modalità inserimento premendo il tasto ESC. In questa modalità si ha la possibilità di digitare il testo in full screen.

5.1.3.Command mode

Digitando il carattere ***:*** si entra nella modalità comandi (la precedente modalità deve essere quella normale). Dopo *:* si possono inserire i seguenti comandi:

- ***:q*** è il comando di quit, chiude la sessione di lavoro e chiede se si vuole salvare il testo
- ***:q!*** quit senza salvare le modifiche effettuate sul testo
- ***:w*** salva il testo
- ***:wq*** salva ed esci

- ***:sp*** *nomefile* divide in due (splitta) la finestra di editor per ospitare un nuovo file di nome nomefile

6.L'interprete Python

6.1.Introduzione

6.2.Ciao mondo!

Cominciamo subito con il primo programma.
Lanciate dalla console di Linux il seguente comando:

```
$vi ciao.py
```

In questo modo aprite l'editor di testi per poter inserire il vostro codice sorgente python. Digitate la seguente riga:

```
print "ciao mondo!"
```

Salvate il file ed eseguitelo con il seguente comando :

```
$python ciao.py
```

Osservate cosa appare a video, dovreste ottenere un output simile a quello indicato di seguito:

```
ciao mondo!
```

Il vostro primo programma stampa a video la frase:"ciao mndo!".
Quindi abbiamo appreso che l'istruzione *print* permette di stampare sullo standard output e le virgolette delimitano una stringa.

6.3.Operatori

Continuiamo con il secondo programma.
Seguendo la stessa procedura del precedente esempio scriviamo il seguente programma:

```
print "2 + 2 = ", 2+2
print "3 * 4 = ", 3*4
print "100-1," = 100-1"
print "(33+2)/5 + 11.5 = ", (33+2)/5+11.5
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
2 + 2 = 4
3 * 4 = 12
99 = 100 – 1
(33+2)/5 + 11.5 = 18.5
```

Abbiamo utilizzato il python come una calcolatrice.
Dall'analisi del codice e dell'output osserviamo che:

- ✓L'istruzione *print* accetta più parametri, questi sono separati dalla virgola e possono essere stringhe o valutazioni di operazioni aritmetiche
- ✓Ogni riga di codice finisce con la fine della riga
- ✓Se un operando è decimale (cioè espresso in floating point) allora il risultato sarà decimale

In python si hanno i seguenti:

OPERATORI ARITMETICI		
Elevamento a potenza	**	5 ** 2 = 25
Moltiplicazione	*	2 * 3 = 6
Divisione	/	14 / 3 = 4
Resto della divisione	%	14 % 3 = 2
Addizione	+	1 + 2 = 3
Sottrazione	-	4 – 3 = 1

Attenzione:

- •14 / 3 = 4 non è un errore! Tutti gli operandi sono interi e quindi il risultato della divisione è intero
- •L'operatore ** si può applicare solo ad operandi interi, per operandi frazionari si deve utilizzare una funzione appositamente realizzata (per la questione si rimanda al capitolo dei moduli *math* e *Numeric*)

6.4.Commento

Terzo programma.
Scriviamo il seguente programma:

```
# questo è un commento
print “prova commento” # anche questo è un commento
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
prova commento
```

Il carattere '#' indica che tutto ciò che è presente sulla destra viene considerato commento sino a fine riga.

6.5.Variabili

Scriviamo il seguente programma:

```
a= 123.4
b23=’pippo’
b=432
c=a+b
d=1
nome=’carlo’
print “a+b= “,c
print “a+d= “,a+d
print ‘il nome è ‘, nome
print nome+b23
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
a+b= 555.4
a+d= 124.4
il nome è carlo
carlopippo
```

Dall’analisi del codice e dell’output osserviamo che:

- ✓Non è necessario dichiarare il tipo di variabili come altri linguaggi di programmazione. L’interprete python riconosce il tipo di variabile quando si effettua l’assegnazione. Questo non sempre è un vantaggio, non dichiarando le variabili si rischia di scrivere un sorgente poco leggibile. Per tali motivi si consiglia di usare abbondantemente il commento per descrivere il significato delle variabili in gioco e in generale la struttura del codice sorgente.
- ✓I nomi delle variabili possono essere alfanumerici o alfabetici, nel primo caso il primo carattere deve essere numerico.
- ✓Un’espressione viene valutata quando richiesto e gli operandi FLOW-CHART
- ✓assumono i valori che hanno al momento della valutazione.
- ✓I valori delle variabili stringhe possono essere delimitati da apici o doppi apici.
- ✓‘+’ è l’operatore che ci permette di concatenare tra loro due o più stringhe alfanumeriche

6.6.Strutture di controllo flusso

Il python ha diverse strutture per controllare il flusso delle informazioni in un algoritmo. Noi ne studieremo tre ed ogni volta confronteremo il programma sorgente con il flow-chart.

6.6.1.Ciclo while

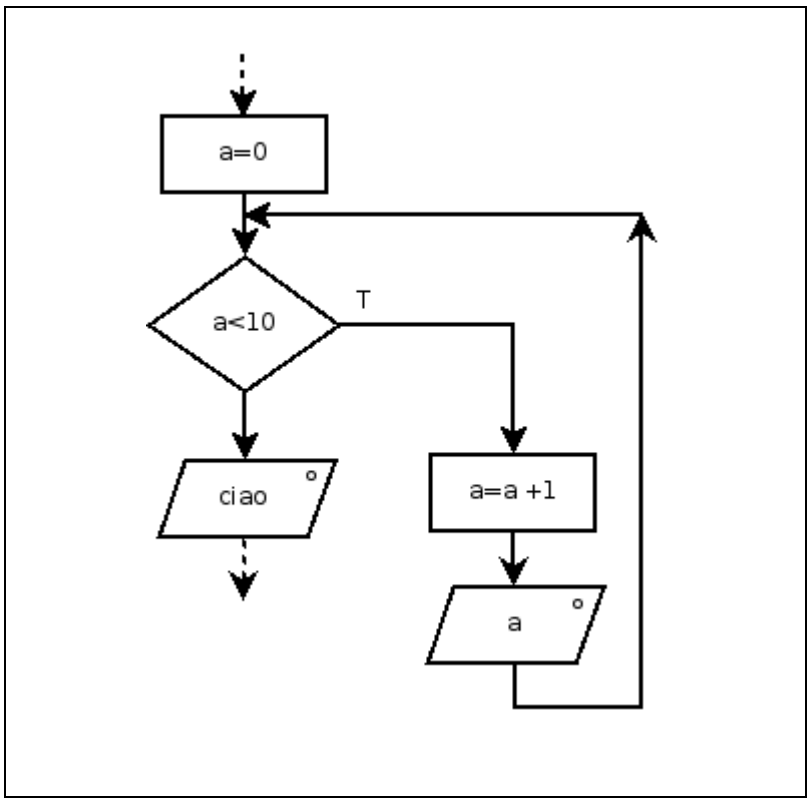
Scriviamo il seguente programma:

```
a = 0
while a < 10 :
     a = a + 1
     print a
print “fine ciclo while”
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
1
2
3
4
5
6
7
8
9
10
fine ciclo while
```

Il flow-chart dell’algoritmo è il seguente:



Dall’analisi del codice e dell’output osserviamo che:

- ✓non esiste un carattere per indicare l'inizio e la fine del corpo del loop. L'interprete riconosce il corpo del loop dall'indentazione dello stesso rispetto all'istruzione while.
- ✓Dopo la condizione di test vanno inseriti obbligatoriamente “:”. Questi introducono l'inizio del loop.
- ✓La prima istruzione eseguita dopo il ciclo while e' la prima istruzione indentata allo stesso livello dell'istruzione while.

6.6.2.Ciclo for

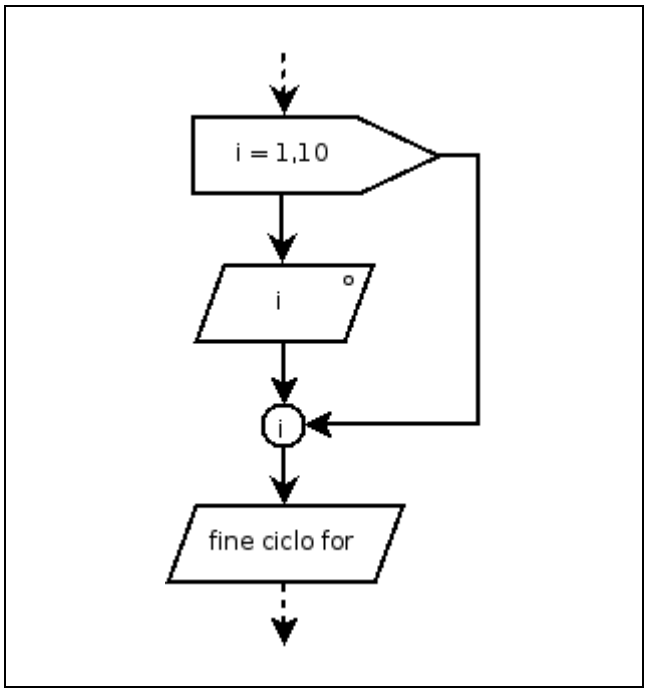
Scriviamo il seguente programma:

```
for i in range (1,11):
     print i
print “fine ciclo for”
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
1
2
3
4
5
6
7
8
9
10
fine ciclo for
```

Il flow-chart dell’algoritmo è il seguente:



Dall’analisi del codice e dell’output osserviamo che:

- ✓non esiste un carattere per indicare l'inizio e la fine del corpo del loop. L'interprete riconosce il corpo del loop dall'indentazione dello stesso rispetto all'istruzione while.
- ✓Dopo la condizione di test vanno inseriti obbligatoriamente “:”. Questi introducono l'inizio del loop.
- ✓La prima istruzione eseguita dopo il ciclo for e' la prima istruzione indentata allo stesso livello dell'istruzione for.
- ✓“range” e' una funzione che restituisce una successione di numeri compresi tra il primo e l'ultimo escluso aventi passo unitario. La sintassi generale e' la

seguente:

```
range (inizio,fine,step)
```

dove inizio= numero di partenza della successione
fine=numero finale della successione escluso
step= passo della successione

p.e. range(1,4,2) fornira' la seguente successione: 1,3
Ovviamente possiamo costruire successioni decrescenti con inizio > fine e step <0 , p.e. range(4,1,-1) dara' origine alla seguente successione: 4,3,2

Solitamente per implementare gli algoritmi di calcolo numerico ci sara' piu' utile lavorare con i cicli while perche' piu' versatili.

6.6.3.Istruzione if

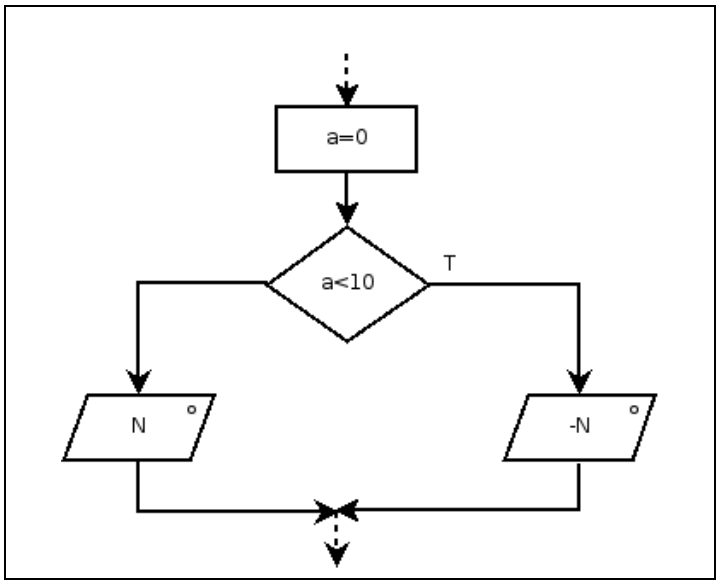
Scriviamo il seguente programma:

```
n=10
if n < 0 :
     print “Il valore assoluto di “,n,” e’ “,-n
else:
     print “Il valore assoluto di “,n,” e’ “,n
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
Il valore assoluto di 10 e’ 10
```

Il flow-chart dell’algoritmo è il seguente:



Dall’analisi del codice e dell’output osserviamo che:

- ✓Anche in questo caso il corpo dell’if viene indicato con l’indentazione cosi’ come quello dell’istruzione else.
- ✓Sia la condizione dell’if che l’istruzione else terminano con “:”, questi vanno inseriti obbligatoriamente.

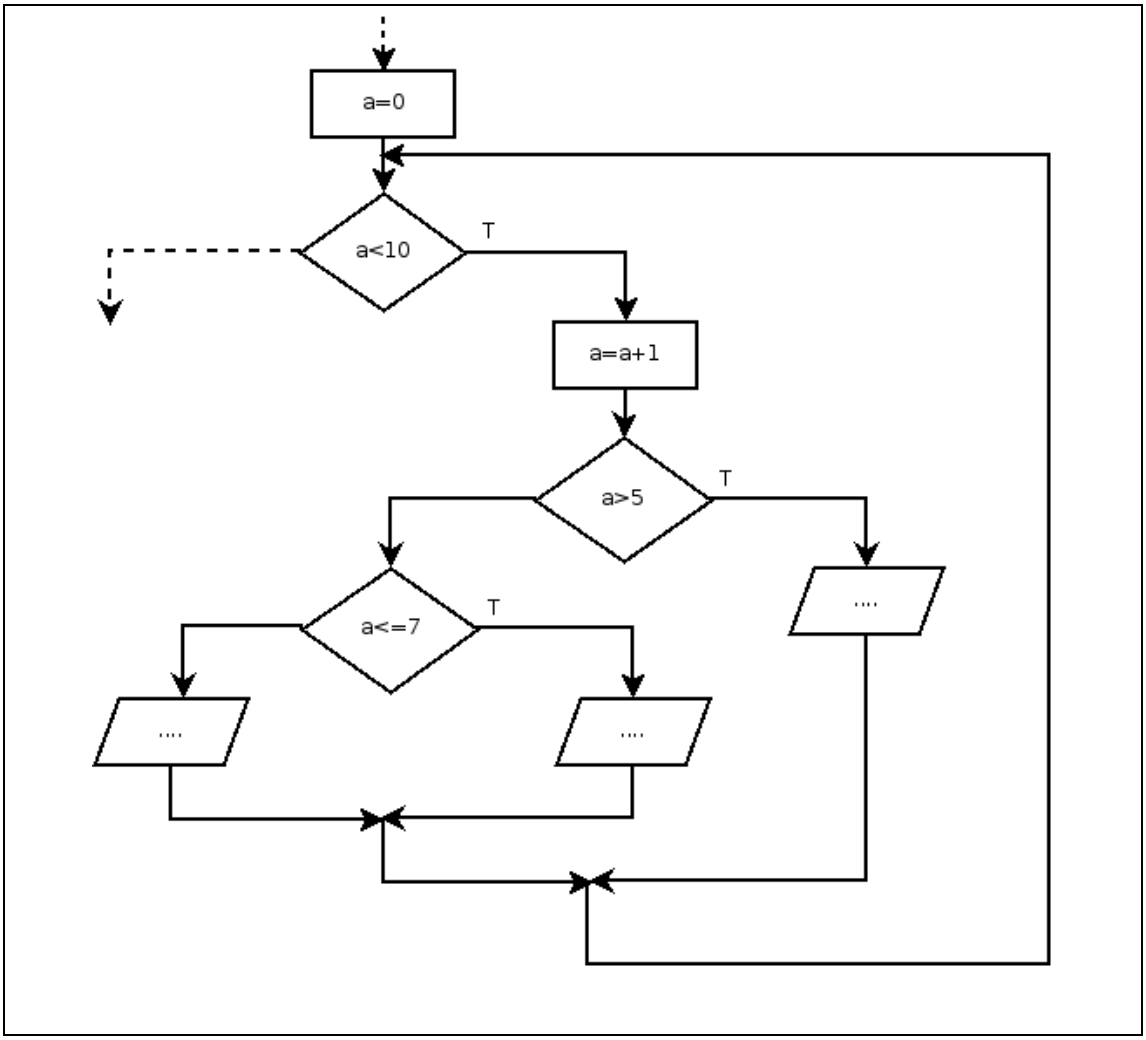
Ora esaminiamo un esempio leggermente piu’ complesso.
Scriviamo il seguente programma:

```
a=0
while a < 0:
     a=a+1
     if a > 1 :
          print a,” > “,5
     elif a <= 7:
          print a,” <= “,7
     else:
          print “nessuna condizione risulta verificata”
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
1 <= 7
2 <= 7
3 <= 7
4 <= 7
5 <= 7
6 > 5
7 > 5
8 > 5
9 > 5
10 > 5
```

Il flow-chart dell’algoritmo è il seguente:



Dall'analisi del codice e dell'output osserviamo che:

- ✓L'istruzione "elif" ci permette di semplificare la scrittura del codice nel caso in cui abbiamo piu' test in cascata.
- ✓Possiamo avere piu' livelli di indentazione, dipende dalla struttura e dalla complessita' del nostro algoritmo.

6.6.4.Operatori di confronto e operatori logici

In questo paragrafo descriveremo gli operatori di confronto e gli operatori logici del python. Questi saranno gli operatori che solitamente utilizzeremo per le nostre condizioni di test del ciclo while e dell'if.

Operatori di confronto	
<	minore
<=	mnore o uguale

>	maggiore
>=	maggiore o uguale
==	uguale
!= oppure <>	diverso

Gli operatori di confronto ci permettono di costruire una condizione di test mettendo a confronto due oggetti.

Operatori logici	
or	oppure
and	e
not	negazione

Gli operatori logici ci permettono di costruire condizioni di test complesse mettendo in relazione espressioni logiche (p.e. le condizioni di test indicate sopra).

6.7.Definizione di funzione

Scriviamo il seguente programma:

```
a=23
b=-40
def my_abs(num):
     if num < 0 :
          num=-num
     return num
if my_abs(a) >= my_abs(b) :
     print my_abs(a)
else:
     print my_abs(b)
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
40
```

Dall'analisi del codice e dell'output osserviamo che:

- ✓L'istruzione def permette di definire una funzione, nel nostro caso la funzione si chiama:"my_abs"
- ✓Non esistono i prototype per le funzioni python, si definisce la funzione e subito dopo il suo corpo.
- ✓Dopo il def è necessario digitare ":", il corpo della funzione si dovrà scrivere indentato. N.B. Il return fa parte del corpo della funzione e quindi va scritto anch'esso indentato.
- ✓La funzione può essere definita in qualsiasi posto nel codice, l'importante è che sia definita prima che venga chiamata.

✓I parametri vengono sempre passati per valore tranne in casi particolari (vedere capitolo 6.10.2)
✓I valori di ritorno si possono assegnare a delle variabili, ma si possono anche richiamare le funzioni direttamente da altre istruzioni come, p.e., l'if o la print.

Vediamo altri esempi di definizione di funzione:

```
def ciao():
     print “Ciao mondo!”

ret= ciao()
print ret
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
Ciao mondo!
None
```

Dall’analisi del codice e dell’output osserviamo che:
✓In questo caso la funzione non accetta argomenti, quindi sia nella definizione che nella chiamata si hanno solo le parentesi tonde aperte e chiuse
✓La funzione non restituisce nessun valore dato che l'istruzione return non appare. Lanciando il programma l'istruzione di assegnazione esegue la funzione ed a video appare l'argomento della print, alla variabile a non viene assegnato alcun valore. Se si stampa a si ottiene:”None” che indica valore “vuoto”.

Altro esempio in cui abbiamo una funzione che accetta più di un argomento:

```
def area_rettangolo(base,altezza):
     return base*altezza
```

In quest'altro esempio vediamo la possibilità di definire una funzione che restituisca più di un valore:

```
def valori(x,y):
     a=x+1
     b=x*y
     c=3*y+x
     return a,b,c

k,l,m=valori(2,3)
print k,l,m
print l
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
3 6 11
6
```

Dall'analisi del codice e dell'output osserviamo che:

- ✓La funzione restituisce 3 valori.
- ✓L'assegnazione a 3 variabili dei 3 valori di ritorno avviene simultaneamente.
- ✓Le avariabili assegnate possono essere utilizzate indipendentemente l'una dalle altre.

6.8.Input da tastiera

Scriviamo il seguente programma:

```
print "Chi sei?"
risposta = raw_input()
print "Benvenuto ",risposta
print "Inserisci N "
N = input()
print "N = ",N
```

Lanciamo il programma (che chiamiamo per comodità inserisci.py) , l'output stampato a video è il seguente:

```
Chi sei?
Neo
Benvenuto  Neo
Inserisci N
61
N =  61
```

Dall'analisi del codice e dell'output osserviamo che:

- ✓L'istruzione deputata ad inserire stringhe alfanumeriche è raw_input()
- ✓L'istruzione deputata ad inserire valori numerici è input()
- ✓Non appena l'interprete python incontra un'istruzione di input si blocca e aspetta un inserimento dati da parte dell'utente. Lo standard input è la tastiera.

N.B. Si può reindirizzare lo standard input con il comando di ridirezione:"<". Eseguendo da console Linux il precedente programma :

```
$ python inserisci.py < input.txt [invio]
```

dove input.txt è un file che contiene 2 righe:

```
input.txt:
Pippo
21
```

Otteniamo il seguente output:

```
Chi sei?
Pippo
Benvenuto  Pippo
Inserisci N
21
N =  21
```

Altro esempio in cui cercheremo di rendere più “gradevole” l'inserimento dati da standard input :

```
a = input(“Inserisci a: “)
b = input(“Inserisci b: “)
print “a+b = “,a+b
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
Inserisci a: 2
Inserisci b: 45
a+b =  47
```

Dall’analisi del codice e dell’output osserviamo che:

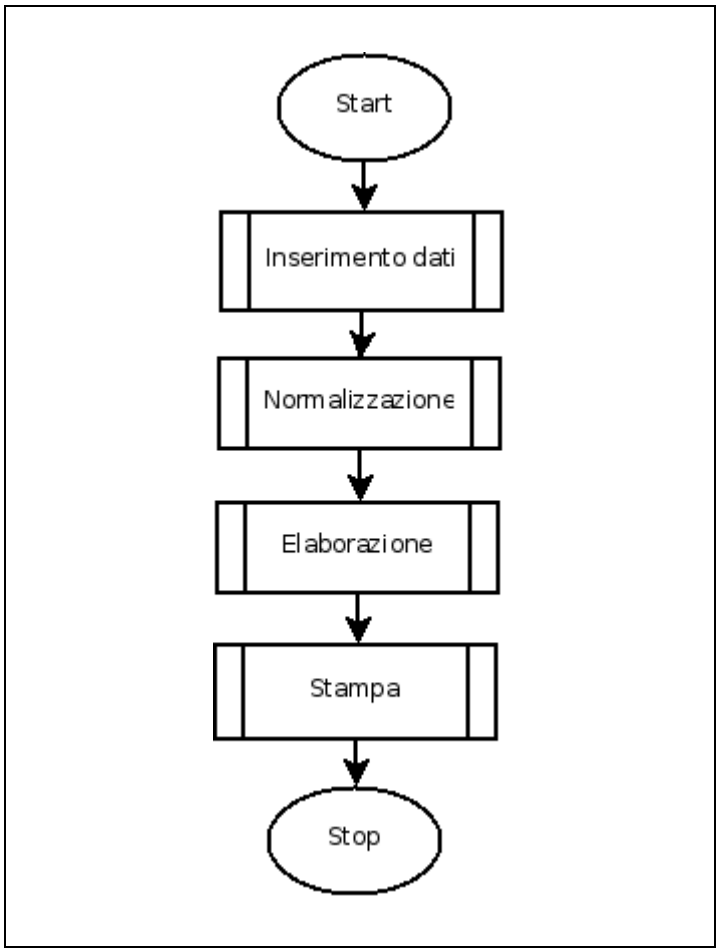
- ✓L'istruzione input può avere come argomento una stringa che può rendere più user-friendly l'inserimento del dato.
- ✓L'inserimento avviene sulla stessa riga della stringa stampata dall'istruzione input.
- ✓Analogo discorso vale anche per l'istruzione raw_input.

6.9.Importazione moduli

6.9.1.Introduzione

Quando si affronta un problema si utlizza una metodologia che fornisce gli strumenti per arrivare alla sua soluzione. Per esempio la metodologia *top-down* affronta il problema suddividendolo in problemi più piccoli, questi, a loro volta, vengono suddivisi in altri e così via, sino ad arrivare a dei problemi la cui soluzione è banale. Mettendo insieme tutte le soluzioni si compone la soluzione

al problema di partenza. Questo è un metodo ricorsivo che viene diffusamente usato per scrivere algoritmi.
Utilizzando la descrizione di un algoritmo mediante flow-chart, possiamo rappresentare con un blocco i sottoproblemi in cui abbiamo diviso il nostro problema di partenza. Il seguente macro-flow ne è un esempio per un semplice generico problema:



Nell'esempio possiamo notare quattro macroblocchi:

- Inserimento dati, in cui andiamo a specificare l'algoritmo che permette di alimentare con i dati di input la nostra procedura;
- Normalizzazione, è una sorta di pre-elaborazione, potrebbe essere utile nel caso in cui dobbiamo elaborare una serie di dati "sporchi". p.e. potremmo andare ad eliminare dai dati quei valori che non sono consoni ai valori di input da noi aspettati.
- Elaborazione, è l'algoritmo di elaborazione vera e propria. Ovviamente a seconda dei casi potrà assumere nomi diversi, l'importante è che il nome descriva l'elaborazione che l'algoritmo andrà a fare. p.e. se abbiamo la necessità di integrare numericamente una funzione reale allora il nome di questo macro-blocco potrà essere:"integrazione numerica".
- Stampa, è la fase di emissione dei risultati, quasi sempre conviene tenerla

separata dalla fase elaborativa pura.

Ogni macro-blocco può essere a sua volta spezzato in altri macro-blocchi. Dopo questo processo di analisi, ogni macro-blocco verrà descritto da un flow-chart in cui si specifica in dettaglio l'algoritmo.
Nella fase di codifica un macro-blocco verrà rappresentato mediante una funzione che esprime l'algoritmo contenuto nel macro-blocco stesso.

P.e. la codifica che rappresenta il macroflow sopra riportato potrebbe essere:

```
# definizione delle funzioni descritte dal macro-flow
def inserimento_dati(..,...,...):
     ...
     ...
     return var
def normalizzazione(..,...,...):
     ...
     ...
     return var
def elaborazione(..,...,...):
     ...
     ...
     return var
def stampa(..,...,...):
     ...
     ...
     return var

# programma principale
# definizione delle variabili
...
...
# chiamata delle funzioni
a=inserimento__dati(...,...,...)
b=normalizzazione(...,...,...)
c=elaborazione(...,...,...)
d=stampa(...,...,...)
```

Analizzando il codice osserviamo che il progamma principale è costituito soltanto dalle chiamate alle funzioni che codificano gli algoritmi dei macro-blocchi. Generalmente questa è la struttura della codifica che genera un macro-flow come quello qui descritto.
N.B.: i puntini nella codifica rappresentano tutte le parti che si sono delibaratamente omesse per non appesantire troppo la codifica di esempio, ma che, necessariamente, in una codifica reale non possono essere tralasciate.

6.9.2.I moduli del Python

Alcune volte risulta comodo utilizzare parti di codifiche scritte in precedenza, per risolvere altri problemi. Per esempio riutilizzare la procedura di normalizzazione o di stampa del precedente esempio potrebbe far risparmiare diverso tempo nello sviluppo totale del problema preso in esame.
Un modo per fare questo la cosa più semplice è quella di copiare la parte di codice che interessa. In questo modo, però, si va incontro alle diverse problematiche che un'operazione del enere comporta, basti pensare solo alla complessità di gestione delle modifiche din una procedura se presente in più codifiche contmporaneamente.

Il python risolve questo problema introducendo l'uso dei moduli.
Si deve seguire la seguente procedura per utilizzare i moduli per l'esempio riportato precedentemente:

- Si scriva il codice di una o più funzioni in un file sorgente separato
- Si scriva il sorgente del programma principale come nella codifica precedente omettendo il corpo delle funzioni.
- Si aggiungano i comandi di importazione delle funzioni nel sorgente del programma principale.

Si hanno i seguenti files sorgenti:
inserimento.py :

```
# definizione delle funzioni
def inserimento_dati(..,...,...):
     ...
     ...
     return var
def funzione_dati(..,...,...):
     ...
     ...
     return var
...
...
```

preelaborazioni.py :

```
# definizione delle funzioni

def normalizzazione(..,...,...):
     ...
     ...
     return var
def medie(..,...,...):
     ...
     ...
     return var
...
```

```
...
```

elaborazioni.py :
```
# definizione delle funzioni
def elaborazione(..,...,...):
     ...
     ...
     return var
def elaborazione2(..,...,...):
     ...
     ...
     return var
...
...
```

stampe.py :
```
# definizione delle funzioni
def stampa(..,...,...):
     ...
     ...
     return var
def stampa2(..,...,...):
     ...
     ...
     return var
...
...
```

main.py :
```
# programma principale
# importazione dei moduli
import inserimento, preelaborazioni, elaborazioni, stampe

# definizione delle variabili
...
...
# chiamata delle funzioni
a=inserimento.inserimento__dati(...,...,...)
b=preelaborazioni.normalizzazione(...,...,...)
c=elaborazioni.elaborazione(...,...,...)
d=stampe.stampa(...,...,...)
```

Analizzando il codice osserviamo che:

- ✔ il programma principale è costituito soltanto dalle definizioni di variabili e dalle chiamate alle funzioni presenti nei moduli
- ✔ le funzioni all'interno dei moduli sono definite in maniera standard
- ✔ non è necessario che all'interno di un modulo sia presente solo una funzione

- ✔ nel programma principale appare l'istruzione `import` che permette di effettuare l'importazione del codice da un file sorgente
- ✔ Nel programma principale le funzioni vengono chiamate apponendo come suffisso il nome del modulo a cui la funzione appartiene seguito da un punto (p.e. `stampe.stampa(...,...,...)`). Si noti che questa sintassi richiama la sintassi dei linguaggi orientati agli oggetti.
- ✔ l'istruzione `import` può essere inserita in qualsiasi posizione del programma principale, l'importante è che sia prima della chiamata alla funzione che contiene

6.9.3.L'istruzione `import`

Il comando che indica al Python di importare un modulo (cioè un file sorgente) è `import`, questa può essere scritta nei seguenti modi:

a) `import` *nomemodulo*
b) `from` *nomemodulo* `import` *nomefunzione1*`,`*nomefunzione2*`,...`
c) `from` *nomemodulo* `import *`

Il primo caso lo abbiamo esaminato nel precedente paragrafo, in questo caso le chiamate alle funzioni prevedono un suffisso composto dal nome del modulo a cui la funzione appartiene seguito da un punto. Con questa istruzione si va a caricare in memoria tutte le funzioni contenute nel modulo importato. Questo potrebbe dare problemi di memoria e di performance della macchina su cui gira lo script.

Nel secondo caso (b) si va a specificare quale funzione voglimo caricare dal modulo preso in considerazione. Utilizzando questa sintassi dobbiamo modificare la chiamata alla funzione, non si porrà più il suffisso ma si scriverà solo il nome della funzione. Si consideri il seguente programma che è la riscrittura di main.py:

main.py :
```
# programma principale
# importazione dei moduli
from inserimento import inserimento_dati
from preelaborazioni import normalizzazione
from elaborazioni import elaborazione
from stampe import stampa

# definizione delle variabili
...
...
# chiamata delle funzioni
a=inserimento__dati(...,...,...)
b=normalizzazione(...,...,...)
c=elaborazione(...,...,...)
```

```
d=stampa(...,...,...)
```

In questa maniera la chiamata alle funzioni presenti nel modulo risulta essere meno tediosa.

Nel terzo caso (c) si vanno ad importare tutte le funzioni presenti all'interno del modulo. La presenza del `from` permette di eseguire la chiamata alle funzioni in maniera più comoda senza dover inserire il suffisso del modulo. Bisogna far attenzione a non abusare di questa istruzione perchè si può rischiare di richiamare moduli che presentano lo stesso nome di funzioni e questo potrebbe portare a conflitti. Per chiarire quanto detto si esaminino i seguenti sorgenti:

modul1.py:
```
def fun():
        return "modul1"
```

modul2.py:
```
def fun():
        return "modul2"
```

main.py:
```
from modul1 import *
from modul2 import *
print fun()
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
modul2
```

Dall'analisi del codice e dell'output osserviamo che le istruzioni di `import` hanno generato una ridefinizione della funzione f*un()* , l'ultimo modulo importato impone la propria funzione (nel caso dell'esempio *modul2*)

6.10.Librerie del Python

Esistono molti moduli sviluppati per risolvere i problemi più disparati: dalle librerie per la manipolazione dei files audio alle librerie grafiche, dalle librerie per la gestione dei database a quelle per i socket TCP.
Sono presenti molte librerie sotto licenza GPL, non sempre sviluppate per un utilizzo con il python. Per rendere utilizzabili tali librerie con il python è necessario sviluppare il cosiddetto "wrapper", cioè un'interfaccia tra le chiamate alle funzioni della libreria con l'interprete python. In questo modo il parco librerie a disposizione del linguaggio è praticamente illimitato.
Nel seguito studieremo 3 particolari librerie:

- math

- Numeric
- pil (Python Imagin Library)

6.10.1.Modulo math

Il modulo *math* è la libreria matematica standard del python. Standard nel senso che in tutte le versioni del python ed in tutte le implementazioni risulta essere sempre presente.

Cominciamo a conoscere alcune funzioni, scriviamo il seguente programma:

```
from math import *
print ceil(2.2)
print ceil(-2.5)
print floor(2.2)
print floor(-3)
print round(2.5)
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
3.0
-2.0
2.0
-3.0
3.0
```

Dall'analisi del codice e dell'output osserviamo che:

- ✓Ceil(x) restituisce l'intero più piccolo >= x
- ✓Floor(x) restituisce l'intero più grande <= x
- ✓Round arrotonda all'intero più vicino

Funzione esponenziale, logaritmo e potenza:

math.exp(x) ⟶ e^x
math.log10(x) ⟶ log(x)
math.log(x) ⟶ ln(x)
math.pow(x,y) ⟶ x^y

N.B. `5**2= 25` `5**(-1)=` Non è valido
pow(5,2)=25 pow(5,-1)= 1/5 = 0,2

Funzioni trigonometriche e loro inverse :

math.cos(x) ⟶ cos(x)
math.sin(x) ⟶ sen(x)
math.tan(x) ⟶ tan(x)
math.acos(x) ⟶ arccos(x)
math.asin(x) ⟶ arcsen(x)
math.atan(x) ⟶ arctan(x)

Gli argomenti delle funzioni trigonometriche sono espressi in radianti.

Funzioni iperboliche :

math.cosh(x) ⟶ cosh(x)
math.sinh(x) ⟶ senh(x)
math.tanh(x) ⟶ tanh(x)

Costanti :

math.e ⟶ numero di Nepero
math.pi ⟶ pi greco

```
import Numeric
```

6.10.2.Modulo Numeric

L'interprete python implementa nativamente tre strutture informative:

- lista
- tupla
- dizionario

Gli array non esistono. Per poterli utilizzare si deve importare un modulo che li implementi. Esistono diversi moduli con cui si possono definire gli array, p.e. il modulo *array*. In questa sede introdurremo il modulo *Numeric* che oltre ad implementare gli array ha tutta una serie di funzioni appositamente studiate per il calcolo numerico. Tutte le funzioni matematiche viste nel modulo *math* sono presenti in *Numeric,* quindi risulta essere ridondante importare insieme i due moduli.

Scriviamo il seguente programma:

```
import Numeric
a=Numeric.array([[1,2,3],[4,5,6]])
print a
```

```
a[0][0]=10
print a
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
[[1 2 3]
 [4 5 6]]
[[10  2  3]
 [ 4  5  6]]
```

Dall’analisi del codice e dell’output osserviamo che:

- ✔ Il file da importare si chiama Numeric con la N maiuscola.
- ✔ *array* è la primitiva che ci permette di definire un array. Lo definiamo elencando gli elementi appartenenti all'array a partire dalla prima riga. Le parentesi quadre delimiteranno la riga.
- ✔ L'istruzione *print* riconosce che ha come argomento un'array e quindi stampa tutti gli elementi di quest'ultimo (ricordiamo che il python è un linguaggio ad oggetti). Non si ha la necessità di eseguire due loop annidati per stampare un'array bidimensionale.
- ✔ Se vogliamo accedere ad un elemento dell'array indichiamo tra parentesi quadre gli indici. Gli indici, al pari del linguaggio C , partono da 0.

Scriviamo, ora, il seguente programma:

```
numeri=Numeric.array((4,9,16),Numeric.Float)
print numeri
radici=Numeric.sqrt(numeri)
print radici
```

Lanciamo il programma, l’output stampato a video è il seguente:

```
[  4.   9.  16.]
[ 2.  3.  4.]
```

Dall’analisi del codice e dell’output osserviamo che:

- ✔ Si può indicare esplicitamente all'interprete il tipo di elementi di un' array con la direttiva *Float.*
- ✔ Si possono usare al posto delle parentesi quadre le parentesi tonde per enumerare gli elementi di un'array.
- ✔ Gli operatori matematici riconoscono argomenti di tipo array in maniera tale da venire applicati a tutti gli elementi dell'array.

Può risultare particolarmente tedioso inserire gli elementi di un array in fase

di inizializzazione, p.e. si consideri una matrice 100 righe x 1000 colonne. un modo alternativo di definire un'array evitando di inserire tutti gli elementi è quello di usare alcune funzioni studiate per il calcolo matriciale.

A tal scopo si consideri il seguente programma:

```
import Numeric
a=Numeric.zeros([2,3])
b=Numeric.ones([3,3])
print a
print b
```

Lanciamo il programma, l'output stampato a video è il seguente:

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

Dall'analisi del codice e dell'output osserviamo che:

- La funzione *zeros* dichiara un array andando ad indicare tra parentesi quadre le dimesioni dell'array stesso. Tutti gli elementi assumono il valore 0.
- La funzione *ones* esegue la stessa funzione di *zeros* con l'unica differenza che gli elementi assumono il valore 1.

Se si vuole che gli elementi siano definiti come floating-point basta aggiungere la direttiva *Float* come indicato nel seguente esempio:

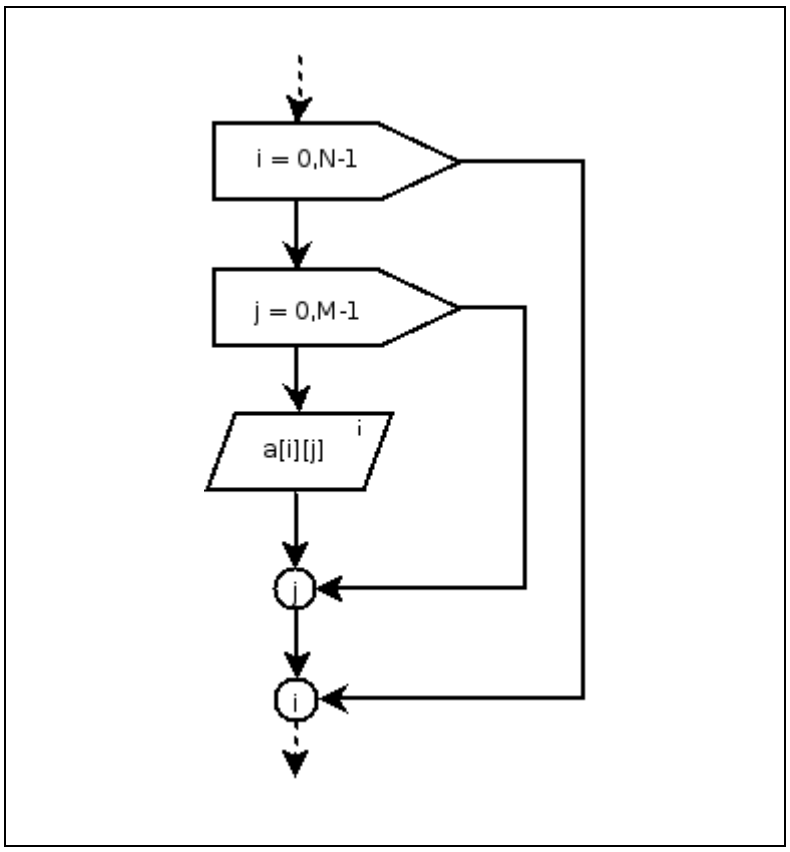
```
import Numeric
a=Numeric.zeros([2,3],Numeric.Float)
print a
```

Lanciamo il programma, l'output stampato a video è il seguente:

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Supponiamo di voler far inserire all'utente gli elementi di un'array, abbiamo visto al 6.8 come far inserire dei valori dallo standard input mediane le funzioni *raw_input()* e *input()*. Non possiamo utilizzare queste funzioni per inserire tutti gli elemnti di un'array, dobbiamo necessariamente eseguire dei loop annidati e richiedere l'inserimento dei singoli elementi.
P.e. se si vuole inserire un'array di N righe x M colonne si deve seguire il

seguente diagramma di flusso:



Il caricamento degli elementi avviene per riga, da questo flow-chart si ottiene la seguente codifica:

```
import Numeric
...
...
for i in range (0,N):
     for j in range (0,M):
          a[i][j]=input()
...
...
```

N.B. Gli estremi del ciclo for, indicati nella codifica sono differenti da quelli indicati nel flow-chart perchè la funzione range non comprende l'estrmo superiore.

6.10.3.Modulo pil

Pil è l'acronimo di Python Imaging Library, consiste in una collezione di moduli per gestire le immagini con l'interprete Python. Noi utilizzeremo il modulo Image che ci consentirà di creare un'immagine e di modificarla.

Prima di tutto si importa il modulo:

```
import Image
```

Creare un'immagine:

```
img=Image.new('RGB',(200,200),(255,255,255))
```

Dove:
il primo parametro è la modalità, si possono avere diverse modalità:

'1': pixel a 1 bit, bianco e nero
'L': pixel a 8 bit, 256 tonalità di grigio
'P': pixel a 8 bit, con una palette di 256 colori
'RGB': 3 byte per pixel, modalità true color
'RGBA': 4 byte per pixel, modalità true color con banda alpha (trasparenza)
'I': pixel interi a 32 bit
'F': pixel a virgola mobile a 32 bit

Il secondo parametro è una coppia di numeri che indica la dimensione dell'immagine in pixel, il primo numero rappresenta la dimensione orizzontale ed il secondo quella verticale.

Il terzo parametro rappresenta il colore di sfondo, se omesso il colore sarà nero. Se mode='RGB' allora si dovrà fornire una terna di numeri, se mode = 'L' si fornirà solo un numero.

img rappresenta l'handle dell'immagine al quale ci rifeririremo per qualsiasi operazione si voglia effettuare sull'immagine stessa.

Modificare un'immagine:

```
img.putpixel((x,y),(r,g,b))
```

Dove:

(x,y) rappresenta la coordinata del punto dell'immagine sulla quale vogliamo agire
(r,g,b) rappresenta la terna del colore (in modalità 'RGB')

Con putpixel si va a colorare il pixel selezionato con il colore indicato.

Salvare un'immagine:

img.save('nomeimmagine.ext')

Dove:

nomeimmagine è il pathname dell'immagine che si vuole salvare
ext è l'estensione del file, individua il tipo di file grafico che si vuole creare. p.e. se si indica 'bmp' allora si creerà un file bitmap, con 'jpg' un file jpeg, e così via.

Con queste tre primitive si riesce a creare tutti i grafici possibili.