a cura di Francesco Romani

Stili di programmazione in *Mathematica*

A grande (!) richiesta sospendiamo la serie di esempi "difficili" per trattare ancora una volta alcune delle principali caratteristiche della programmazione in Mathematica.

Introduzione

Uno dei vantaggi di *Mathematica* è la presenza di molti diversi costrutti di programmazione che permettono a chiunque di "quasi-continuare a programmare" nel linguaggio che conosceva (C, Pascal, Lisp, etc.). Purtroppo però i programmi che ne derivano sono poco eleganti, difficili da leggere e decisamente poco efficienti.
D'altra parte *Mathematica* è un linguaggio interpretato, che fa uso di una notevole quantità di routine interne scritte in linguaggio a basso livello (C++ ottimizzato). Solo utilizzando queste ultime ogni volta che è possibile si riesce a conciliare efficienza, facilità di programmazione ed eleganza del programma.

Definizione di Funzioni

Cosa rappresenta l'espressione seguente?
In[1]:=

```
a x^2 + b x+ c
```

La maggior parte dei lettori (memore degli studi liceali) risponderà: "un polinomio di secondo grado in **x** dipendente dai parametri **a**, **b**, **c**". Una breve meditazione mostra come tale affermazione abbia solo motivazioni psicologiche, perché nulla nel linguaggio afferma che **x** è una variabile indipendente e **a** un parametro (e bisognerebbe pure definire prima rigorosamente questi concetti). Se invece scriviamo:
In[2]:=

```
f[x_] := a x^2 + b x + c
```

è univocamente stabilito che **f** è una funzione di **x** mentre **a**, **b**, **c** sono parametri che possono essere definiti esternamente oppure rimanere in forma simbolica.
Esiste un terzo modo di scrivere un polinomio di secondo grado. L'espressione
In[3]:=

```
a #^2 + b #+ c &
```

rappresenta una funzione pura in cui il simbolo # rappresenta la variabile indipendente a cui non viene attribuito un nome. Vedremo tra poco l'estrema utilità di questa notazione per scrivere programmi compatti.

Definizioni che ricordano i valori calcolati

Si tratta di un "trucco" che basterebbe da solo per giustificare la pena di imparare a programmare in *Mathematica*. Quando in una definizione di funzione usiamo l'operatore := si intende che il secondo membro deve essere valutato **ogni volta** che la funzione viene chiamata. Quando usiamo l'operatore = si intende che il secondo membro deve essere valutato **una sola volta** al momento della definizione. Se scriviamo `f[x_]:= (f[x] = ...)` al momento della valutazione di `f[...]` il valore assegnato viene assegnato una volta per tutte e in futuro non ci sarà più bisogno di valutarlo di nuovo.
Vediamo come esempio la classica definizione ricorsiva dei numeri di Fibonacci che abbia complessità lineare (e non esponenziale):
In[4]:=

```
Clear[f];
f[0]=0;
f[1]=1;
f[n_]:=(f[n]=f[n-1]+f[n-2])
```

Dopo l'esecuzione di questa cella **f** ha due valori speciali e uno generale:
In[8]:=

```
??f
```

Out[8]=

```
f[0] = 0
f[1] = 1
f[n_] := f[n] = f[n - 1] + f[n - 2]
```

Se calcoliamo `f[4]` i valori calcolati vengono incorporati nella definizione come se fossero stati dati esplicitamente:
In[9]:=

```
f[4]
```

Out[9]=

```
3
```

In[10]:=

```
??f
```

Out[10]=

```
Global`f
f[0] = 0
f[1] = 1
f[2] = 1
f[3] = 2
f[4] = 3
f[n_] := f[n] = f[n - 1] + f[n - 2]
```

Definizioni condizionali

Nelle definizioni di funzione si può specificare che la definizione va utilizzata se gli argomenti hanno una certa forma. Nell'ultima definizione il *pattern* ___ (tre volte il simbolo *underscore*) sta a significare un numero qualsiasi di argomenti;

in pratica **f[___]** "cattura" qualsiasi cosa:
In[11]:=
```
Clear[f];
f[0] = "Zero";
f[x_Integer] = "Intero non nullo";
f[s_String] = "Stringa";
f[{x_,y_}] = "Coppia";
f[___] = "Errore";
```
In[17]:=
```
f[0]
```
Out[17]=
```
Zero
```
In[18]:=
```
f[2]
```
Out[18]=
```
Intero non nullo
```
In[19]:=
```
f["2"]
```
Out[19]=
```
Stringa
```
In[20]:=
```
f[{a,6}]
```
Out[20]=
```
Coppia
```
In[21]:=
```
f[x]
```
Out[21]=
```
Errore
```

Liste, Matrici e Vettori

Supponiamo di dover costruire un vettore di 10.000 zeri consecutivi. Un modo semplice (e molto ingenuo) potrebbe essere di partire con la lista vuota e aggiungere uno per volta tutti gli zeri desiderati.
In[1]:=
```
LL={};
Do[AppendTo[LL,0],{10000}];//Timing
```
Out[2]=
```
{26.2167 Second, Null}
```
Il costrutto **Array** permette di costruire un vettore delle dimensioni volute valutando una funzione sugli interi:
In[3]:=
```
LL=Array[f,{10}]
```
Out[3]=
```
{f[1],f[2],f[3],f[4],f[5],f[6],f[7],f[8],f[9]
,f[10]}
```
In questo modo utilizzando la funzione pura **0&** (che vale sempre zero) si può costruire una lista dieci volte più lunga in meno di un secondo con uno speed-up di circa un fattore 300:
In[4]:=
```
LL=Array[0&,{100000}];//Timing
```
Out[4]=
```
{0.966667 Second, Null}
```

Matrici a banda

Il costrutto **Array** può essere usato anche per costruire matrici con struttura. Definiamo una funzione **a[i,j]** che vale **2** se i=j, **1** se i=j±1 e **0** altrimenti. (Si noti l'uso della condizione posta dopo i simboli **/;**):
In[5]:=
```
Clear[a];
a[i_,i_]:=2;
a[i_,j_]:=1/;Abs[i-j]==1
a[___]:=0;
```
Utilizzandola in **Array** si ottiene una matrice tridiagonale di Toeplitz:
In[9]:=
```
A=Array[a,{6,6}]
```
Out[9]=
```
2   1   0   0   0   0
1   2   1   0   0   0
0   1   2   1   0   0
0   0   1   2   1   0
0   0   0   1   2   1
0   0   0   0   1   2
```

Matrici di Toeplitz

Utilizzando una funzione generica **f** applicata al valore assoluto della differenza degli indici si ottiene la generica matrice di Toeplitz simmetrica:
In[10]:=
```
A=Array[f[Abs[#1-#2]]&,{6,6}]
```
Out[10]=
```
f[0]   f[1]   f[2]   f[3]   f[4]   f[5]
f[1]   f[0]   f[1]   f[2]   f[3]   f[4]
f[2]   f[1]   f[0]   f[1]   f[2]   f[3]
f[3]   f[2]   f[1]   f[0]   f[1]   f[2]
f[4]   f[3]   f[2]   f[1]   f[0]   f[1]
f[5]   f[4]   f[3]   f[2]   f[1]   f[0]
```

Matrici circolanti

Se invece di **Abs** si usa **Mod** si ottiene una generica matrice circolante (ogni riga successiva alla prima è ottenuta per *shift* circolare della precedente):
In[11]:=
```
A=Array[f[Mod[#2-#1,6]]&,{6,6}]
```
Out[11]=
```
f[0]   f[1]   f[2]   f[3]   f[4]   f[5]
f[5]   f[0]   f[1]   f[2]   f[3]   f[4]
f[4]   f[5]   f[0]   f[1]   f[2]   f[3]
f[3]   f[4]   f[5]   f[0]   f[1]   f[2]
f[2]   f[3]   f[4]   f[5]   f[0]   f[1]
f[1]   f[2]   f[3]   f[4]   f[5]   f[0]
```

Map, Apply e Scan

Quando si deve lavorare sulle liste è forte la tentazione di continuare ad usare i costrutti classici (**For, Do, While**) accedendo ad un elemento per volta. In questi casi il fatto che *Mathematica* sia un linguaggio interpretato viene duramente scontato con una pesante inefficienza. Se però si usano le funzioni scritte per operare sulle liste si sfrutta in pieno l'efficienza dei programmi a basso livello che formano il *Kernel di Mathematica*. Vediamo qualche esempio.
Innanzi tutto ripuliamo **f...**
In[1]:=
```
Clear[f]
```
... e costruiamoci con **Range** un vettore (piccolo per non incasinare le stampe):
In[2]:=
```
vet=Range[6]
```
Out[2]=
```
{1, 2, 3, 4, 5, 6}
```
Map[f,vet] applica una funzione f ad ogni elemento del vettore. La stessa cosa è automatica se **f** gode dell'attributo **Listable** (come per esempio **Plus**, **Abs**, ecc.):
In[3]:=
```
Map[f,vet]
```
Out[3]=
```
{f[1], f[2], f[3], f[4], f[5], f[6]}
```
Si può scrivere anche in forma abbreviata:

In[4]:=
```
f/@vet
```
Out[4]=
```
{f[1], f[2], f[3], f[4], f[5], f[6]}
```
Apply sostituisce il primo argomento all'espressione che riceve come secondo argomento. Se l'espressione è un vettore, questo diviene l'insieme degli argomenti della funzione:
In[5]:=
```
Apply[f,vet]
```
Out[5]=
```
f[1, 2, 3, 4, 5, 6]
```
Anche in questo caso esiste la forma abbreviata:
In[6]:=
```
f@@vet
```
Out[6]=
```
f[1, 2, 3, 4, 5, 6]
```
Al contrario se si usa **List** si può trasformare una funzione aritmetica in una lista:
In[7]:=
```
Apply[List,a+b]
```
Out[7]=
```
{a,b}
```
Scan[f,vet] esegue **f** su ogni elemento della lista **vet** senza restituire nulla. In questo caso quello che interessa sono solo gli effetti di bordo:
In[8]:=
```
Scan[Print,vet]
```
```
1
2
3
4
5
6
```
Un pochino più complicato l'uso di **NestList[f,expr,n]**. Viene restituita una lista di **n** elementi formata da **expr**, **f[expr]**, **f[f[expr]]**, ...:
In[9]:=
```
NestList[f,0,6]
```
Out[9]=
```
{0,f[0],f[f[0]],f[f[f[0]]],f[f[f[f[0]]]],f[f[
f[f[f[0]]]]], f[f[f[f[f[f[0]]]]]]}
```
Analogamente **FoldList[f,expr,vet]** restituisce la lista formata da **expr**, **r1=f[expr, vet[[1]]]**, **f[r1,vet[[2]]]**, ecc.:
In[10]:=
```
FoldList[f,0,vet]
```
Out[10]=
```
{0, f[0, 1], f[f[0, 1], 2], f[f[f[0,
1], 2], 3], f[f[f[f[0, 1], 2], 3], 4],
f[f[f[f[f[0, 1], 2], 3], 4], 5],
f[f[f[f[f[f[0, 1], 2], 3], 4], 5], 6]}
```
L'uso tipico di **FoldList** è l'accumulo delle somme parziali di un vettore:
In[11]:=
```
FoldList[Plus,0,vet]
```
Out[11]=
```
{0, 1, 3, 6, 10, 15, 21}
```
Esistono anche le funzioni **Nest** e **Fold**, che restituiscono l'ultimo elemento calcolato invece dell'intera lista, e **FixedPoint**, che è come **Nest**, ma si ferma quando due valori successivi sono uguali (ovvero quando si è trovato il punto fisso di **f**).

Norme di matrici e vettori

La **Norma 1** di un vettore è definita come la somma dei valori assoluti dei suoi elementi:
In[12]:=
```
n1[x_List]:=Apply[Plus,Map[Abs,x]]
```
ovvero in forma compatta:
In[13]:=
```
n1[x_List]:=Plus@@Abs[x]
```
La **Norma Infinito** di un vettore è definita come il massimo dei valori assoluti dei suoi elementi:
In[14]:=
```
ni[x_List]:=Max[Abs[x]]
```
La **Norma Infinito** di una matrice è più complicata: è definita come il massimo delle norme 1 delle sue righe. In questo caso, per distinguere le due possibilità si può usare nella definizione il predicato MatrixQ (che vale True se l'argomento è una matrice):
In[15]:=
```
ni[A_?MatrixQ]:=Max[n1/@A]
```
In[16]:=
```
ni[{1,2,13}]
```
Out[16]=
```
13
```
Si vede subito che **NON FUNZIONA**, nel calcolo seguente il risultato dovrebbe essere 20 e non 13:
In[17]:=
```
ni[{{1, 2,13},
    {6,-4,10}}]
```
Out[17]=
```
13
```
L'errore è stato di definire il caso meno generale (la matrice che è anche una lista) **dopo** quello più generale. Mettendo le definizioni nell'ordine giusto torna tutto:
In[18]:=
```
Clear[ni];
ni[A_?MatrixQ]:=Max[(Plus@@Abs[#])&/@A]
ni[x_List]:=Max[Abs[x]]
```
In[21]:=
```
ni[{1,2,13}]
```
Out[21]=
```
13
```
In[22]:=
```
ni[{{1, 2,13},
    {6,-4,10}}]
```
Out[22]=
```
20
```

Bibliografia

Antognini, P., Barozzi, G.C. **Matematica & Mathematica**. Zanichelli 1995.
Banzi, M. **Usare Mathematica**. Jackson Libri 1993.
Falco, G. **Mathematica**: Principi ed applicazioni. Addison Wesley, 1993.
Gray, J. **Mastering Mathematica**. AP Professional.
Maeder, R. **Programming in Mathematica III Edition**. AP Professional.
Maeder, R. **The Mathematica Programmer II**. AP Professional (con CD-ROM).
Maeder, R. **The Mathematica Programmer**. AP Professional (con Floppy Disk).
Wagon, S. **Guida a Mathematica**, McGraw Hill Italia, 1995 (traduzione di Mathematica in Action con riadattamento dei programmi alla versione 2.0).
Wolfram, S. **The Mathematica Book, 3rd ed**. Wolfram Media/Cambridge University Press, 1996.

MOA CASE & COSE

MOSTRA DI ARREDO
RISTRUTTURAZIONE
BIOARCHITETTURA
INFORMATICA

DAL 22 AL 30 MAGGIO

La mostra vuole dare differenti chiavi di lettura del modus vivendi dell'uomo alle porte del 2000. Dall'arredamento al rapporto Architettura-Uomo-Ambiente, dalla realtà informatica a quella di approccio della Capitale con il Giubileo.
Vieni a MOA Case & Cose, puoi vincere uno dei viaggi premio(*)

Turisanda dal 1984

Fiera di Roma

Ingresso Feriali 16.00- 23.00
Ingresso Sabato 10.00 - 23.00
Ingresso Domenica 10.00 - 22.00

(*) Aut. Min. Rich.

Presentando questo Coupon alle casse della Mostra potrete acquistare un biglietto di ingresso intero e ricevere un biglietto di

INGRESSO OMAGGIO

MOA CASE & COSE

MC_M

Una iniziativa
MOA Cooperativa

per informazioni: Tel. 06/72.900.200 - Fax 06/72.900.184
www.moacasa.com

Banca di Credito Cooperativo di Roma
Gruppo Cassa Rurale ed Artigiana di Roma