
La programmazione: alcuni esempi

1. Distribuire una funzione su più valori

Nella programmazione funzionale uno dei criteri essenziali è quello dell'applicazione di funzioni su insiemi di dati. Molto spesso i dati sono il risultato stesso dell'applicazione di una funzione, ossia l'output generato da una funzione fornisce i dati per la funzione successiva. Per attuare tali criteri c'è bisogno di un set di istruzioni idoneo per gestire casi anche complessi di applicazione di funzioni non solo a tutto l'insieme dei dati ma anche a parte di essi. *Mathematica* mette a disposizione una serie di funzioni tipiche dei linguaggi funzionali, quali la *Map* (e le sue varianti), la *Apply*, la *Select*, ecc.

Seguono alcuni esempi su *Map*, *MapAll*, *MapAt* e *Apply*.

2. Il comando *Map*

```
In[1]:= ?? Map
Map[f, expr] or f /@ expr applies f to each
element on the first level in expr. Map[f, expr, levelspec]
applies f to parts of expr specified by levelspec. More...

Attributes[Map] = {Protected}

Options[Map] = {Heads -> False}
```

La funzione *Map* applica la funzione passata come primo argomento a tutti gli elementi dell'espressione passata come secondo argomento. Per default la funzione viene applicata solo agli elementi del primo livello, ma si può indicare un parametro opzionale per specificare un diverso livello o più livelli.

```
In[2]:= Map[funzione, {1, 2, 3, simbolo, -2}]
Out[2]= {funzione[1], funzione[2], funzione[3], funzione[simbolo], funzione[-2]}
```

```
In[3]:= Map[funzione, {1, 2, 3, {simbolo1, simbolo2}, -2}]
Out[3]= {funzione[1], funzione[2], funzione[3],
funzione[{simbolo1, simbolo2}], funzione[-2]}
```

Si noti che sebbene il quarto elemento della lista sia una lista, la *funzione* non viene applicata anche ai suoi elementi, come anticipato.

```
In[4]:= Map[funzione, {1, 2, 3, {simbolo1, simbolo2}, -2}, 2]
Out[4]= {funzione[1], funzione[2], funzione[3],
funzione[{funzione[simbolo1], funzione[simbolo2]}], funzione[-2]}
```

Il numero 2 dopo l'espressione indica alla Map di estendere l'applicazione della *funzione* a tutti gli elementi fino al livello 2. Ovviamente in tal caso la *funzione* verrà applicata anche al livello 1. Se si desidera applicarla, invece, solo al livello 2 bisognerà racchiudere il parametro di livello tra le parentesi {}.

```
In[5]:= Map[funzione, {1, 2, 3, {simbolo1, simbolo2}, -2}, {2}]
Out[5]= {1, 2, 3, {funzione[simbolo1], funzione[simbolo2]}, -2}
```

```
In[6]:= Map[Sort, {{1, 2, 5, 21}, {6, 3, 8, -1}, {0, 4, 2, 65}}]
Out[6]= {{1, 2, 5, 21}, {-1, 3, 6, 8}, {0, 2, 4, 65}}
```

3. I comando MapAll e MapAt

Il comando Map permette di specificare il livello di profondità su cui applicare la funzione sull'espressione. Alle volte si ha l'esigenza di applicare una funzione a tutti gli elementi a qualsiasi livello di una espressione. In tali casi si può direttamente utilizzare il comando MapAll

```
In[7]:= ?? MapAll
MapAll[f, expr] or f //@ expr applies f to every subexpression in expr.
More...

Attributes[MapAll] = {Protected}

Options[MapAll] = {Heads -> False}
```

```
In[8]:= exp = x3 + (1 + z)2
General::spell1 : Possible spelling error: new
symbol name "exp" is similar to existing symbol "Exp". More...
Out[8]= x3 + (1 + z)2
```

```
In[9]:= Map[Sin, exp]
Out[9]= Sin[x3] + Sin[(1 + z)2]
```

```
In[10]:= MapAll[Sin, exp]
Out[10]= Sin[Sin[Sin[x]Sin[3]] + Sin[Sin[Sin[1] + Sin[z]]Sin[2]]]
```

Altre volte ancora si desidera specificare un particolare elemento di una espressione su cui applicare la funzione, senza che essa venga applicata a tutti gli elementi sullo stesso livello. La funzione MapAt serve a tale scopo.

```
In[11]:= ?? MapAt
MapAt[f, expr, n] applies f to the element at position n in expr. If
n is negative, the position is counted from the end. MapAt[f,
expr, {i, j, ...}] applies f to the part of expr at position {i,
j, ...}. MapAt[f, expr, {{i1, j1, ...}, {i2, j2, ...}, ...}]
applies f to parts of expr at several positions. More...

Attributes[MapAt] = {Protected}
```

```
In[12]:= MapAt[Sin, exp, {{1, 1}, {2, 1, 2}}]
Out[12]:= Sin[x]3 + (1 + Sin[z])2
```

4. Il comando Apply

Il comando Apply fornisce un utile modo di applicare una funzione contemporaneamente a tutti gli elementi di una espressione. In realtà ciò che avviene con l'Apply è che la Head dell'espressione passata come argomento viene sostituita con la funzione desiderata.

```
In[13]:= ?? Apply
Apply[f, expr] or f @@ expr replaces the head of expr by f. Apply[f, expr,
levelspec] replaces heads in parts of expr specified by levelspec. More...

Attributes[Apply] = {Protected}

Options[Apply] = {Heads -> False}
```

```
In[14]:= Apply[funzione, List[1, 2, 3]]
Out[14]:= funzione[1, 2, 3]
```

```
In[15]:= Apply[Plus, {1, 2, 3}]
Out[15]:= 6
```

```
In[16]:= Apply[And, {True, True, False}]
Out[16]:= False
```

```
In[17]:= Apply[Or, {True, True, False}]
Out[17]:= True
```

```
In[18]:= Apply[And, {True, True, True}]
Out[18]:= True
```

```
In[19]:= Remove["Global`*"]
```

L'Apply lavora per default solo sulla Head dell'espressione passata come argomento, ma non sulle Head dei suoi elementi interni. Se si desidera applicare una funzione a degli elementi particolari si può indicare un parametro aggiuntivo, ossia il livello su cui deve operare la Apply.

```
In[20]:= Apply[Sin, {{f[a], f[b]}, {f[c], f[d]}}, {3}]
Out[20]= {{f[a], f[b]}, {f[c], f[d]}}
```

```
In[21]:= Apply[Sin, {f[a], f[b], f[c], f[d]}, {2}]
Out[21]= {f[a], f[b], f[c], f[d]}
```

```
In[22]:= Apply[Sin, {f[a], f[b], f[c], f[d]}, {1}]
Out[22]= {Sin[a], Sin[b], Sin[c], Sin[d]}
```

```
In[23]:= TreeForm[{f[a], f[b], f[c], f[d]}]
Out[23]/TreeForm=
```

```
List[|      |      |      |
      f[a]  f[b]  f[c]  f[d]]
```

```
In[24]:= Apply[Sin, {{f[a], f[b]}, {f[c], f[d]}}, {2}]
Out[24]= {{Sin[a], Sin[b]}, {Sin[c], Sin[d]}}
```

```
In[25]:= TreeForm[{{f[a], f[b]}, {f[c], f[d]}}]
Out[25]/TreeForm=
```

```
List[|
      List[|      |      |      |]  List[|      |      |      |]
          f[a]  f[b]                f[c]  f[d]]
```

5. La definizione di funzione pura

Spesso quando si utilizzano i comandi quali Map, Apply, ecc. non si ha realmente bisogno di definire preventivamente una funzione da distribuire, in quanto, finita la computazione della Map o Apply che sia, non si ha più bisogno di quella funzione. Un rimedio viene offerto dalla possibilità di creare le Funzioni Pure, ossia funzioni che hanno significato solo nella computazione richiesta, o meglio, che non vengono memorizzate in alcuna variabile. Facciamo un esempio banale per chiarire il concetto.

Si supponga di dover eseguire su una lista di dati la somma dei risultati delle funzioni $f[x]$ e $g[x]$.

Quello che viene spontaneo fare è costruire la funzione

```
In[26]:= Remove["Global`*"]
```

```
In[27]:= h[x_] := f[x] + g[x]
```

```
In[28]:= lista = Table[Random[Integer, {-5, 5}], {6}];
```

```
In[29]:= Map[h, lista]
Out[29]:= {f[-4] + g[-4], f[-1] + g[-1], f[0] + g[0],
           f[-4] + g[-4], f[-5] + g[-5], f[-1] + g[-1]}
```

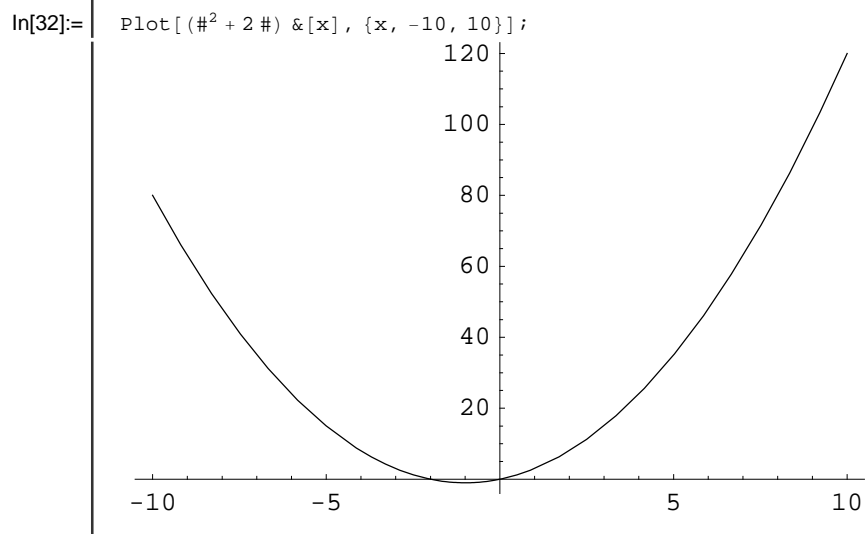
Se la funzione $h[x]$ non viene utilizzata successivamente, si può evitare di definirla, introducendo al suo posto una funzione pura

```
In[30]:= Map[Function[{x}, f[x] + g[x]], lista]
Out[30]:= {f[-4] + g[-4], f[-1] + g[-1], f[0] + g[0],
           f[-4] + g[-4], f[-5] + g[-5], f[-1] + g[-1]}
```

In forma contratta

```
In[31]:= Map[(f[#] + g[#]) &, lista]
Out[31]:= {f[-4] + g[-4], f[-1] + g[-1], f[0] + g[0],
           f[-4] + g[-4], f[-5] + g[-5], f[-1] + g[-1]}
```

In sostanza il comando `Function` definisce una funzione locale, che non viene memorizzata in maniera definitiva, ma solamente usata all'interno della `Map`. Altro esempio: vogliamo graficare la funzione x^2+2x , senza definirla.



6. L'importanza di alcuni attributi, come ad esempio `Listable`

Diverse funzioni, come ad esempio quelle trigonometriche, hanno l'attributo `Listable`. Le funzioni che hanno questa caratteristica possono lavorare sia su espressioni atomiche sia su espressioni di tipo lista; in pratica quando vengono utilizzate su una lista quello che accade è che esse estendono l'applicazione della funzione stessa su tutti gli elementi della lista e non sulla lista nella sua interezza. D'altronde l'unica cosa che avrebbe significato nella richiesta `Sin[{ $\frac{\pi}{2}$, $\frac{3}{2}\pi$, 2π }]` è quella di cercare il valore del seno nei singoli punti della lista. Tecnicamente si può dire che, nell'esempio appena riportato, la funzione `Sin` si commuta con la head `List` e quindi il risultato è un'espressione del tipo `List[Sin[0], Sin[$\frac{\pi}{2}$], Sin[π], Sin[$\frac{3}{2}\pi$], Sin[2π]].`

Come si deduce dagli esempi riportati, molte operazioni aritmetiche hanno l'attributo `Listable`, quindi significa che è possibile eseguire operazioni aritmetiche direttamente sulle liste, come negli esempi che seguono:

```
In[33]:= Remove["Global`*"] (* rimuove eventuali variabili definite precedentemente *)
```

```
In[34]:= basi = {a, b, c}
Out[34]= {a, b, c}
```

```
In[35]:= esponenti = {d, e, f}
Out[35]= {d, e, f}
```

```
In[36]:= basiesponenti
Out[36]= {ad, be, cf}
```

```
In[37]:= basi * esponenti
Out[37]= {a d, b e, c f}
```

```
In[38]:= basi + esponenti
Out[38]= {a + d, b + e, c + f}
```

```
In[39]:= espressioni = (basi - x)esponenti + 1
Out[39]= {1 + (a - x)d, 1 + (b - x)e, 1 + (c - x)f}
```

```
In[40]:= basi = {1, 2, 3}
Out[40]= {1, 2, 3}
```

```
In[41]:= esponenti = {4, 5, 6}
Out[41]= {4, 5, 6}
```

```
In[42]:= espressioni = (basi - x)esponenti + 1
Out[42]= {1 + (1 - x)4, 1 + (2 - x)5, 1 + (3 - x)6}
```

```
In[43]:= Expand[espressioni] // TableForm
Out[43]//TableForm=
  2 - 4 x + 6 x2 - 4 x3 + x4
  33 - 80 x + 80 x2 - 40 x3 + 10 x4 - x5
  730 - 1458 x + 1215 x2 - 540 x3 + 135 x4 - 18 x5 + x6
```

```
In[44]:= D[espressioni, x]
Out[44]= {-4 (1 - x)3, -5 (2 - x)4, -6 (3 - x)5}
```

```
In[45]:= D[Expand[espressioni], x]
Out[45]= {-4 + 12 x - 12 x2 + 4 x3, -80 + 160 x - 120 x2 + 40 x3 - 5 x4,
-1458 + 2430 x - 1620 x2 + 540 x3 - 90 x4 + 6 x5}
```

```
In[46]:= Expand[D[espressioni, x]]
Out[46]= {-4 + 12 x - 12 x2 + 4 x3, -80 + 160 x - 120 x2 + 40 x3 - 5 x4,
-1458 + 2430 x - 1620 x2 + 540 x3 - 90 x4 + 6 x5}
```

Se si utilizzano tutte funzioni e/o comandi con l'attributo Listable, si può comporre un'unica linea con tutte le operazioni richieste. Per esempio se si desidera conoscere il valore nel punto $x = 0$ delle derivate delle espressioni sopra calcolate si può scrivere:

```
In[47]:= ReplaceAll[D[(basi - x)esponenti + 1, x], x -> 0]
Out[47]= {-4, -80, -1458}
```

La programmazione: alcuni esempi

7. Esempio FactorInteger (stile funzionale)

Si vuole creare una funzione che a partire dalla lista dei fattori primi di un numero ci restituisca il numero stesso. La lista dei fattori primi viene data ad esempio dalla FactorInteger

```
In[48]:= FactorInteger[283500]
Out[48]= {{2, 2}, {3, 4}, {5, 3}, {7, 1}}
```

Ora a partire da tale lista bisogna ricostruire il numero 283500.

Si crea prima una funzione pura che eleva il primo argomento di una lista al secondo argomento

```
In[49]:= f = Function[{x}, Apply[Power, x]]
Out[49]= Function[{x}, Power@@x]
```

o equivalentemente utilizzando la forma contratta della funzione pura

```
In[50]:= g = (Apply[Power, #] &)
Out[50]= Power@@#1 &
```

```
In[51]:= {f[{a, b}], g[{a, b}]}
Out[51]= {ab, ab}
```

poi si usa la Map per lavorare sulla lista dei fattori primi

```
In[52]:= Map[f, FactorInteger[283500]]
Out[52]= {4, 81, 125, 7}
```

infine si utilizza l'Apply per ricomporre il tutto con la moltiplicazione

```
In[53]:= Apply[Times, %]
Out[53]= 283500
```

Nel paradigma funzionale quando possibile si riscrivono le operazioni in un'unica sequenza di chiamate nidificate; pertanto si può definire la funzione come segue

```
In[54]:= ExpandFactorization[factors_] := Apply[Times, Map[Apply[Power, #] &, factors]]
```

```
In[55]:= ExpandFactorization[FactorInteger[283500]]
Out[55]= 283500
```

```
In[56]:= fattori = Table[Random[Integer, {2, 10}], {30000}, {2}];
```

```
In[57]:= Timing[ExpandFactorization[fattori]]
```

Per brevità, il risultato viene ommesso in stampa

Un approccio tipico del paradigma procedurale suggerirebbe la seguente implementazione

```
In[58]:= ExpandFactorization2[factors_] :=
(
  lista = Table[factors[[i, 1]]^factors[[i, 2]], {i, Length[factors]};
  risultato = 1;
  Table[risultato = risultato * lista[[i]], {i, Length[lista]};
  risultato)

```

```
In[59]:= fattori = Table[Random[Integer, {2, 10}], {15000}, {2}];
```

```
In[60]:= Timing[ExpandFactorization2[fattori];]
Out[60]= {0.391 Second, Null}
```

```
In[61]:= Timing[ExpandFactorization[fattori];]
Out[61]= {0.14 Second, Null}
```

```
In[62]:= %%[[1]] / %%[[1]]
Out[62]= 2.79286
```

8. Esempio Distanza di due punti (pattern matching)

Distanza tra due punti nel piano cartesiano $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

```
In[63]:= A = {x1, y1}
Out[63]= {x1, y1}
```

```
In[64]:= B = {x2, y2}
Out[64]= {x2, y2}
```

sfruttando l'attributo Listable della funzione Power e della funzione Plus, si può scrivere direttamente

```
In[65]:= (A - B)^2
Out[65]= {(x1 - x2)^2, (y1 - y2)^2}
```

applicando poi la somma alle singole componenti ottenute e successivamente la radice quadrata, si ottiene la formula della distanza

```
In[66]:= Sqrt[Apply[Plus, (A - B)^2]]
Out[66]= Sqrt[(x1 - x2)^2 + (y1 - y2)^2]
```

Si noti la comodità della perfetta integrazione tra il front end ed il kernel; in questo esempio si è utilizzato il simbolo di radice quadrata direttamente su una funzione del kernel. Il front end si preoccupa di convertire il simbolo SqrtBox (che a video mostra il simbolo $\sqrt{\quad}$) con la corrispondente funzione Sqrt.

Ora si crea la funzione Distanza

```
In[67]:= Distanza[A_, B_] := Sqrt[Apply[Plus, (A - B)^2]]
```

```
In[68]:= Distanza[A, B]
Out[68]= Sqrt[(x1 - x2)^2 + (y1 - y2)^2]
```

```
In[69]:= Distanza[{1, 2}, {4, 8}]
Out[69]= 3 Sqrt[5]
```

```
In[70]:= Distanza[3]
Out[70]= Distanza[3]
```

```
In[71]:= Distanza[{3, 2}]
Out[71]= Distanza[{3, 2}]
```

```
In[72]:= Distanza[3, 2]
Out[72]= 1
```

```
In[73]:= Distanza[3, pippo]
Out[73]=  $\sqrt{5 - \text{pippo}}$ 
```

```
In[74]:= Distanza[3, {3, 2}]
Out[74]= 1
```

```
In[75]:= Distanza[{3, 0}, {3, 2}]
Out[75]= 2
```

```
In[76]:= Distanza[{0, 3}, {3, 2}]
Out[76]=  $\sqrt{10}$ 
```

Dagli ultimi esempi si deduce che la funzione `Distanza` non risponde correttamente a tutti gli input. Per ovviare al problema basta inserire un *filtro*, applicato all'input della funzione `Distanza`, per mandare in esecuzione il corpo della funzione solo se l'argomento passato è del tipo desiderato, ossia in questo caso solo se si passano in input due coppie di numeri.

```
In[77]:= Remove[Distanza]
```

```
In[78]:= ?? Distanza
Information::notfound: Symbol Distanza not found. More...
```

```
In[79]:= Distanza[A_List, B_List] :=  $\sqrt{\text{Apply}[\text{Plus}, (A - B)^2]}$ 
```

```
In[80]:= Distanza[3]
Out[80]= Distanza[3]
```

```
In[81]:= Distanza[{3, 4}, {0, 3}]
Out[81]=  $\sqrt{10}$ 
```

```
In[82]:= Distanza[{3, 4}]
Out[82]= Distanza[{3, 4}]
```

```
In[83]:= Distanza[3, {3, 2}]
Out[83]= Distanza[3, {3, 2}]
```

```
In[84]:= Distanza[{pippo, pluto}, {3, 2}]
Out[84]=  $\sqrt{(-3 + \text{pippo})^2 + (-2 + \text{pluto})^2}$ 
```

come si osserva dai risultati, il problema è stato in parte risolto, ma si desidera fare ancora meglio. Si vuole aggiungere un commento nel caso di input sbagliato e si vuole evitare che l'input sia non numerico. Si introduce così il concetto di pattern in maniera più evidente.

Si consideri innanzitutto quali sono i modi possibili di specificare i due punti su cui calcolare la Distanza. Ad esempio si potrebbero ammettere i due seguenti input:

```
Distanza[{x1, y1}, {x2, y2}]
Distanza[{{x1, y1}, {x2, y2}}]
```

inoltre, si deve prevedere che l'utente potrebbe digitare erroneamente qualcosa di diverso. Bisogna coprire tutti i possibili casi con una definizione completa della funzione Distanza.

La soluzione che tiene conto di tutte queste indicazioni potrebbe essere la seguente

```
In[85]:= Remove[Distanza]
```

```
In[86]:= ?? Distanza
Information::notfound : Symbol Distanza not found. More...
```

```
In[87]:= Distanza[A : {_?NumericQ, _?NumericQ}, B : {_?NumericQ, _?NumericQ}] :=
 $\sqrt{\text{Apply}[\text{Plus}, (A - B)^2]}$ 
```

```
In[88]:= Distanza[punti : {{_, _}, {_, _}}] := Distanza[punti[[1]], punti[[2]]]
```

```
In[89]:= Distanza[___] := "Input errato"
```

```
In[90]:= ?? Distanza
Global`Distanza
```

```
Distanza[A : {_?NumericQ, _?NumericQ}, B : {_?NumericQ, _?NumericQ}] :=
 $\sqrt{\text{Plus}@@(A - B)^2}$ 
```

```
Distanza[punti : {{_, _}, {_, _}}] := Distanza[punti[[1]], punti[[2]]]
```

```
Distanza[___] := Input errato
```

```
In[91]:= Distanza[3]
Out[91]= Input errato
```

```
In[92]:= Distanza[{3, 4}]
Out[92]= Input errato
```

```
In[93]:= Distanza[{3, 4}, {2, 3}]
Out[93]=  $\sqrt{2}$ 
```

```
In[94]:= Distanza[{{3, 4}, {1, 2}}]
Out[94]=  $2\sqrt{2}$ 
```

```
In[95]:= Distanza[{2, s}, {1, 2}]
Out[95]= Input errato
```

```
In[96]:= Distanza[{{3\sqrt{2}, 9}, {0, \frac{1}{3}}}]
Out[96]=  $\frac{\sqrt{838}}{3}$ 
```

La programmazione: alcuni esempi

9. Comparazione dei tempi di calcolo

Un problema sulla manipolazione di liste

Data una lista L di n elementi del tipo a o b (esempio $L = \{a, b, b, a, b, a, a, a, b, b, a, a\}$) bisogna costruire una lista L' secondo il criterio:

$$L'_k = \begin{cases} 1 & \text{se } L_k = L_{k+1} \\ 2 & \text{se } L_k \neq L_{k+1} \end{cases} \quad \text{con } k = 1, 2, \dots, n-1$$

$$L'_n = \begin{cases} 1 & \text{se } L_n = L_1 \\ 2 & \text{se } L_n \neq L_1 \end{cases}$$

nell'esempio:

```
L = {a, b, b, a, b, a, a, a, b, b, a, a}
L' = {2, 1, 2, 2, 2, 1, 1, 2, 1, 2, 1, 1}.
```

Si crea una lista L^F con una partizione di L in coppie di elementi contigui. In seguito basta operare dei controlli su ciascuna coppia e restituire 1 o 2 a seconda del caso indicato nel criterio. Bisogna considerare la lista L come ciclica, ossia l'ultimo elemento deve essere confrontato con il primo. Per fare questo si può utilizzare la funzione `Partition`, che prevede il caso di liste cicliche.

```
In[97]:= ?Partition
Partition[list, n] partitions list into non-overlapping sublists of length
n. Partition[list, n, d] generates sublists with offset d. Partition[
list, {n1, n2, ...}] partitions a nested list into blocks of size
n1 × n2 × ... . Partition[list, {n1, n2, ...}, {d1, d2, ...}] uses
offset di at level i in list. Partition[list, n, d, {kL, kR}]
specifies that the first element of list should appear at position
kL in the first sublist, and the last element of list should appear
at or after position kR in the last sublist. If additional elements
are needed, Partition fills them in by treating list as cyclic.
Partition[list, n, d, {kL, kR}, x] pads if necessary by repeating the
element x. Partition[list, n, d, {kL, kR}, {x1, x2, ...}] pads if
necessary by cyclically repeating the elements xi. Partition[list,
n, d, {kL, kR}, {}] uses no padding, and so can yield sublists of
different lengths. Partition[list, nlist, dlist, {klistL, klistR},
padlist] specifies alignments and padding in a nested list. More...
```

In particolare, si osserva dall'help l'indicazione:

`Partition[list, n, d, {kL, kR}]` specifies that the first element of list should appear at position kL in the first sublist, and the last element of list should appear at or after position kR in the last sublist. If additional elements are needed, Partition fills them in by treating list as cyclic.

E' esattamente ciò che occorre. Infatti, basta specificare come primo parametro della Partition la dimensione delle sottoliste (nel caso in esame vale 2), poi il valore dell'offset, ossia la sovrapposizione che deve essere considerata per creare la sottoliste di lunghezza due (nel caso in esame vale 1). Infine, bisogna specificare che il primo elemento della lista L deve comparire al primo posto della prima sottolista, mentre l'ultimo elemento della lista L deve comparire al primo posto dell'ultima sottolista. In tal modo la Partition automaticamente prende come secondo elemento dell'ultima sottolista il primo elemento della lista L.

```
In[98]:= Remove["Global`*"]
```

```
In[99]:= L = {a, b, b, a, b, a, a, a, b, b, a, a}
Out[99]:= {a, b, b, a, b, a, a, a, b, b, a, a}
```

```
In[100]:= L1 = Partition[L, 2, 1, {1, 1}]
Out[100]:= {{a, b}, {b, b}, {b, a}, {a, b}, {b, a},
{a, a}, {a, a}, {a, b}, {b, b}, {b, a}, {a, a}, {a, a}}
```

A questo punto basta implementare una funzione che per ciascuna coppia della partizione L1 restituisca 1 o 2 a seconda che gli elementi siano uguali o differenti.

Vi sono molti modi di scrivere questa funzione, basta pensare ad un ciclo di controlli sulle coppie con un If interno.

Di seguito si riportano alcune soluzioni, tutte formalmente corrette ma non con le stesse prestazioni.

```
In[101]:= PrimaSoluzione[lista_List] :=
  Table[ If[ lista[[i, 1]] === lista[[i, 2]], 1, 2], {i, Length[lista]}];

SecondaSoluzione[lista_List] := Map[ If#[[1]] === #[[2]], 1, 2] &, lista];

TerzaSoluzione[lista_List] := Replace[lista, {{u_, u_} -> 1, {u_, v_} -> 2}, 1];

FirstTestQ[{a_, b_}] := 1 /; SameQ[a, b];
FirstTestQ[{a_, b_}] := 2;
QuartaSoluzione[lista_List] := Map[FirstTestQ, lista];

SecondTestQ[a_, b_] := 1 /; SameQ[a, b];
SecondTestQ[a_, b_] := 2;
QuintaSoluzione[lista_List] := Apply[SecondTestQ, lista, 2];
```

Si verifica che tutte le funzioni restituiscono lo stesso risultato

```
In[110]:= {
  PrimaSoluzione[L1], SecondaSoluzione[L1], TerzaSoluzione[L1],
  QuartaSoluzione[L1], QuintaSoluzione[L1]} // TableForm
```

Out[110]//TableForm=

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 |

Ora, a parte alcune considerazioni sullo stile di programmazione utilizzato, quello che interessa è il confronto sui tempi di calcolo.

La riga di codice seguente genera una lista di 50.000 elementi del tipo a o b in sequenza casuale.

```
In[111]:= L = Table[{a, b}][[Random[Integer, {1, 2}]]], {50000}];
```

Si costruisce la partizione L1 comune a tutte le funzioni implementate:

```
In[112]:= L1 = Partition[L, 2, 1, {1, 1}];
```

Si calcolano i tempi per ciascuna funzione

```
In[113]:= {
  Timing[PrimaSoluzione[L1]]; ,
  Timing[SecondaSoluzione[L1]]; , Timing[TerzaSoluzione[L1]]; ,
  Timing[QuartaSoluzione[L1]]; , Timing[QuintaSoluzione[L1]];
} // TableForm
```

Out[113]//TableForm=

| | |
|--------------|------|
| 0.297 Second | Null |
| 0.156 Second | Null |
| 0.032 Second | Null |
| 0.109 Second | Null |
| 0.109 Second | Null |

Questo esempio mostra come anche per semplici problemi lo stile di programmazione incide sull'efficienza delle funzioni.

Il crivello di Eratostene: il calcolo dei numeri primi

```
In[114]:= Eratostene1[n_] := Rest[Complement[Range[n],
      Flatten[Table[Range[2 i, n, i], {i, 2, Floor[Sqrt[n]}]]]];

Eratostene2[n_] := Rest[Complement[Range[n],
      Flatten[Map[Range[##, n, #] &, Eratostene2[Floor[Sqrt[n]]]]]];
Eratostene2[2] := {2};
Eratostene2[3] := {2, 3};

Eratostene3[n_] := Rest[Complement[Range[n], Flatten[
      Map[Range[2 #, n, #] &, Select[Range[2, Floor[Sqrt[n]]], PrimeQ]]]];

Eratostene4[start_Integer?Positive, stop_Integer?Positive] :=
Module[{},
  primi = FoldList[
    Function[{lista, numero}, Cases[lista, _? (Mod[#, numero] != 0 &)],
      Range[start, stop], Range[2, IntegerPart[Sqrt[stop]]]];
  numeprimi = Last[primi];
];
Eratostene4[stop_Integer?Positive] := Eratostene4[2, stop];
```

```
In[121]:= {
  Timing[Eratostene1[500000];],
  Timing[Eratostene2[500000];],
  Timing[Eratostene3[500000];],
  Timing[Eratostene4[500000];]
} // TableForm
```

```
Out[121]//TableForm=
1.422 Second      Null
0.469 Second      Null
0.547 Second      Null
82.062 Second     Null
```