Tecnologie dell'Informazione e della Comunicazione

Capitolo 6
Iterazione

Prof. Mauro Gaspari: gaspari@cs.unibo.it

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Assegnamenti multipli

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Assegnamento e uguaglianza

- E' importante distinguere tra l'operatore di assegnamento (=) e l'uguaglianza (che in realta' e' ==).
  - NB. l'uguaglianza e' commutativa, l'assegnamento no.
  - NB2. un uguaglianza matematica e' in genere sempre vera, mentre una volta fatto un assegnamento il valore di una variabile puo' cambiare.

```
a = 5
b = a    # a and b are now equal
a = 3    # a and b are no longer equal
```

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Il concetto di **iterazione**

- Spesso i computer sono utilizzati per eseguire in modo automatico task ripetitivi.
- Le operazioni ripetitive sono quelle che i computer riescono a fare meglio rispetto a noi (che spesso ci possiamo sbagliare).
- Tecnicamente la ripetizione di comandi si dice **iterazione**.
- Un effetto simile si puo' ottenere con la ricorsione.

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Il comando **while**

```
def countdown(n):
  while n > 0:
    print n
    n = n-1
  print "Blastoff!"
```

- NB. la ricorsione e' stata levata.
- Si puo' leggere il comando **while** in modo letterale: while significa "**fino a quando**" ed e' associato ad una condizione.
- Fino a quando la condizione vale si esegue il body del while.

Semantica del while

1) Si valuta la condizione booleana.

2) Se la condizione e' falsa (0): si esce dal while e si continua con l'esecuzione del comando successivo.

3) Se la condizione e' vera (1): si eseguono tutti i comandi del body e alla fine si torna al passo 1.

- Questo tipo di flusso di controllo si chiama anche **ciclo** (= **loop)**, perchè con il passo 3) si torna indietro, all'inizio del comando.

Osservazioni sul loop.

- NB. Se la condizione e' falsa i comandi del body non vengono mai eseguiti.
- E' opportuno che nel body ci siano dei comandi che cambiano il valore di variabili che appaiono nella condizione.
- Altrimenti si ottengono dei **loop infiniti**.

Esempio

```
def sequence(n):
  while n != 1:
    print n,
    if n%2 == 0:           # n is even
      n = n/2
    else:                  # n is odd
      n = n*3+1
```

Questo programma termina sempre?

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Osservazioni

- Dato che a volte il numero cresce e a volte decresce non e' facile dimostrare in modo semplice che il programma termina sempre.
- E' possibile dimostrare la terminazione c'e' per qualche n. Ad esempio se n e' una potenza di 2.
- Ma e' possibile dimostrare che il programma termina per tutti i valori di n?
- **Fino ad ora nessuno e' riuscito a dimostrare che questa affermazione vale e nemmeno che questa non vale.**

Non terminazione e decidibilita'

- Un problema e' **decidibile** quando si riesce sempre a dare una risposta.
- Ad esempio si puo' dimostrare e quindi decidere che il programma countdown visto prima termina sempre con gli interi positivi.
- Una proprieta' importante che vale nell'informatica e' la seguente: dato un qualsiasi programma non e' sempre possibile stabilire se questo termina per tutti i possibili input (**problema della non terminazione - delle macchine di Turing**).
- Quindi il problema della terminazione non e' decidibile.

Decidibile e semidecidibile

- Come facciamo a stabilire se un problema e' decidibile?
- Se troviamo un programma (algoritmo) che termina sempre un problema e' **decidibile**.
- Se invece troviamo un programma che potrebbe non terminare ma termina sempre se la risposta al problema e' SI. Il problema si dice **semidecidibile**.

Problemi semidecidibili

- Un esempio di problema semidecidibile: **dire se un programma ha un errore a tempo di esecuzione** (a runtime)?
  - Se il programma termina perche' si verifica l'errore allora la risposta e' SI.
  - Se il programma termina correttamente la risposta e' NO.
  - Pero' un programma potrebbe non terminare, in questo caso non sono in grado ne di dare una risposta positiva ne negativa.
  - Sono pero' sicuro che se la risposta e' SI prima o poi riesco a trovarla (ovvero se c'e' un errore a tempo di esecuzione questo prima o poi si verifica, anche dopo giorni).

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Problemi indecidibili

- Ci sono problemi che non sono ne decidibili, ne semidecidibili, questi si dicono indecidibili (ovvero insolubili, non calcolabili).
- Ad esempio il problema inverso a quello nella precedente slide: determinare se un programma non ha errori a tempo di esecuzione (determinare se un programma e' corretto a runtime) non e' ne decidibile ne semidecidibile.
- Un altro problema indecidibile che abbiamo visto in precedenza e' il problema della terminazione di un qualsiasi programma con un qualsiasi input.
- Un altro problema indecidibile e' quello di stabilire se due programmi qualsiasi danno gli stessi risultati.

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Tabelle

- I loop sono ottimi per generare tabelle.
- Una volta ad esempio le funzioni matematiche come seno e coseno venivano calcolate con l'aiuto di tabelle.
- Si puo' usare un computer per generare queste tabelle, ma ovviamente il computer puo' fare di piu' calcolare direttamente le funzioni! Per noi le tabelle sono diventate obsolete.
- A volte pero' i computer le usano per fare approssimazioni.
- Un esempio e' la tabella del processore intel-pentium usata per fare operazioni floating-point.

Esempio di tabella

- Questo programma restituisce una sequanza di valori nella colonna a sinistra e il loro logaritmo nella colonna a destra.

```
import math

x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Tabelle bidimensionali

- Una tabella bidimensionale e' una tabella con piu' righe e colonne in cui in genere si leggono i valori all'intersezione tra righe e colonne.

- Esempio: tabella moltiplicazione da 1 a 6.

```
i = 1
while i <= 6:
  print 2*i, '    ',
  i = i + 1
print
```

Incapsulamento e generalizzazione

- L' **incapsulamento** di un pezzo di codice si ottiene inserendo quel codice in una funzione.
- La generalizzazione si ottiene prendendo un pezzo di codice specifico e rendendolo piu' generale, adotto cioe' a risolvere altri problemi simili.
- Ad esempio si puo' generalizzare un pezzo di codice che stampa i multipli di 2 per fargli stampare i multipli di un numero qualsiasi.

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Esempio

- Questa funzione incapsula il codice visto in precedenza e lo generalizza per trattare multipli di qualsiasi numero.

```
def printMultiples(n):
  i = 1
  while i <= 6:
    print n*i, '\t',
    i = i + 1
  print
```

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Osservazioni

- Per incapsulare un pezzo di codice quello che bisogna fare e' scrivere la prima linea con il nome della funzione e la lista dei parametri.
- Per generalizzare si rimpiazzano costanti con parametri.

Il risultato:

```
i = 1
while i <= 6:
  printMultiples(i)
  i = i + 1
```

E' possibile incapsulare e generalizzare ancora?

La stampa di una tabella

```
def printMultTable():
  i = 1
  while i <= 6:
    printMultiples(i)
    i = i + 1
```

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Metodologia di sviluppo

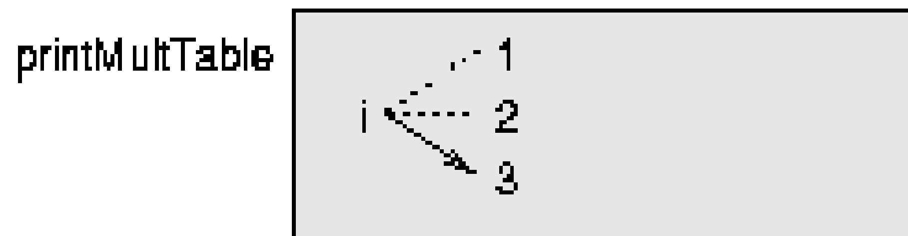
- Prima si scrive codice isolato che risolve un certo **sottoproblema** ed eventualmente si prova nell'interprete.
- Una volta testato si incapsula il codice in una funzione.
- Dopo di che si usa quella funzione per risolvere un problema piu' grosso.
- Questo procedimento e' molto utile per cominciare a risolvere un problema anche quando non si hanno le idee chiare riguardo ai sottoproblemi in cui suddividerlo.

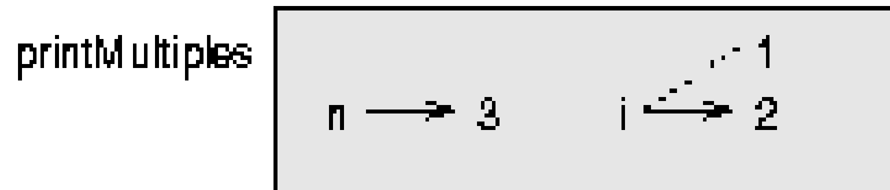
Osservazioni

- Si usa la stessa variabile nelle due funzioni ma questo non crea problemi: **le variabili sono locali alle singole funzioni**.
- Non e' possibile accedere alle variabili locali al di fuori della funzione dove sono definite.
- Questo si puo' illustrare con il diagramma a stack.

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Diagramma a stack





© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

Ancora generalizzazione

- E' possibile realizzare una tabella della moltiplicazione di qualsiasi dimensione?

```
def printMultTable(high):
  i = 1
  while i <= high:
    printMultiples(i)
    i = i + 1
```

Mi serve una tabella NxN

```
def printMultiples(n, high):
  i = 1
  while i <= high:
    print n*i, '\t',
    i = i + 1
  print

def printMultTable(high):
  i = 1
  while i <= high:
    printMultiples(i, high)
    i = i + 1
```

© Mauro Gaspari - University of Bologna - gaspari@cs.unibo.it

A cosa servono le funzioni?

- Dare un nome a una sequenza di comandi per rendere un programma piu' semplice da leggere e da debuggare.
- Dividere un programma in sottoproblemi (sottoprogrammi) che si possono provare isolatamente e comporre.
- Semplificano sia l'iterazione che la ricorsione.
- Si possono riutilizzare in altri programmi.