

# Struttura di Mathematica

## Introduzione

*Mathematica* è costituito da due componenti: il Kernel e il Front End. Il Kernel (nucleo) è il motore di calcolo del programma, mentre il Front End è l'interfaccia utente, naturalmente grafica. Durante una sessione di lavoro l'utente interagisce con il Kernel attraverso il Front End. A loro volta, Kernel e Front End scambiano dati mediante un protocollo di comunicazione denominato MathLink. *Mathematica* dà la possibilità di chiudere il Kernel separatamente, in modo da riprendere un nuovo ciclo di calcolo durante la medesima sessione.

Per default i file di *Mathematica* vengono salvati come Notebook aventi estensione .nb. L'interazione tra il Kernel e il Front End avviene tramite le cosiddette "celle" di input. Al termine della computazione, il Kernel restituisce il risultato attraverso celle di output.

Per quanto detto, il Kernel può essere disattivato durante la sessione di lavoro direttamente dalla Toolbar (Evaluation->Quit Kernel). Si osservi che la chiusura del Kernel causa la perdita di tutti i dati presenti nel notebook. Al contrario, se si vuole interrompere una sola operazione, si può interrompere il processo, "abortendolo". Per fare ciò, basta andare sul pulsante "Evaluation" sulla Toolbar e selezionare la voce "Abort Evaluation". Si suggerisce la chiusura del Kernel quando c'è il rischio di un crash di sistema dovuti a cicli infiniti.

Per quanto riguarda l'aspetto grafico del Front End, osserviamo che esso può

essere personalizzato attraverso i “fogli di stile”. Il percorso è: Format->Style-Sheet. Qui è possibile scegliere tra vari Templates.

# Sintassi

*Mathematica* è case-sensitive, cioè distingue le maiuscole dalle minuscole. Ad esempio, i nomi delle funzioni built-in sono maiuscoli, come pure i comandi. In generale, le istruzioni hanno la seguente sintassi: **Command**[**expr1**, **expr2**, ..., **exprn**]

Qui **Command** è un comando generico, mentre **expr1**,... sono espressioni che verranno valutate. È buona norma scrivere le funzioni definite dall'utente con lettere minuscole, in modo da distinguerle dalle funzioni built-in. Per quanto riguarda le

funzioni, gli argomenti sono racchiuse tra parentesi quadre, anziché tonde. Per quanto detto, un notebook è un set di celle di input e output. Per non mostrare l'output si utilizza il terminatore ;

Se in una data fase di sessione caratterizzata dalla k-esima cella di output, si desidera avere in output il contenuto della cella n-esima con  $n < k$ , dobbiamo utilizzare il comando **Out**[**n**], che restituisce l'n-esima cella. Alternativamente si può utilizzare la forma **%%...%**

Esempio:

```
a = 1.2;
```

```
b = 0.8;
```

```
%%
```

```
1.2
```

La sintassi per le righe di commento è: (**\* commento \***)

# Operatori aritmetici

Le operazioni aritmetiche ordinarie sono  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $^$ . Il simbolo  $*$  utilizzato nella moltiplicazione può essere omissso a patto di lasciare uno spazio. Ad esempio:

**ab**

ab

è equivalente a

**a \* b**

a b

In alcuni casi particolari, lo spazio non è necessario. Ad esempio:

**4 Cosh [x]**

4 Cosh [x]

è equivalente a:

**4 \* Cosh [x]**

4 Cosh [x]

però **xy** ≠ **x\*y**

Quindi è preferibile lasciare uno spazio o utilizzare il simbolo  $*$ .

# Costanti built-in

Le principali costanti built-in sono:

**Infinity**

$\infty$

**(\*digitando escape inf escape\*)**

$\infty$

$\infty$

**Pi**

$\pi$

**(\*digitando escape pi escape\*)**

$\pi$

$\pi$

Base dei logaritmi naturali:

**E**

$e$

Unità immaginaria:

**I**

$i$

Per passare dal valore simbolico al valore numerico è necessario utilizzare la funzione **N[ ]**. Ad esempio

**N[ $\pi$ ]**

3.14159

Può anche essere utilizzata la forma abbreviata

**$\pi$  // N**

3.14159

## Operatori Booleani

Gli operatori booleani (connettivi logici) sono: **&&** (**And**), **||** (**Or**), **!** (**Not**), **Xor**

# Funzioni Built-in

Le principali funzioni built-in sono:

Funzione logaritmo in base  $e$

**Log [x] ;**

Funzione  $\log_a x$

**Log [a, x] ;**

La funzione valore assoluto

**Abs [x] ;**

Le usuali funzioni trigonometriche:

**Sin [x] ;**

**Cos [x] ;**

**Tan [x] ;**

**ArcCos [x] ;**

**ArcSin [x] ;**

**ArcTan [x] ;**

# Funzioni definite dall'utente

## Assegnazione immediata

Esistono diversi modi per definire una funzione. L'approccio più immediato ha il seguente costrutto: `f[x_]=expr[x]`

Analizziamo questo codice. L'“underscore” (cioè il trattino “\_”) è il cosiddetto pattern (“modello”). Il pattern comunica al Kernel che la variabile  $x$  rappresenta “qualunque cosa”. Il secondo membro è invece l'espressione analitica della funzione

$f$ .

Ad esempio, se scriviamo:

```
f[x] = x^2
x^2
```

e proviamo a calcolare il valore assunto dalla funzione  $f(x) = x^2$  nel punto  $x = 2$ , *Mathematica* restituisce  $f(2)$ .

```
f[2]
f[2]
```

Invece, se usiamo la sintassi con l'underscore:

```
f[x_] = x^2
x^2

f[2]
4
```

cioè il risultato corretto. Il costrutto: `f[x_]=expr[x]` definisce ciò che si chiama *assegnazione immediata* della funzione  $f(x)$ .

## Assegnazione ritardata

Nell'assegnazione ritardata la funzione viene valutata solo quando viene richiamata dall'utente. La sintassi è: `f[x_] := expr[x]`

In output non viene restituito nulla. Per visualizzare l'espressione della funzione, dobbiamo richiamare `f[x]`; l'output sarà `expr[x]`

Per comprendere la differenza tra i due tipi di assegnazione - immediata e ritardata - consideriamo la funzione:

```
f[x_] := Simplify[x^2 + 1 - 3 + x + 2 * x]
```

essendo `Simplify[]` un comando built-in utilizzato per semplificare algebricamente un'espressione. Utilizziamo il comando `?` per chiedere a *Mathematica* cos'è la funzione `f[x]` appena definita. Abbiamo :

```
? f
```

```
Global`f
```

```
f[x_] := Simplify[x^2 + 1 - 3 + x + 2 x]
```

Adesso definiamo:

```
g[x_] = Simplify[x^2 + 1 - 3 + x + 2 x]
- 2 + 3 x + x^2
```

che restituisce il risultato dell'applicazione di `Simplify`

Un modo veloce per visualizzare il valore assunto da una funzione  $f$  in un punto, consiste nell'usare il comando `//f`. Consideriamo ad esempio, la funzione

```
f[x_] := Cosh[x^2 + 1]
```

Se scriviamo:

```
2 // f
```

```
Cosh[5]
```

Volendo il valore numerico:

```
N[f[2]]  
74.2099
```

Ma `//Command` agisce anche sulle funzioni built-in, quindi:

```
f[2] // N  
74.2099
```

Evidentemente lo stesso risultato può essere raggiunto più velocemente scrivendo:

```
2 // f // N  
74.2099
```

**Osservazione.** Durante una sessione di lavoro con Mathematica, il Kernel enumera progressivamente i comandi inseriti, nonché le funzioni definite dall'utente. Gli output corrispondenti vengono poi caricati dal Kernel, anche se una cella di

output viene cancellata manualmente. Da ciò segue che se definiamo una funzione `f`, il Kernel non perde memoria di tale oggetto, quindi volendo definire una nuova funzione, sarà necessario un nuovo simbolo. Per evitare conflitti tra simboli o una

proliferazione degli stessi è conveniente rimuovere dal Kernel tutte quelle informazioni divenute inutili a quel punto della sessione. Una funzione built-in adatta allo scopo è `Clear`, la cui sintassi è:

```
Clear[symbol1, symbol2, ...]
```

Dove i vari `symbol` sono oggetti definiti dall'utente. Nell'esempio precedente, volendo definire una nuova funzione con lo stesso simbolo, scriviamo:

```
Clear[f]
```

Un altro comando utilizzato per rimuovere informazione è **Remove**, che è più potente di **Clear**, in quanto rimuove completamente un simbolo da un'intera sessione.

Ad esempio, se con l'input precedente scriviamo:

```
? f
```

```
Global`f
```

Adesso utilizziamo **Remove**:

```
Remove[f]
```

```
? f
```

```
Information::notfound: Symbol f not found. >>
```

## Definizione di funzioni composte

Vediamo ora come definire una funzione composta, cioè una funzione del tipo  $f(x(t))$ . Per fissare le idee consideriamo la funzione:

$f(x) = x^5 + 6x^4 - 2x + \ln(x^2 + 1)$ . Supponiamo che  $x$  sia una

funzione della variabile reale  $t$ . Precisamente:  $x(t) = \sin(t)$ ,

per cui:  $f(t) = \sin^5(t) + 6\sin^4(t) - 2\sin(t) + \ln(\sin^2(t) + 1)$ .

Per definire tale funzione con *Mathematica*, definiamo prima la  $f(x)$  tramite assegnazione

ritardata:

```
f[x_] := x^5 + 6 * x^4 - 2 * x + Log[x^2 + 1]
```

dopodichè utilizziamo il comando **ReplaceAll**

```
g[t_] := ReplaceAll[f[x], x -> Sin[t]]
```

```
g[t]
```

```
Log[1 + Sin[t]^2] - 2 Sin[t] + 6 Sin[t]^4 + Sin[t]^5
```

Alternativamente si può utilizzare l'operatore di sostituzione /.

```
Clear[g]
g[t_] := f[x] /. x -> Sin[t]
g[t]
Log[1 + Sin[t]^2] - 2 Sin[t] + 6 Sin[t]^4 + Sin[t]^5
```

Il procedimento si generalizza al caso di funzioni di più variabili. Ad esempio:

$$f(x) = \sqrt{x^2 - y^2} \text{ con } x(t) = \cos(t), y(t) = \sin(t)$$

```
Clear[f, g]
f[x_, y_] := Sqrt[x^2 - y^2]
g[t_] := f[x, y] /. {x -> Cos[t], y -> Sin[t]}
g[t]
Sqrt[Cos[t]^2 - Sin[t]^2]
```

## Assegnazione condizionata

Vediamo ora come si definisce una *piecewise function*, cioè una funzione di una o più variabili che ha diverse espressioni in differenti sottoinsiemi dell'insieme di definizione. Ad esempio, nel caso di una variabile:

$$f(x) = \sin(x) \text{ se } x \in (-\infty, 0], f(x) = \sin\left(\frac{1}{x}\right) \text{ se } x \in (0, +\infty)$$

Possiamo applicare l'operatore /; la cui sintassi è **f[x\_] := expr /; condizione**

```
Clear[f, g]
f[x_] := Sin[x] /; x <= 0
f[x_] := Sin[1/x] /; x > 0
```

Nel caso di sottinsiemi più complessi si utilizzano gli operatori **And**, **Or**. È anche possibile utilizzare il comando **Piecewise[]**

```
Clear[f]

f[x_] := Piecewise[{{Sin[x], x ≤ 0}, {Sin[1/x], x > 0}}]

Plot[f[x], {x, -4, 4}]
```

## Caching (memorizzazione dei valori calcolati)

La tecnica caching permette di ridurre il carico computazionale nei programmi ricorsivi. Prima di illustrare tale tecnica introduciamo il comando **Table** e la funzione built-in **Prime[]**.

**Table[]** è un generatore di liste, la cui sintassi è:

```
Table[expr, {k, kmin, kmax, kstep}]
```

In questo costrutto **k** è un iteratore che prende valori nell'intervallo discreto **kmin**, **kmax**, mentre **kstep** individua il passo. Per default è **kmin=kstep=1**. Cioè, se scriviamo: **Table[expr, {k, kmax}]** *Mathematica* restituisce i valori assunti da **expr** per **k** variabile da **1** a **kmax** con passo pari a **1**.

La funzione **Prime[n]** restituisce l'*n*-esimo numero primo. Ciò premesso, definiamo la seguente funzione ricorsiva:

```
Clear[f]

f[n_] := f[n - 1] * Prime[n]

?? f
```

```
Global`f
```

```
f[n_] := f[n - 1] Prime[n]
```

Assegnamo il valore iniziale di **f[n]**

```
f[1] = 2;
```

Qui abbiamo fatto uso del terminatore `;` il quale impedisce la visualizzazione dell'output. Possiamo determinare i valori assunti da `f[n]` per assegnati valori discreti della variabile in un certo range `nmin`, `nmax` utilizzando il comando `Table[]`. Ad esempio:

```
Table[f[n], {n, 2, 20}]
```

```
{6, 30, 210, 2310, 30 030, 510 510, 9 699 690, 223 092 870,
 6 469 693 230, 200 560 490 130, 7 420 738 134 810,
 304 250 263 527 210, 13 082 761 331 670 030,
 614 889 782 588 491 410, 32 589 158 477 190 044 730,
 1 922 760 350 154 212 639 070, 117 288 381 359 406 970 983 270,
 7 858 321 551 080 267 055 879 090,
 557 940 830 126 698 960 967 415 390 }
```

```
?? f
```

```
Global`f
```

```
f[1] = 2
```

```
f[n_] := f[n - 1] Prime[n]
```

Abbiamo così generato una successione per ricorsione. La tecnica *caching* in questo caso consiste nel combinare le definizioni di funzione utilizzate da *Mathematica*.

Precisamente:

```
Clear[f]
```

```
f[n_] := f[n] = f[n - 1] * Prime[n]
```

```
Table[f[n], {n, 2, 20}]
```

```
{6, 30, 210, 2310, 30030, 510510, 9699690, 223092870,
 6469693230, 200560490130, 7420738134810,
 304250263527210, 13082761331670030,
 614889782588491410, 32589158477190044730,
 1922760350154212639070, 117288381359406970983270,
 7858321551080267055879090,
 557940830126698960967415390 }
```

In questo modo abbiamo che  $f[n-1]$  restituisce il valore richiesto, e al tempo stesso viene salvato dal Kernel, grazie all'assegnazione ritardata `:=`

```
?? f
```

```
Global`f
```

$$f[1] = 2$$

$$f[2] = 6$$

$$f[3] = 30$$

$$f[4] = 210$$

$$f[5] = 2310$$

$$f[6] = 30\,030$$

$$f[7] = 510\,510$$

$$f[8] = 9\,699\,690$$

$$f[9] = 223\,092\,870$$

$$f[10] = 6\,469\,693\,230$$

$$f[11] = 200\,560\,490\,130$$

$$f[12] = 7\,420\,738\,134\,810$$

$$f[13] = 304\,250\,263\,527\,210$$

$$f[14] = 13\,082\,761\,331\,670\,030$$

$$f[15] = 614\,889\,782\,588\,491\,410$$

$$f[16] = 32\,589\,158\,477\,190\,044\,730$$

$$f[17] = 1\,922\,760\,350\,154\,212\,639\,070$$

$$f[18] = 117\,288\,381\,359\,406\,970\,983\,270$$

$$f[19] = 7\,858\,321\,551\,080\,267\,055\,879\,090$$

$$f[20] = 557\,940\,830\,126\,698\,960\,967\,415\,390$$

$$f[n_] := f[n] = f[n - 1] \text{Prime}[n]$$

# Strutture dati

## Formati numerici

Quando si introduce un numero bisogna informare *Mathematica* sul tipo di numero introdotto cioè se si tratta di un intero (**Integer**), di un razionale (**Rational**), di un reale (**Real**) o di un numero complesso (**Complex**). Ad esempio scriviamo:

**2.0**

2.

O la forma abbreviata:

**2.**

2.

In tal modo abbiamo dato in input il numero reale 2. Se invece avessimo scritto:

**2**

2

avremmo dato in input il numero intero 2.

Un razionale è il rapporto tra due interi. Esempio:

**- 220 / 12**

$-\frac{55}{3}$

Il numero complesso  $6 + 5i$  si scrive:

```
6. + 5. I
```

```
6. + 5. i
```

Una funzione built-in utilizzata per esplicitare il formato numerico è **FullForm**. Esempio:

```
FullForm[8 + 3 I]
```

```
Complex[8, 3]
```

```
FullForm[12 / 9]
```

```
Rational[4, 3]
```

Nota: per scrivere rapidamente un comando è utile il completamento automatico tramite i tasti di scelta rapida **Ctrl + K**.

È possibile controllare il formato dei numeri attraverso la funzione **Head[]**. Precisamente, **Head[x]** restituisce il tipo di numero a cui appartiene **x**.

```
Head[2.1]
```

```
Real
```

```
Head[4]
```

```
Integer
```

```
Head[2 + 3 I]
```

```
Complex
```

Se lasciamo inespresa la variabile **x**, **Head[x]** restituisce **Symbol**.

```
Head[x]
```

```
Symbol
```

La funzione **NumberQ[]** restituisce **True** se il suo argomento è un qualunque formato numerico. Restituisce la costante logica **False** in tutti gli altri casi.

```
NumberQ[2.101]
```

```
True
```

```
NumberQ[11 / 20]
```

```
True
```

```
NumberQ[a]
```

```
False
```

### Osservazione:

```
NumberQ[ $\sqrt{2}$ ]
```

```
False
```

poiché Mathematica interpreta  $\sqrt{2}$  alla stregua di un simbolo e non di un numero.

Altre funzioni che operano un controllo sugli argomenti sono:

**IntegerQ[x]** controlla se  $x$  è un intero

```
IntegerQ[2]
```

```
True
```

```
IntegerQ[2.]
```

```
False
```

**EvenQ[x]** e **OddQ[x]** controllano la parità di  $x$

```
EvenQ[4]
```

```
True
```

```
EvenQ[3]
```

```
False
```

```
OddQ[4]
```

```
False
```

```
OddQ[3]
```

```
True
```

**PrimeQ[x]** controlla se  $x$  è numero primo

```
PrimeQ[19]
```

```
True
```

```
PrimeQ[9]
```

```
False
```

**SameQ[x,y]** controlla se  $x$  e  $y$  sono identici

```
SameQ[2, 2]
```

```
True
```

```
SameQ[2, 3]
```

```
False
```

**UnSameQ[x,y]** è la funzione opposta di **SameQ[x,y]**

```
UnsameQ[2, 4]
```

```
True
```

Osserviamo che **SameQ[]** e **UnSame[]** sono applicabili anche ad espressioni.

Ad esempio:

```
SameQ[Sin[x], Sin[x]]
```

```
True
```

Una sintassi alternativa per **SameQ[]** è **x===y**. Per **UnsameQ[]** invece **x!=y**

```
2 === 2
```

```
True
```

```
2 === 3
```

```
False
```

```
2 != 3
```

```
True
```