Informatica per Statistica
Riassunto della lezione del 15/11/2013

Igor Melatti

Altre istruzioni e costrutti del Python

- **L'istruzione `break` ha l'effetto di interrompere il ciclo più interno nel quale venga eseguita; dopodiché si continua con l'istruzione successiva al ciclo interrotto**
  - è pertanto sufficiente posizionare un `break` alla fine di un blocco istruzioni per far sì che venga eseguito solo quello
- Altre due istruzioni simili al `break`: `return` <`espressione`> e `continue`
- **L'istruzione `return` E ha l'effetto di valutare l'espressione E; ne sia v il valore. Dopodiché, interrompe la funzione nella quale venga eseguita, ritorna v alla funzione chiamante (o alle istruzioni principali)**
- **se l'istruzione `return` si trova all'interno non di una (definizione di) funzione, ma delle istruzioni principali, viene ritornato errore**
  - si può anche omettere l'espressione, e in quel caso viene ritornata la costante `None`
  - se la chiamata di funzione si trovava all'interno di una espressione più complessa, allora si procede a valutare il resto di questa espressione
  - ad esempio, nella seguente chiamata a funzione: `var = 4*f1(var2) + var3;`, dopo che `f1` ha tornato il suo valore, occorre ancora moltiplicarlo per 4 e sommargli il valore di `var3`
- **L'istruzione `continue` ha l'effetto di interrompere l'esecuzione del blocco di istruzioni del ciclo più interno nel quale venga eseguita, tornando subito all'intestazione del ciclo stesso**
  - questo vuol dire che se viene fatto dentro un `for` si procede automaticamente con l'istruzione successiva
  - vedere la differenza tra

```
for i in range(0, 10):
  print "Questa e' l'iterazione " + i
```

e

```
for i in range(0, 10):
  if (i == 3):
    continue
  print "Questa e' l'iterazione " + i
```

- **esercizio:** come si può scrivere un ciclo `for` equivalente a quest'ultimo, senza usare `continue`?
- se invece si è dentro un `while`, occorre fare più attenzione: si potrebbe ciclare per sempre
- vedere la differenza tra

```
i = 0
while i < 10:
  print "Questa e' l'iterazione " + i
  i = i + 1
```

e

```
i = 0
while i < 10:
  if (i == 3):
    continue
  print "Questa e' l'iterazione " + i
  i = i + 1
```

- **esercizio:** correggere il ciclo `while` precedente
- **esercizio:** cosa succede sostituendo `continue` con `break` negli esempi precedenti?
- **esercizio:** e sostituendolo con una `return`?

Le Matrici in Python

- Una *matrice* è una struttura dati in grado di memorizzare una matrice con m righe ed n colonne in $I^{m \times n}$, dove I è un opportuno insieme (si supporrà nel seguito che sia numerico, ma non è necessario)
- Viene rappresentata in modo tale da poter distinguere tra righe e colonne, usando due indici come nelle notazioni matematiche
  - concettualmente è come se le sequenze fossero unidimensionali (viene usato un solo indice) e le matrici fossero bidimensionali (vengono usati 2 indici)

- ad esempio, la matrice (matematica) $M = \begin{pmatrix} 3 & 4 & 5 \\ 13 & 14 & 15 \end{pmatrix}$ è tale che $M \in \mathbb{N}^{2\times 3}$, e ad esempio $M_{2,2} = 14$ e $M_{1,3} = 5$
- in pseudocodice, il fatto che $M \in \mathbb{N}^{2\times 3}$ è rappresentato dal fatto che $\text{cols}[M] = 3$ e $\text{rows}[M] = 2$
- ovvero, anziché avere 1'attributo length[A] come accadeva per le sequenze A, per le matrici ci sono 2 attributi che dicono il numero di righe (rows) ed il numero di colonne (cols)
- si dà per scontato che ogni riga abbia lo stesso numero di colonne e ogni colonna lo stesso numero di righe
- per accedere all'elemento che si trova sulla colonna j della riga i di M, si può scrivere $M[i][j]$
- quindi $M[2][2] = 14$ e $M[1][3] = 5$

- Problema della somma di matrici intere (o comunque numeriche)

  Input: due matrici A e B in $\mathbb{Z}^{m\times n}$

  Output: una matrice C t.c. $C_{i,j} = A_{i,j} + B_{i,j}$ per ogni $i \in [1, m]$ e $j \in [1, n]$

- Dimensione dell'input: numero di righe m e numero di colonne n
- L'algoritmo che risolve il problema dato è in Figura 1

```
1 SommaMatrici(A, B)
2   if (rows[A] ≠ rows[B])
3     return DIM_ERROR
4   if (cols[A] ≠ cols[B])
5     return DIM_ERROR
6   for i ← 1 to rows[A]
7     for j ← 1 to cols[A]
8       C[i][j] ← A[i][j] + B[i][j]
9   return C
```

Figure 1: Somma tra matrici

- come prima cosa, controlla che le dimensioni siano corrette (si possono sommare due matrici solo se le dimensioni coincidono)
- **esercizio**: scrivere un unico `if` e un unico `return` per la condizione di errore
- dopodiché scorre per intero sia A che B, costruendo di volta in volta il risultato C
- per farlo, occorrono due cicli: uno su i che individua le righe, e uno su j che all'interno della riga i individua una colonna

- La complessità è ovviamente $\Theta(nm)$, risultante dal doppio ciclo for (uno di lunghezza n, l'altro di lunghezza m)
  - da notare che non ci sono casi migliori o peggiori (a meno di considerare un input con due matrici non delle stesse dimensioni...)
- In Python, le matrici si possono rappresentare come *liste di liste* (o *sequenze di sequenze*)
  - la matrice M di cui sopra può essere quindi creata con l'assegnamento `M = [[3, 4, 5], [13, 14, 15]]`
  - da notare che l'espressione `M[0]` viene valutata come `[3, 4, 5]`, e l'espressione `M[1]` viene valutata come `[13, 14, 15]` (provarlo sull'interprete iterattivo)
  - quindi, l'operatore di dereferenziazione sta ritornando una sequenza, anziché un singolo elemento com'è stato finora
  - questo perché `M` è stata definita come `[[3, 4, 5], [13, 14, 15]]`, quindi è composta da due elementi, che sono a loro volta delle sequenze
  - l'espressione `len(M)` viene valutata uguale a 2, dato che due sono effettivamente gli elementi di `M`: si tratta di `[3, 4, 5]` (che è il primo) e `[13, 14, 15]` (che è il secondo)
  - quindi `len(M)` restituisce il numero di righe di `M`, se `M` è una matrice
  - per sapere il numero di colonne, occorre considerare un qualsiasi elemento di `M` (ovvero, una qualsiasi delle sequenze che lo compongono) e vedere quanti elementi ha
  - dato che almeno un elemento, ovvero una riga, c'è sempre (altrimenti si avrebbe la matrice vuota, che non ha senso matematico), si può quindi usare l'espressione `len(M[0])`
  - nell'esempio di cui sopra, dato che `M[0]` viene valutata come `[3, 4, 5]`, si ha che `len(M[0])` viene valutata 3, che è effettivamente il numero di colonne
  - per accedere ad un elemento in particolare, si procede come nello pseudocodice (ma con gli indici scalati di 1...), ovvero `M[1][1] = 14` e `M[0][2] = 5`
  - questa notazione è perfettamente coerente con le liste di Python: ad esempio, dato che `M[0]` è valutato `[3, 4, 5]`, il terzo elemento di `M[0]` (ovvero quello all'indice 2) è 5
  - quindi `(M[0])[2]` è 5, e si può scrivere più brevemente come `M[0][2]`
- Creazione di una matrice in Python
  - non si sempre si può usare un assegnamento ad una lista di liste costante come sopra (`M = [[3, 4, 5], [13, 14, 15]]`)

- più spesso, occorre creare man mano la matrice `M`, aggiungendo di volta in volta righe e colonne
- a tal proposito si può procedere come segue
- si supponga di costruire la matrice per righe; quindi nell'esempio di cui sopra vengono dati prima il 3, poi il 4, poi il 5, poi si dice che è finita la prima riga, poi il13, poi il 14, poi il 15 e infine si dice che è finita la seconda riga e l'intera matrice
- corrispondentemente, la matrice `M` assumerà i seguenti valori: `[[3]]`, `[[3, 4]]`, `[[3, 4, 5]]`, `[[3, 4, 5], [13]]`, `[[3, 4, 5], [13, 14]]`, `[[3, 4, 5], [13, 14, 15]]`
- ovviamente, non si può cominciare assegnando `M[0][0] = 3`: esattamente come un assegnamento `A[0] = 3` darebbe errore, dato che le liste `M` ed `A` non hanno un primo elemento (sono vuote)
- occorre quindi riservare spazio per il primo elemento; si potrebbe pensare di assegnare `M = []` e poi fare `M = M + [3]` come per le liste monodimensionali (sequenze)
- tuttavia, questo creerebbe una sequenza per l'appunto monodimensionale, quindi fatta così: `[3]`
- invece, qui serve che il primo passo sia `[[3]]`
- inoltre, per far funzionare il tutto, occorre in realtà considerare la situazione di partenza (lista vuota); quindi in realtà `M` dovrà assumere i seguenti valori:
  1. `[]`,
  2. `[[]]`,
  3. `[[3]]`
  4. `[[3, 4]]`
  5. `[[3, 4, 5]]`
  6. `[[3, 4, 5], []]`
  7. `[[3, 4, 5], [13]]`
  8. `[[3, 4, 5], [13, 14]]`
  9. `[[3, 4, 5], [13, 14, 15]]`
- a tal proposito, è sufficiente effettuare i seguenti assegnamenti successivi
  1. `M = []`,
  2. `M = M + [[]]`,
  3. `M[0] = M[0] + [3]`
  4. `M[0] = M[0] + [4]`
  5. `M[0] = M[0] + [5]`
  6. `M = M + [[]]`,
  7. `M[1] = M[1] + [13]`

8. `M[1] = M[1] + [14]`
9. `M[1] = M[1] + [15]`

– **esercizio**: quali riordinamenti della successione di assegnamenti di cui sopra producono lo stesso risultato di quest'ultima?

- Si può ora dare il codice Python che implementa l'algoritmo di cui a Figura 2

```
1 def sum(A, B):
2     C = []
3     for i in range(0, len(A)):
4         C = C + [[]]
5         for j in range(0, len(A[0])):
6             C[i] = C[i] + [A[i][j] + B[i][j]]
7     return C
```

Figure 2: Somma tra matrici

- **esercizio:** scrivere pseudocodice e codice di altre operazioni sulle matrici (prodotto, determinante...)
- **esercizio:** scrivere una funzione Python che prenda un argomento e stabilisca se è una matrice (ovvero, se è una sequenza semplice, oppure se è una sequenza di sequenze tutte con la stessa lunghezza)